ISSN 0190-3918
also listed under
IEEE Catalog Number 82CH1794-7
Library of Congress Number 79-640377
IEEE Computer Society Order No. 421

INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING

PROCEEDINGS

OF THE

1982 INTERNATIONAL CONFERENCE

ON

PARALLEL PROCESSING

August 24–27, 1982

Kenneth E. Batcher, Willard C. Meilander, and Jerry L. Potter
Editors

Co-Sponsored by

Department of Computer and Information Science
OHIO STATE UNIVERSITY
Columbus, Ohio

and the

IEEE Computer Society

In Cooperation with the

Association for Computing Machinery

IEEE COMPUTER SOCIETY PRESS

1982 INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING
Kenneth E. Batcher, Willard C. Meilander, and Jerry L. Potter

ISSN 0190-3918
IEEE Catalog Number 82CH1794-7
Library of Congress Number 79-640377
IEEE Computer Society Order No. 421
IEEE COMPUTER SOCIETY PRESS

PROCEEDINGS

OF THE

1982 INTERNATIONAL CONFERENCE

ON

PARALLEL PROCESSING

August 24-27, 1982

Kenneth E. Batcher, Willard C. Meilander, and Jerry L. Potter
Editors

Co-Sponsored by

Department of Computer and Information Science
OHIO STATE UNIVERSITY
Columbus, Ohio

and the

IEEE Computer Society

In Cooperation with the

Association for Computing Machinery

ISSN 0190-3918 (also listed under)
IEEE Catalog Number 82CH1794-7
Library of Congress Number 79-640377
IEEE Computer Sociely Order No. 421

International Conference on Parallel Processing.
Proceedings of the International Conference on Parallel Processing. 1972–
[New York, Institute of Electrical and Electronics Engineers; available from the IEEE Computer Society]

v. ill. 29 cm. annual.

Title varies slightly.
Conferences for 1972– co-sponsored by the Dept. of Electrical and Computer Engineering, Wayne State University, Detroit, and the IEEE Computer Society in cooperation with the Association for Computing Machinery.
Key title: Proceedings of the International Conference on Parallel Processing, ISSN 0190-3918.

1. Parallel processing (Electronic computers) I. Institute of Electrical and Electronics Engineers. II. Wayne State University, Detroit. Dept. of Electrical and Computer Engineering. III. IEEE Computer Society. IV. Association for Computing Machinery. V. Title.

QA76.6.I548a 001.6'4 79-640377
MARC-S

Library of Congress 79

Published by IEEE Computer Society Press
1109 Spring Street
Silver Spring, MD 20910

ISSN 0190-3918
IEEE Catalog No. 82CH1794-7
Library of Congress No. 79-640377
IEEE Computer Society Order No. 421

Order from: IEEE Computer Society
Post Office Box 80452
Worldway Postal Center
Los Angeles, CA 90080

IEEE Service Center
445 Hoes Lane
Piscataway, NJ 08854

The Institute of Electrical and Electronics Engineers, Inc.

PREFACE

This, the eleventh conference devoted to parallel processing, marks the beginning of our second decade. The history of parallel processing and what transpired during the first decade were discussed at the keynote session of the 1981 conference. It is appropriate at this year's keynote to learn about a development which will have a major impact during the second decade - VHSIC, a program of the Department of Defense to develop very dense VLSI chips. This effort will also improve the capability of several integrated circuit manufacturers. While denser VLSI will let us build more powerful parallel processors it raises the question of how best to use this new capability.

As in previous years, the conference received many papers from around the world. Of the 124 papers submitted, 46 came from 14 countries outside of the United States. There were many papers of very high quality - far too many to be accomodated at the conference. Final selection of the 67 contributed papers to be presented was very difficult. To fit all these papers into the schedule we were forced to ask the authors of twenty regular papers to condense their material to our short paper format. We regret that we could not accept more papers and that we had to have a number of them trimmed down. Attendees at previous conferences have indicated a preference for maintaining our tradition of no parallel sessions so the schedule is tight and we can only accomodate a limited number of papers and give them a limited amount of time. The conference benefits greatly from this intense competition. We sincerely thank the authors of all submissions for their time and effort.

We owe a deep debt of gratitude to the 153 referees who took time out from their normal duties to evaluate the manuscripts we sent them and give us their opinions. The job of selecting papers would have been impossible without their help.

The program committee thanks Goodyear Aerospace Corporation and Kent State University for their cooperation and support of our committee work. A number of individuals helped us with our work including Lynne Brocco, Hazeljean Cheeseman, Bob Cronauer, Pat Hawkins, Carl Mickelson, Martha Moffat, Jan Pavkov, Carole Rey, and Elizabeth Young. We also extend thanks to the mail service at Goodyear Aerospace for ably handling the extra load we gave them (we received and dispatched over 1000 pieces of mail in connection with our work).

Kenneth E. Batcher - Goodyear Aerospace
Willard C. Meilander - Goodyear Aerospace
Jerry L. Potter - Kent State University
1982 ICPP Program Committee

TABLE OF CONTENTS

SESSION 5: NETWORK DIAGNOSIS AND FAULT TOLERANCE

SESSION 6: DATAFLOW AND REDUCTION MACHINES

SESSION 7: LANGUAGES

SESSION 8: NON-NUMERIC ALGORITHMS I

SESSION 9: NON-NUMERIC ALGORITHMS II

SESSION 10: LARGE-SCALE SCIENTIFIC PROCESSING

SESSION 11: ARRAY PROCESSORS

SESSION 12: MIMD PROCESSING

AUTHOR INDEX

LIST OF REFEREES

George B. Adams III
Siroos K. Afshar
Dharma P. Agrawal
A. P. Andrews
Arvind
Venkataraman Ashok
D. E. Atkins
Jean-Loup Baer
Utpal Banerjee
Helmut K. Berg
S. Ya. Berkovich
Bruce Berra
A. T. Berztiss
Lubomir Bic
Jeffrey G. Bonar
Andrew Boughton
Fayé A. Briggs
Edward C. Bronson
J. C. Browne
Forbes J. Burkowski
Bill Buzbee
F. G. Carty
Tung-Liang Chang
Francis Chin
Thomas W. Christopher
Ronald Cytron
A. L. Davis
Eliezer Dekel
Sudharsan Dhall
Daniel M. Dias
Nikitas Dimopoulos
Michel Dubois
Douglas D. Dunlop
Milos D. Ercegovac
Martha Evens
Bahaa W. Fam
James Fawcett
P. David Fisher
Gary N. Fostel
Garth H. Foster
Caxton C. Foster
Deborah Franke
Mark A. Franklin
Paul Fredrickson
Martin Freeman
Daniel D. Gajski
Dennis Gannon
Mario Gerla
Barry K. Gilbert
Paul A. Gilmore
Manio J. Gonzalez
Robert Gordon
John J. Grefenstette
William I. Grosky
Amar Gupta
Hossam El Halabi
Kye S. Hedlund
Robert Hiromoto
Eugene B. Hogenauer
L. A. Hollaar
Peter Yan Tek Hsu
Kai Hwang
Richard C. Jaeger
Ramesh Jain
Steven F. Jennings
Harry F. Jordan
Rajan Kapur
Svetlana P. Kartashev
Robert M. Keller
F. H. Kierstead
John C. Knight
Hideaki Kobayashi
Harvey S. Koch
Man C. Kong
Janusz S. Kowalik
Clyde P. Kruskal
Annette Krygiel
H. T. Kung
S. Lakshmivarahan
Duncan Lawrie
Kyungsook Yoon Lee
Dennis Leinbaugh
Steven P. Levitan
Sigurd L. Lillevik
Willie Lim
Huai-An Lin
M. W. Linder
G. Jack Lipovski
J. W. S. Liu
K. Y. Liu
B. D. Lubachevsky
Joon Maeng
Gary K. Maki
Miroslaw Malek
P. N. Marinos
W. M. McCormack
Robert J. McMillen
Leslie Miller
Jack Minker
Jamshed H. Mirza
Robert K. Montoye
Amar Mukhopadhyay
W. D. Murphy
Lionel M. Ni
A. A. Nilsson
R. Y. Oh
Arthur E. Oldehoeft
Eli Opper
James M. Ortega
Fusun Ozguner
M. Tamer Ozsu
Janak H. Patel
Daniel J. Pease
Terrence W. Pratt
Robert W. Priester
C. S. Raghavendra
Malcolm Railey
Kamesh Ramakrishna
Jayashree Ramanathan
Anthony P. Reeves
Robert G. Reynolds
Sartaj Sahni
A. H. Sameh
David H. Schaefer
Michael A. Shanblatt
John P. Shen
Kang G. Shin
Daniel P. Siewiorek
Michael L. Skinner
Richard B. Smith
Seshadri Sowrirajan
John A. Stankovic
William J. Stewart
N. C. Strole
Philip H. Swain
Sowmitri Swamy
Mark Taylor
Hoo-Min D. Toong
Wing N. Toy
Vito A. Trujillo
Leonard Uhr
P. S. Wang
Donald F. Wann
Robert G. Wedig
Yiu-min Wei
Bruce W. Weide
Terry A. Welch
Jack C. Wileden
Chuan-lin Wu
W. C. Yen
Pen-Chung Yew
Mark Yoder
Matthew Yuschik

THE VHSIC PROGRAM AND ITS IMPACT ON PARALLEL PROCESSING

Dr. D. W. Burlage
Acting Deputy Director
VHSIC Program, OUSDRE
Department of Defense

ABSTRACT

The VHSIC Program is now approaching the midpoint of its Phase I effort that involves establishment of pilot line capabilities for 28 complex silicon chips employing 1.25 micrometer or smaller feature size processing. These chips, which will be applied in system brassboards for electro-optical, communication, acoustical, missile guidance, electronic warfare, and radar signal processing functions, are amenable to highly parallel system architectures. With one of the chip sets, for example, two chip types are employed to provide a system with 32 parallel processors, each with more than 20,000 gates, to obtain a system capable of several billion operations per second. In effecting these architectures, it is becoming apparent that new combinations of skills and technologies are required to realize the potential of this new generation of VLSI, and that the greatest challenges are now in system design, not in device technology.

DESIGN AND PERFORMANCE OF A GENERAL CLASS OF INTERCONNECTION NETWORKS

Laxmi N. Bhuyan
Department of Electrical Engineering
University of Manitoba, Winnipeg
Manitoba, Canada R3T 2N2

Dharma P. Agrawal
Department of Electrical Engineering
North Carolina State University
Raleigh, NC 27650

ABSTRACT

This paper introduces a general class of self routing Interconnection Networks for tightly coupled multiprocessor systems. The proposed network called "Radix Shuffle Network (RSN)" is based on a new interconnection pattern called Radix Shuffle and is capable of connecting any number of processors M to any number of memory modules N. The technique results in a variety of Interconnection Networks depending on how M and N are factorized. The network covers a broad spectrum of interconnections starting from shared bus to crossbar switches and various Multistage Interconnection Networks (MINs). The permutation capabilities of such networks are outlined. The performance of the networks with respect to their Bandwidth and cost is analyzed and compared with that of a crossbar. Design procedures for obtaining an optimal network with highest cost efficiency is also presented.

I. INTRODUCTION

The performance of a tightly coupled multiprocessor system rests primarily on an efficient design of the network interconnecting the processors to the memory modules. A crossbar switch [1] allows all possible one to one connections between the processors and memories but, the cost grows rapidly with increase in the network size. As an alternative to crossbar, Multistage Interconnection Network (MINs), both nonblocking and blocking have assumed paramount importance in recent times [2-11]. A MIN is basically a blocking network which does not allow all possible permutations but is far less expensive compared to a crossbar switch. A conflict arises when two or more processors need the same link between two successive stages in reaching their destinations. Due to this interference, a subset of processors might be blocked thus giving a degradation in performance. Band Width (BW) of a network is defined as the expected number of memory modules remaining busy in a cycle or the number of memory requests accepted per cycle. Clearly, this is a parameter which specifies to what extent a network is efficient. In a crossbar, all the memory requests are accepted as long as no two or more processors address the memory module. In a random mode of request, the memory BW of even a crossbar is much less than the actual capacity [12]. In a MIN, this value ought to be still less because of additional conflicts in the network. The interference analysis of such networks have been reported in a few papers recently [7,13,14].

The usual design of a MIN employs 2x2 switching elements. However, with the advancement in LSI technology, it might be better to employ a larger module if the network performance could be improved. It is also known that for a crossbar the BW increases with increase in the number of memory modules [12]. So a study on the design of MxN MIN with N>M seems appropriate. Patel's Delta network [7] is a logical approach in this direction. Delta network is a self routing (digit controlled) network connecting a^n inputs to b^n outputs through axb crossbar switches at each stage. Networks like Omega [4], Indirect binary n-cube [6], Baseline [8] etc. form a subset of Delta networks with a=b=2.

This paper presents a still broader class of networks called Radix Shuffle Network (RSN). It connects M inputs to N outputs, for any arbitrary values of M and N. As a result, the existing general network like Delta becomes a special case of the proposed RSN. A Mx1 RSN represents a shared bus multiprocessor system and a single stage MxN RSN represents a crossbar switch. Although several networks can be obtained by constructing de-multiplexer trees [7] from inputs to the outputs, a new interconnection pattern called "Radix shuffle" will be considered throughout this paper.

II. A MIXED RADIX NUMBER SYSTEM

Let M be a decimal number and be represented as a product of r factors as

$$M = m_1 \times m_2 \times \ldots \times m_r$$

Then, each number x between 0 to M-1 can be expressed as a r-tuple $(x_1\ x_2\ \ldots\ x_i\ \ldots\ x_r)$ for $0 \leq x_i \leq (m_i-1)$. x_r is the least significant digit and x_1 is the most significant digit. Associated with each x_i is a weight w_i such that

$$\sum_{i=1}^{r} x_i w_i = x \text{ and } w_i = \frac{M}{m_1 m_2 \ldots m_i} \text{ for all } 1 \leq i \leq r.$$

$$w_r = \frac{M}{m_1 m_2 \ldots m_r} = 1 \text{ always}$$

The lowest number 0_{10} = (0 0 ... 0) and the highest number $(M-1)_{10} = (m_1-1,\ m_2-1, \ldots, m_r-1)$.

Whenever needed, a number x will be represented as $(x)_{m_1,m_2,\ldots m_r}$ to specify the radix involved.

Example 1. Let M = 6 = 3 x 2

$m_1 = 3,\ m_2 = 2;\ w_1 = 2,\ w_2 = 1$

$0_{10} = (00),\ 1_{10} = (01),\ 2_{10} = (10)$

$3_{10} = (11),\ 4_{10} = (20),\ 5_{10} = (21)$

0190-3918/82/0000/0002$00.75 © 1982 IEEE

This mixed radix number system forms the basis of our interconnection. Although the same radix system was used for Omega Networks of Lawrie [5], there are two basic differences between the proposed RSN and Omega. First, RSN is a MxN network for any arbitrary values of M and N as against a NxN Omega Network. Secondly, the interconnection pattern between two stages of our RSN is completely different and is based on a new term "Radix Shuffle" as defined below.

Definition 1 In the above mixed radix system, the radix shuffle of a number $x = (x_1x_2\ldots x_r)_{m_1, m_2,\ldots,m_r}$ will be defined as $S_{rx} = (x_2x_3\ldots x_rx_1)_{m_2, m_3,\ldots,m_r,m_1}$.

Example 2 For M = 3 x 4, the numbers are represented from 00 to 23. Any $(x_1x_2)_{3,4}$ will be connected to $S_{rx} = (x_2x_1)_{4,3}$ as shown in Fig. 1. The connection procedure is as follows.

Number the inputs in radix (3,4) and outputs in (4,3). Make a perfect shuffle of the input and connect to the particular output.

Definition 2 S_m, a m-shuffle of an integer x is given by,

$$S_m = mx \mod(M-1) \text{ for } 0 \leq x < M-1$$
$$= x \text{ for } x = M-1$$

As the example, Fig. 1 again shows a 3-shuffle of 12 inputs. There is a definite relationship between "radix shuffle" and m-shuffle as stated in the following theorem.

Theorem 1 A radix shuffle of $(x)_{m_1, m_2,\ldots,m_r}$ is identical to a m_1-shuffle of x.

Proof of above theorem is omitted because of space restriction.

III. A RADIX SHUFFLE NETWORK (RSN)

Let M and N be represented as products of r-terms as $M=m_1xm_2x\ldots xm_r$ and $N = n_1xn_2x\ldots xn_r$. A MxN RSN with M inputs and N outputs is a r-stage Interconnection Network, consisting of a few crossbar switches of size (m_ixn_i) at the ith stage for all $1 \leq i \leq r$. The inputs and outputs are numbered with base $(m_1,m_2,\ldots,m_r)$ and base $(n_1, n_2,\ldots,n_r)$ respectively in the mixed radix number system. The switches at stage 'i' will be set as per the ith digit of the destination tag. When either M or N is a prime number the RSN reduces to a MxN crossbar switch.

Let M_i and N_i indicate the number of inputs and outputs at the ith stage of RSN. The first stage will consist of $(\frac{M}{m_1})$ number of (m_1xn_1) crossbar switches producing $N_1 = \frac{M}{m_1}\cdot n_1$ outputs with $M_1 = M$. The second stage will have $\frac{N_1}{m_2} = \frac{M\cdot n_1}{m_1 m_2}$ number of (m_2xn_2) crossbar switches producing $N_2 = \frac{Mn_1n_2}{m_1 m_2}$ outputs. In general, the ith stage will consist of $\frac{Mn_1n_2\ldots n_{i-1}}{m_1m_2\ldots\ldots m_i}$ switches of size (m_ixn_i) each and will produce $N_i = \frac{Mn_1n_2\ldots n_i}{m_1m_2\ldots m_i}$ outputs.

The rth or the final stage will have $\frac{Mn_1n_2\ldots n_{r-1}}{m_1m_2\ldots\ldots m_r} = \frac{N}{n_r}$ number of (m_rxn_r) crossbar switches producing N_r=N outputs.

Demultiplexer trees can be drawn from a particular input to all outputs for full connectivity and the overlap of such M trees will give rise to a self routing Interconnection network. However the interconnection pattern Radix shuffle is of interest throughout this paper. The ith stage of RSN will be preceded by a m_i-shuffle (also radix shuffle) for all $1 \leq i \leq r$, as shown in Fig. 2.

Let us consider the interconnection pattern in some more detail. The M inputs are numbered in base $(m_1,m_2,\ldots,m_r)$. The first stage of switches will be preceded by a radix shuffle which is m_1 shuffle in this case. The inputs to the first stage are numbered in base $(m_2,m_3,\ldots,m_r,m_1)$ following the radix shuffle. The outputs of the first stage of switches of size (m_1xn_1) will be numbered in base $(m_2,m_3,\ldots,m_r,n_1)$. The inputs to the second stage of switches will be numbered in base $(m_3,m_4,\ldots,m_r,n_1,m_2)$ following a radix shuffle interconnection. The outputs of the second stage of switches will be numbered in base $(m_3,m_4,\ldots,m_r,n_1,n_2)$ and so on. We have the following theorem.

Theorem 2 The M-input N-output Radix Shuffle Network constructed as above is indeed self routing.

Proof Let the source $S = (s_1s_2\ldots s_r)_{m_1,m_2,\ldots,m_r}$ be desired to be connected to the destination $D=(d_1d_2\ldots d_r)_{n_1,n_2,\ldots,n_r}$. After the first stage of Radix shuffle, the source converts to $(s_2s_3\ldots s_rs_1)_{m_2,m_3,\ldots,m_r,m_1}$ at the input of the 1st stage of switches. The particular (m_1xn_1) switch at the 1st stage will connect the source to the output $(s_2s_3\ldots s_rd_1)_{m_2,m_3,\ldots,m_r,n_1}$ depending on the destination digit d_1. After the second radix

shuffle it becomes $(s_3 s_4 \ldots s_r d_1 s_2)_{m_3, m_4, \ldots, m_r, n_1}$, m_2 at the input of the 2nd stage of switches and $(s_3 s_4 \ldots s_r d_1 d_2)_{m_3, m_4, \ldots, m_r, n_1, n_2}$ at the output. At the output of the ith stage of switches it becomes

$$(s_{i+1}\ s_{i+2} \ldots s_r\ d_1\ d_2 \ldots d_i)_{m_{i+1}, m_{i+2}, \ldots, m_r, n_1, n_2, \ldots, n_i}.$$

After the rth stage of switches the source S is connected to $D = (d_1\ d_2 \ldots d_r)_{n_1, n_2, \ldots, n_r}$. Since the mixed radix system is unique, there exists a unique path between any input and an output.

Q.E.D.

Example 3 Let M = 6 = 3 x 2 and N = 8 = 4 x 2

$m_1 = 3$, $m_2 = 2$ and $n_1 = 4$, $n_2 = 2$.

The RSN consists of two (3x4) crossbar switches in the 1st stage and four (2x2) switches at the second stage as shown in Fig. 3. The inputs are numbered in base (3,2) and the outputs in base (4,2). The 1st stage of interconnection is a radix shuffle of base (3,2) giving rise to a 3-shuffle. The inputs to the 1st stage of switches are numbered in base (2,3). The outputs of the 1st stage are numbered in base (2,4). The inputs to the 2nd stage of switches are numbered in base (4,2) following the 2-shuffle interconnection. Finally, the outputs are in base (4,2). The connection between input 3 = $(11)_{3,2}$ and output 1=$(01)_{4,2}$ is shown with dark line in Fig. 3.

The RSN is self routing in the sense that the connection in a $(m_i x n_i)$ crossbar module at the ith stage is controlled by the ith digit d_i of the desired output, $0 \leq d_i \leq n_i - 1$. When all m_i's are equal to a and all n_i's are equal to b, the RSN reduces to a $a^n x b^n$ Delta network [7]. The mixed radix system becomes a simple higher radix system. The interconnection before stage 'i' becomes m_i = a-shuffle for all $1 \leq i \leq r$. The first stage of the interconnection in RSN allows the identity permutation. With r=1, any MxN RSN is equivalent to a crossbar switch.

When N=1, M number of processors share a common memory through a Mx1 switch. This is equivalent to a shared bus system. Although different interconnection networks can be obtained by constructing Demultiplexer trees from input to output, they are all equivalent in terms of total number of permutations, Bandwidth, probability of acceptance etc. The Radix shuffle is just a convenient and useful way of interconnection. The Multistage Interconnection Networks (MIN) such as Omega [4], Inelirect Binary n-cube [6], Baseline [8] etc. employing 2x2 switches are essentially a part of our RSN with $M=N=2^n$.

IV. PERMUTATION CAPABILITIES OF RSN

Let the capacity C be defined as the maximum number of simultaneous input-output connections that can be achieved through a network. For a MxN crossbar C = Min{M,N} [15]. In a NxN multistage interconnection network, although some permutations are not possible still the capacity remains equal to N. In a RSN, the capacity is upper bounded by the minimum number of inputs/outputs at any stage. For example, in a 6x8 RSN with M=6=3x2 and N=8=2x4, the number of outputs from first stage N_1=4 which is even less than the number of processors. As a result no more than 4 processors can be simultaneously connected to the output.

Lemma 1: In a RSN the capacity C is upperbounded by $\text{Min}\{M, N_i \mid 1 \leq i \leq r\}$.

In a single mxn crossbar, the capacity being min{m,n}, it is worthwhile looking into how many possibilities of connections exist. For example, if $m \leq n$, all inputs can be simultaneously connected to m out of n outputs provided no two or more inputs address the same memory module. In a nxn crossbar n! such different mappings or permutations are possible. In a mxn crossbar for $m \leq n$, there can be $\binom{n}{m}$ combinations of choosing m numbers out of n memory modules. Associated with each combination, there are m! permutations. The following lemma results:

Lemma 2 The number of permutations achievable by a $m_i x n_i$ crossbar module at the ith stage of RSN is given by

$$s_i = \binom{n_i}{m_i} m_i! \quad \text{for} \quad m_i \leq n_i$$

$$= \binom{m_i}{n_i} n_i! \quad \text{for} \quad m_i \geq n_i$$

Theorem 3 If RSN is obtained such that $\forall\ 1 \leq i \leq r$, $M_i \geq M_{i+1} = N_i$ for $M \geq N$ and $M_i \leq M_{i+1} = N_i$ for $M \leq N$, the total number of permutations achievable is

$$P = \prod_{i=1}^{r} s_i^{k_i},$$

where k_i is the number of switches at the ith stage and s_i is as given by lemma 2.

The proofs of the above lemmas and theorem are omitted because of space limitation.

A conflict is said to occur in a network when two or more sources require the same link between two stages for reaching their destinations. For example, in Fig. 3, the connections $0 \rightarrow 4$ and $2 \rightarrow 5$ require the same link and cannot be simultaneously achieved. In case of conflicts, the connections are usually achieved in two or more cycles. The following theorem characterizes the conflict situation in a RSN.

Theorem 4 In a RSN, there occurs a conflict if at least two sources s_x, s_y try to reach destinations d_x,d_y such that for $1 \leq i \leq r$,

$$(d_1 d_2 \cdots d_i)_x \Leftrightarrow (d_1 d_2 \cdots d_i)_y$$

$$(s_{i+1} s_{i+2} \cdots s_r)_x = (s_{i+1} s_{i+2} \cdots s_r)_y$$

Proof s_x and s_y are in base $(m_1, m_2, \ldots, m_r)$. d_x and d_y are in base $(n_1, n_2, \ldots, n_r)$.

From theorem 2, a conflict will occur at the output of ith stage if $(s_{i+1} s_{i+2} \cdots s_r \; d_1 d_2 \cdots d_i)_x = (s_{i+1} s_{i+2} \cdots s_r \; d_1 d_2 \cdots d_i)_y$.

Since both are in base $(m_{i+1}, m_{i+2}, \cdots m_r, n_1, n_2, \ldots n_i)$ which is a mixed radix system with unique representation of a number, the theorem holds. Q.E.D.

V. COST MODEL OF RSN

Before modeling RSN, it is imperative to look into the complexity of a self routing mxn crossbar module which forms the basic block in RSN. At the ith stage, the $m_i x n_i$ crossbar should be able to connect any one of its m_i inputs to any one of its n_i outputs as determined by the ith digit d_i of the destination tag. This would necessiate $\lceil \log_2 n \rceil$ control lines from each processor. In addition, there may be one request and another acknowledge line from each crossbar module. The control unit inside the switch will decode this address and connect the data lines of the particular processor to one of the output data lines. This will be achieved through a bi-directional data switch available in the crossbar module. The number of data lines will depend on the pattern of data transfer. In a serial data transfer, there will be only one line per processor. It may also be practical to transmit data in a bit slice mode with some 'w' bits per processor. The block diagram of a mxn switching element is shown in Fig. 4a. The complexity of the control unit as well as the data switch grow of the order of (mn). Assuming unit cost for a one input/ one output switch, the cost of a mxn switch = mn. The model results in a crossbar with mn cross points as shown in Fig. 4b. For a 2x2 switching element, the cost is 4 units. For a MIN employing $\log_2 N$ stages the total cost = $4 \times \frac{N}{2} \times \log_2 N = 2N\log_2 N = O(N\log_2 N)$. For a NxN crossbar, cost = N^2. This modeling is in agreement with the model developed in [5] in terms of the logic gates.

A RSN employs $\frac{Mn_1 n_2 \cdots n_{i-1}}{m_1 m_2 \cdots m_i}$ number of $(m_i x n_i)$ switching modules at the ith stage. The cost associated with the ith stage = $\frac{Mn_1 n_2 \cdots n_i}{m_1 m_2 \cdot m_{i-1}}$. Hence, the total cost of the RSN = $\sum_{i=1}^{r} \frac{Mn_1 n_2 \cdots n_i}{m_1 m_2 \cdots m_{i-1}}$.

A special case of interest is a NxN RSN where $m_i = n_i$, $\forall\ 1 \le i \le r$. The total cost is

$$N \sum_{i=1}^{r} n_i .$$

We get the following results.

Lemma 3 The cost of a NxN RSN for some fixed r stages of switching elements is minimum when realized as $N = n^r$.

Theorem 5 The overall cost of a NxN RSN is absolute minimum when realized with all the factors of N as prime numbers.

The proofs of the above lemma and theorem are obvious and hence, have been omitted.

VI. ANALYSIS OF RSN

In this section, we will analyze the RSN with respect to its Bandwidth and Probability of acceptance of a request and compare it with those of a crossbar. Bandwidth (BW) is defined as the expected number of memory requests accepted per cycle. Probability of acceptance (PA) is the ratio of expected BW to the expected number of requests generated per cycle. The RSN and crossbar will be analyzed under the following identical assumptions.

1. The operation is synchronous i.e. the messages begin and end simultaneously.

2. Each processor generates a random and independent request. The requests are uniformly distributed over all the memory modules .

3. At the beginning of a cycle, each processor generates a new request with a probability p. Thus p is the average number of requests generated per cycle by each processor.

4. The requests which are not accepted are ignored. The requests issued at the next cycle are independent of the requests of the previous cycle.

Various simulation results indicate that the above set of assumptions does not result in a significant difference in the performance. Moreover, it stands well for comparison purposes.

The BW and PA of a mxn crossbar module are given by [7,12,13].

$$BW = n - n\left(1 - \frac{p}{n}\right)^m$$

$$PA = \frac{n}{p \cdot m} - \frac{n}{p \cdot m}\left(1 - \frac{p}{n}\right)^m .$$

where p is the average number of messages generated per processor per cycle.

The above equations are quite simple and they compare well with the results of simulation [12]. We compared the results of a 2x2 Delta network [7] using the above equations with those reported by Nelson [14]. Patel's analysis shows a better closeness to the simulation results. We will simply use these equations for analysis of RSN instead of pursuing the matter further.

Dividing the bandwidth by n, gives us the rate of requests on any one of the n output lines of a mxn crossbar module, as a function of its input rate

$$p_{out} = 1 - (1 - \frac{p_{in}}{n})^m.$$

In a RSN, the output rate of ith stage is also the input rate to (i+1)th stage. Hence, one can recursively evaluate the output rate of any stage starting with the input rate of the first stage. The output rate of stage r determines the BW of a RSN.

Let p_i be the rate of request at the output of the ith stage. Then

$p_i = 1 - (1 - \frac{p_{i-1}}{n_i})^{m_i}$, p_0 is the rate of requests generated by the processors.

A column Bandwidth (CBW) is the BW at the output of a particular column.

$$CBW_i = p_i \times N_i = \frac{Mn_1n_2\cdots n_{i-1}n_i}{m_1m_2\cdots m_i} p_i$$

The output BW is the CBW at stage r. $BW=N \cdot p_r$.

The probability that a request will be accepted in RSN = $P_A = \frac{Np_r}{Mp_0}$.

VII. NETWORK OPTIMIZATION

In this section, we present some interesting results on how to design a cost effective interconnection network. The BW reflects the performance of a network and a cost model was obtained in section V. We will define a cost factor ξ as the ratio of BW to cost. Since we do not have a closed form solution for BW, most of the results presented in this section are experimental, obtained through computation. We will study the characteristics of both MxN and NxN networks.

Given a value of N for a NxN RSN, there may be several ways to factorize N into r components. As an example, for N=16 and r=2, N can be expressed as 8x2 or 4x4. The obvious question arises, for given values of N and r, what is the optimum realization of a network. This will be referred to as local optimization. The following observation is made.

Conjecture 1 The most cost effective realization of a NxN RSN in some r-stages is obtained when $m_i = n_i = \sqrt[r]{N}$ for all $1 \leq i \leq r$.

The conjecture is obtained from computational results. Let r=2. The cost factor ξ=BW/cost is plotted in Fig. 5 for various values of N, a power of two. The peaks are obtained at a theoretical value of $\sqrt{N}$, even if this is not an integer. A computer search was carried out for a few values of r which resulted $\sqrt[r]{N}$ as the optimal realization. Since $\sqrt[r]{N}$ may not be an integer for a fixed r, the m_i's should be as close to $\sqrt[r]{N}$ as possible.

There is another tradeoff in building a RSN. For example, 16 can be factorized as 2x2x2x2, 4x2x2, 4x4 etc. The design which gives the highest cost factor is optimal. This will be referred to as global optimization of RSN.

It has been impossible to derive a closed form solution for the optimal value of r (r_{opt}). For $N=n^r$ and for values n=2 and 3, the optimal values of r are plotted in Fig. 6. For N, a power of two the following conjecture states the most important observation.

Conjecture 2 For NxN RSN with N a power of two as many 4x4 switches as possible should be employed to yield the most cost effective realization.

From Fig. 6 for a 4x4 network, $r_{opt} = 1$. This means one 4x4 switch should be employed instead of conventional four (2x2) switches. For N=8, r_{opt}=2, thus the realization should be as N=4x2 employing one stage of 4x4 switches and another stage of 2x2 switches. For N=16, two stages of 4x4 switches are desired. So, for N, a power of four, 4x4 switches should be employed at all stages and for N a power of 2 but not a power of 4, the last stage will consist of 2x2 switches and all the previous stages should consist of 4x4 switches.

A study on loosely coupled system (distributed) with a sort of hypercube topology had also resulted in a similar observation [16]. With N, a power of 3, optimal structure is obtained when 3x3 switching elements are utilized. In general for any N, a discrete optimization may be carried out to determine the most effective realization. The networks realized in this manner will be referred to as Optimal RSN (ORSN) in this paper. The BW, PA and cost efficiency obtained in a NxN ORSN for N, a power of two are compared in Figs. 7, 8 and 9 respectively with those of RSN(2) and crossbar. RSN(2) is the MIN obtained with 2x2 switches at each stage which is equivalent to an Omega network.

We will now examine the performance of a MxN RSN. It is known from Bhandarkar's analysis [12] that if the number of memory modules is increased compared to the number of processors, the BW increases. This is evident because, with the availability of more memory modules, less conflicts will occur and more number of processors can be kept busy in a probabilistic view. The variations of BW and cost efficiency by adding more memory modules are plotted in Figs. 10 and 11 respectively. The number of processors is kept constant as M=16. Whenever a few memory modules are added fresh design of the RSN is carried out and the cost efficiency (ξ) is calculated to yield a new ORSN. The performance of ORSN is plotted together with that of a MxN crossbar switch.

The BW of a crossbar increases exponentially and theoretically reaches 16 at N=∞. For ORSN,

the saturation starts earlier and remains constant at about 10. This is because of the conflicts inherent in a RSN. It may be pointed out here that ORSN is designed such that the cost efficiency is at the highest level of all the designs. It was observed that the BW improves if other size of the switches were allowed but this will happen at the expense of cost effectiveness. The crossbar itself is also a part of RSN any way.

A similar experiment was carried out to see the effect of adding a processor to a fixed number of memory modules. Figs. 12 and 13 are obtained with various values of M with N kept fixed at 16. The variation of curves obtained for crossbar can be easily predicted theoretically from equations, $BW = N - N(1 - \frac{1}{N})^M$, $p_o = 1$ and cost efficiency $\xi = \frac{BW}{MN} = \frac{1-(1-\frac{1}{N})^M}{M}$. As $M \to \infty$, BW of crossbar approaches N. However, with a recursive equation for BW in case of RSN and the discrete optimization required, it was not possible to theoretically predict the characteristics of ORSN. The results however seem to be quite realistic.

VIII. CONCLUSIONS

A broad class of networks called Radix Shuffle Networks (RSN) was introduced. The network is self routing in the sense that the output of the switches at ith stage is selected by the ith digit of the destination address. The network is so general that it includes systems ranging from a shared bus connection to a crossbar. The cost modeling was approximate but truly represents the complexity involved. The bandwidth was choosen as a performance measure with the assumption that the cycle time is almost same for all the realizations of RSN including crossbar. Thus, depending on the actual parameters, the cost efficiency curve may shift a little. However, we are convinced that the comparisons reported in the paper will still stand.

The observations indicate that by adding more number of memory modules the bandwidth increases and the RSN provides an efficient design for such an interconnection. Many useful results were presented. An important observation is that a NxN MIN seems to employ (4x4) switches instead of (2x2) switches conceived of so far. When N is a power of two but not a power of four, all the stages can employ (4x4) switching elements except the last one which will employ (2x2) switches. The Radix Shuffle makes that connection possible.

REFERENCES

1. W.A. Wulf and C.G. Bell, "Cmmp - A Multi-miniprocessor," Proc. AFIPS, Fall Joint Computer Confernce, Dec. 1972, pp. 765-777.

2. V.E. Benes, "Mathematical Theory of Connecting Networks and Telephone Traffic," Academic Press, New York, 1965.

3. L.R. Goke and G.J. Lipovski, "Banyan Networks for Partitioning Multiprocessor Systems," Proc. 1st Int. Symp. on Computer Architecture, Florida, Dec. 1973, pp. 21-28.

4 D.H. Lawrie, "Access and Alignment of Data in an Array Processor," IEEE Trans. on Computers, C-24, Dec. 1975, pp. 1145-1155.

5. D.H. Lawrie, "Memory-Processor Connection Networks," Ph.D. Thesis, University of Illinois, 1973.

6. M.C. Pease, "The Indirect Binary N-cube Microprocessor Array," IEEE Trans. on Computers, C-26, May 1977, pp. 458-473.

7. J.H. Patel, "Performance of Processor-Memory Interconnections for Multiprocessors," IEEE Trans. on Computers, C-30, Oct. 1981, pp. 771-780.

8. C.L. Wu and T.Y. Feng, "On a Class of Multistage Interconnection Networks," IEEE Trans. on Computers, C-29, Aug. 1980, pp. 694-702.

9. H.J. Siegel, "Analysis Techniques for SIMD Machine Interconnection Networks and the Effect of Processor Address Masks," IEEE Trans. on Computers, C-26, Feb. 1977, pp. 153-161.

10. D.K. Pradhan and K.L. Kodandapani, "A Uniform Representation of Single and Multistage Interconnection Networks Used in SIMD Machines," IEEE Trans. on Computers, C-29, Sept. 1980, pp. 777-790.

11. M.A. Abidi and D.P. Agrawal, "On Conflict Free Permutations in Multistage Interconnection Network," Journal of Digital Systems, Special Issue on Parallel Processing, Vol. IV, 2, Summer 1980, pp. 115-134.

12. D.P. Bhandarkar, "Analysis of Memory Interference in Multiprocessor," IEEE Trans. on Computers, C-24, Sept. 1975, pp. 897-908.

13. D.M. Dias and J.R. Jump, "Analysis and Simulation of Buffered Delta Networks," IEEE Trans. on Computers, C-30, April 1981, pp. 273-282.

14. S. Thanawastien and V.P. Nelson, "Interference Analysis of Shuffle/ Exchange Network," IEEE Trans. on Computers, C-30, Aug. 1981, pp. 545-556.

15. G.M. Masson, G.C. Gingher and S. Nakamura, "A Sampler of Circuit Switching Networks," Computer, Vol. 12, June 1979, pp. 32-48.

16. L.N. Bhuyan and D.P. Agrawal, "A General Class of Processor Interconnection Strategies," Proc. 9th Int. Symp. on Computer Architecture, April 1982, pp 90-98.

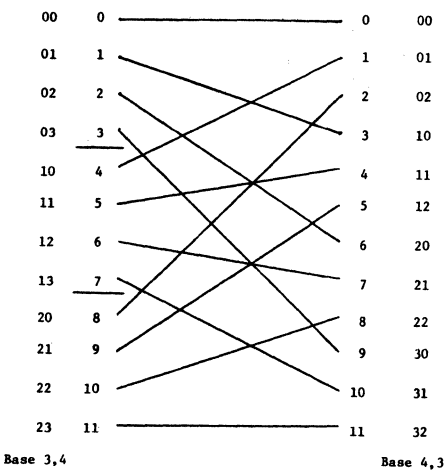


Fig. 1. A Radix Shuffle of N = 3 x 4

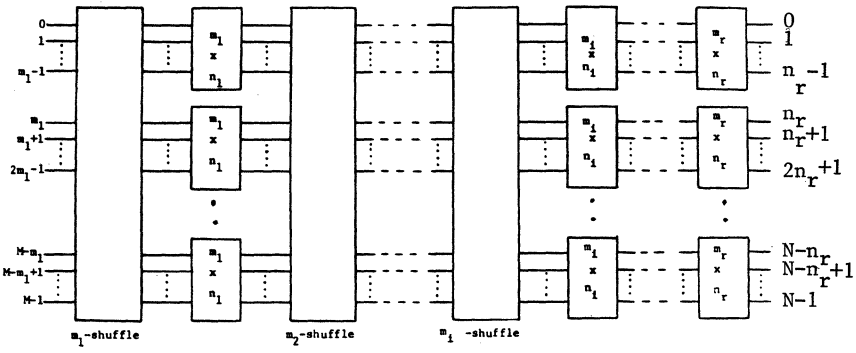


Fig.2. A M x N Radix Shuffle Network.

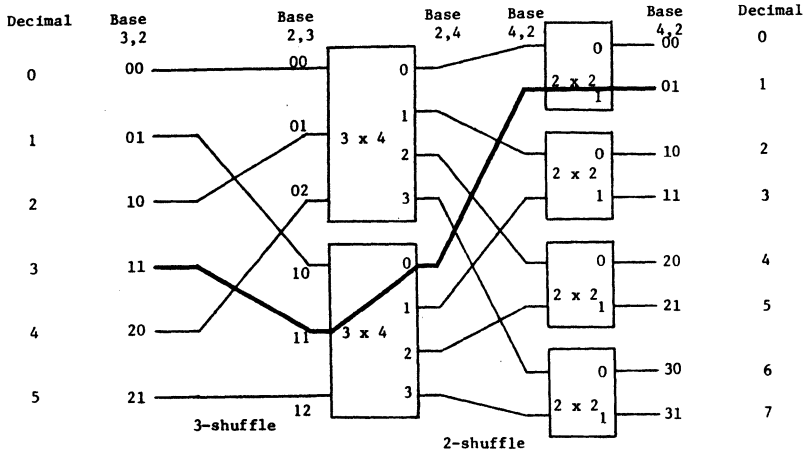


Fig. 3. A 6 x 8 Radix Shuffle Network with 6 = 3 x 2 and 8 = 4 x 2

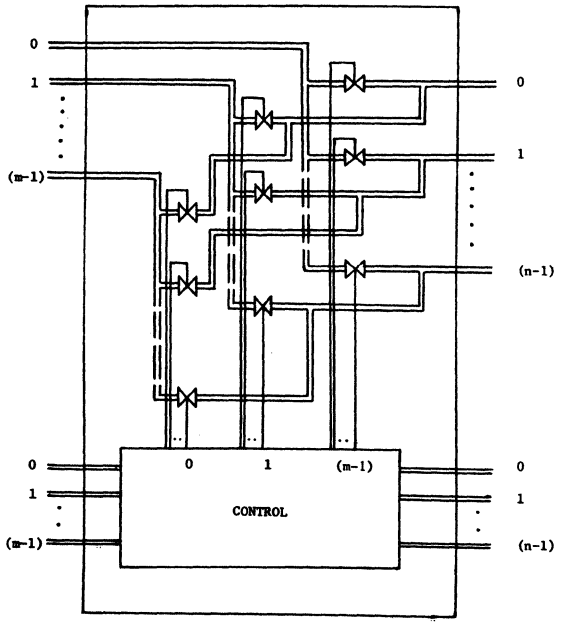


Fig. 4a. Schematic Diagram of a m x n Switch

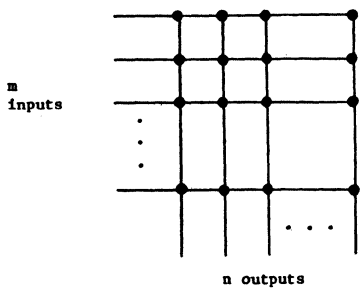


Fig. 4b. Cost Model of a m x n Switch

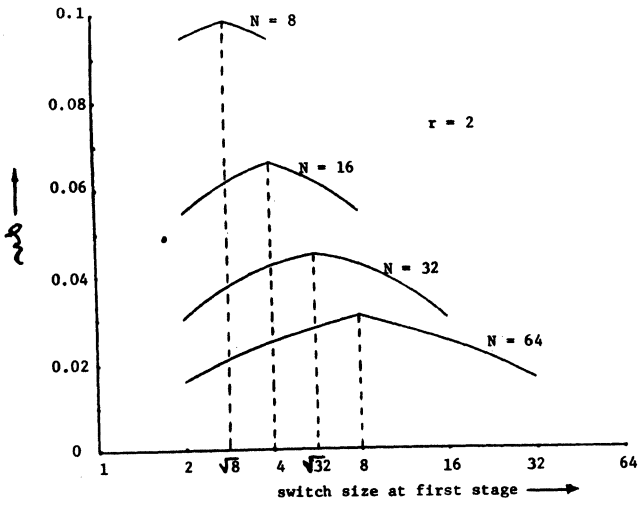


Fig. 5 Cost Efficiency vs. Switch Size at First Stage

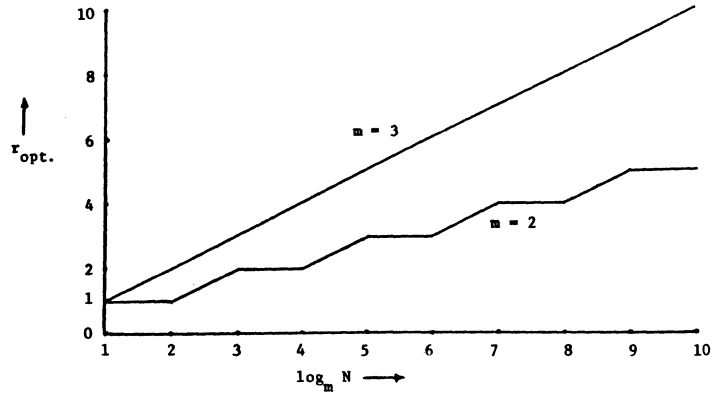


Fig. 6 Value of $r_{opt.}$ for $N = m^r$

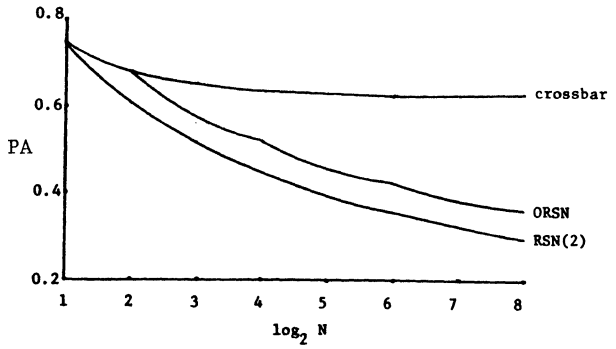


Fig. 8 Probability of Acceptance of N x N Networks

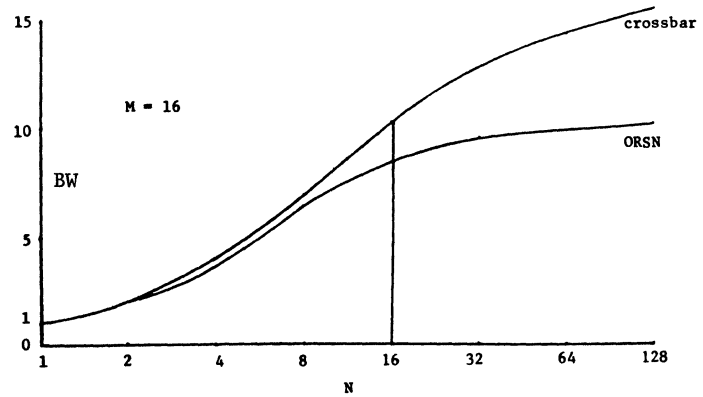
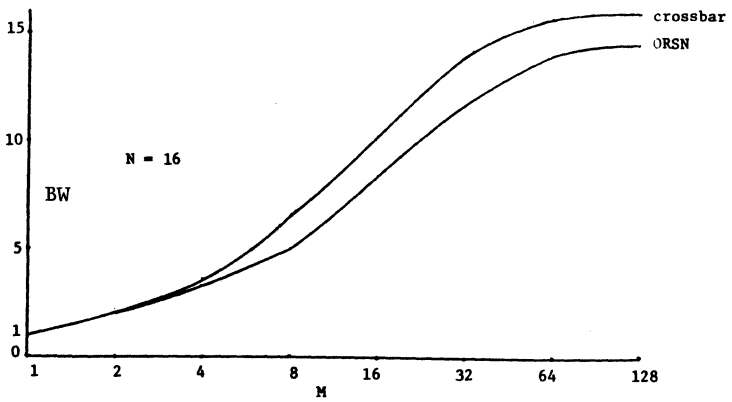


Fig. 10. Effect of Adding a Memory Module on Bandwidth



Fib. 12. Effect of Adding a Processor on Bandwidth

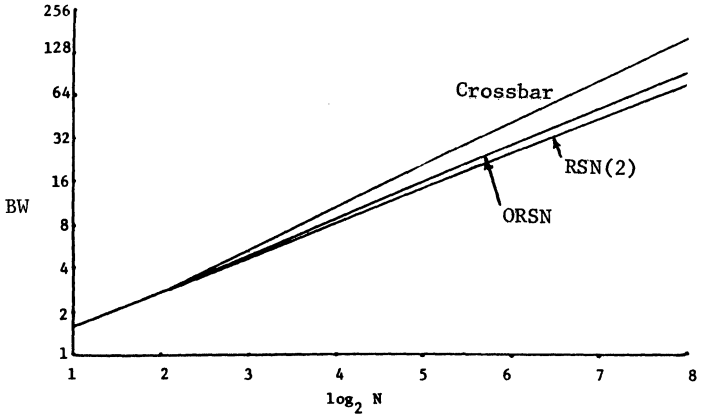


Fig. 7 Bandwidth of N x N Networks

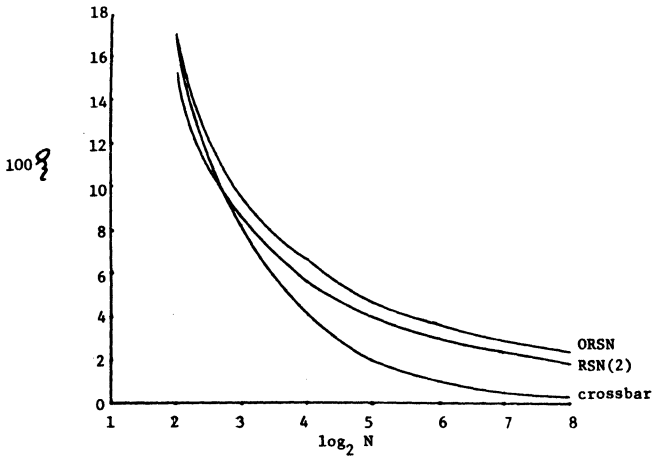


Fig. 9 Cost Effectiveness of N x N Networks

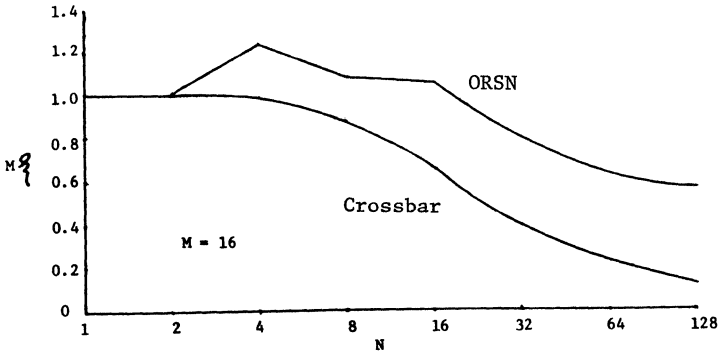


Fig. 11. Effect of Adding a Memory Module on Cost Efficiency

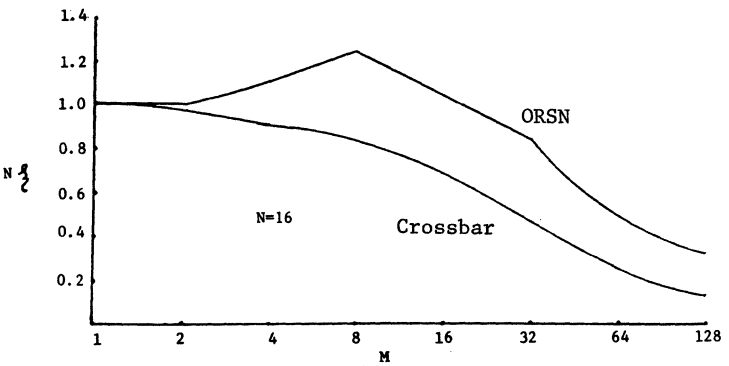


Fig. 13. Effect of Adding a Processor on Cost Efficiency

AUGMENTED AND PRUNED N LOG N MULTISTAGE NETWORKS: TOPOLOGY AND PERFORMANCE

Daniel M. Dias, Mitre Corporation, Houston, Texas
and
J. Robert Jump, Rice University, Houston, Texas

Abstract -- In this paper N log N Multistage Interconnection Networks (MINs) are augmented for better reliability and pruned to have differing numbers of input and output links. Augmented MINs have multiple disjoint paths from network input to output links. Optimal pruned MINs with buffers between stages have a non-intuitive topology.

Summary

Several (NxN) multistage networks with log N stages (referred to as MINs) have been proposed in the literature [1-4]. It can be shown [5] that these networks can be constructed recursively as illustrated in Figure 1. MINs can be used to interconnect modules of a computing system that communicate by passing fixed size packets through the MIN [6]. Each packet contains data and a destination address. At each stage, one digit of the destination address is used to route the packet to the next stage via the link with the same label (see Figure 1). This is referred to as digit controlled routing.

The MINs in [1-4] have a unique path from network input to network output links leading to poor reliability. One method of augmenting MINs is to add an extra stage as illustrated in Figure 2. It can be shown [5] that with a judicious selection of the interconnection to stage S* in Figure 2, there are exactly b disjoint paths (except for the common input and output links) from each network input link to each network output link and digit controlled routing can still be used. An example of a $(2^3 \times 2^3)$ augmented MIN showing the two disjoint paths appears in Figure 3. Additional redundant paths through the network can be obtained by adding further stages.

MINs can be pruned by eliminating some of the switches. This paper considers regular pruned MINs only [5]. A $(b^n \times b^m)$ regular pruned MIN consists of arbitration, distribution and square stages. Networks with n greater than m are called arbitration networks while networks with n less than m are called distribution networks [7]. The construction of arbitration stages is shown in Figure 4. Distribution stages with b^n inputs and b^{n+1} outputs are constructed in an analogous manner. At square stages, the original MIN is left untouched. This paper will present performance results for $(2^n \times 2^m)$ arbitration networks only. There are $\binom{n}{m}$ different ways of choosing arbitration stages in a $(2^n \times 2^n)$ MIN to produce different $(2^n \times 2^m)$ arbitration networks. Extreme arbitration networks, shown in Figure 5, have the arbitration stages concentrated either at the input stages or at the output stages of the network.

Unbuffered networks are modelled as operating in time slots. In each time slot an attempt is made to pass packets at input links to the desired output links. If two packets must pass through the same switch, a conflict is said to occur and one of packets is selected and passed while the other is rejected. The performance results reported here are for the case when rejected packets are discarded. Approximate estimates of performance for the case when rejected packets are retried in the next time slot can be found in [5].

Buffered networks have first-in-first-out buffers of fixed maximum length between stages. The operation of a (2 x 2) switch is modelled essentially as follows [8]. The (minimum) packet delay at a switch is modelled in terms of two timing parameters: time t_select to select the output link to which the packet is to be passed and time t_pass to move the packet through the switch. The t_select phase for two packets at different links can occur simultaneously. However, if two packets (after the t_select phase) are directed to the same switch output link, one is randomly picked for transfer and the other is blocked. Further, the selected packet must wait until a buffer at the input of the successor switch becomes available; after a successor switch buffer becomes available, the packet takes time t_pass to pass through the switch into this buffer.

The throughput is defined informally as the average rate at which packets are put out by the network. The normalized throughput (NTP) is the ratio of the throughput obtained to the maximum possible throughput assuming that no conflicts occur in the network. For unbuffered networks, p_0 is the probability that a packet exists at a network input link in a time slot and p_a is the probability that a packet at a network input buffer is accepted in a time slot. The results reported here were obtained using both event driven simulation and an approximate model with coupled Markov chains [5].

The NTP for augmented MINs are shown in Figures 6 and 7. It is seen that, for fault-free operation, there is a small decrease in the NTP for the augmented MIN as compared to the corresponding MIN. The worst case single internal switch or link failure leads to about one half the fault-free performance. However, for most failures, the performance is only slightly below the fault-free performance.

0190-3918/82/0000/0010$00.75 © 1982 IEEE

For unbuffered arbitration networks, the extreme networks are seen to have extremes in performance as shown in Figure 8. The same is true for buffered arbitration networks if all switches have the same speed (i.e., the maximum packet output rate at a link is the same at all switches). The network stages can be matched so that all stages have the same maximum throughput. This may be done by a serial to parallel transformation at an arbitration stage [5]. Matched arbitration networks with maximum throughput have a non-intuitive topology. Examples of matched arbitration networks with maximum throughputs for a single buffer between stages are shown in Table 1. Here, the arrangement of stages that give the highest normalized throughput is given as a string of A's and S's, where A denotes an arbitration stage and S a square stage.

References

[1] D. H. Lawrie, "Access and Alignment of data in an array processor," IEEE Trans. Comput., Vol.C-24, Dec. 1975.

[2] M. C. Pease, "The Indirect n-Cube Microprocessor Array," IEEE Trans. Comput., Vol.C-24, Dec. 1975.

[3] H. J. Seigel and D. S. Smith, "Study of Multistage SIMD Interconnection Networks," Proc. 5th Annual Symp. Comput. Arch., NY, NY, Apr. 1978.

[4] J. H. Patel, "Processor-Memory Interconnections for Multiprocessor," IEEE Trans. Computers, VolC-30, No. 10, Oct. 1981, pp. 771-780.

[5] D. M. Dias, "Packet Communication in Delta and Related Networks," Ph.D. dissertation, Rice University, Apr. 1981.

[6] D. M. Dias and J. R. Jump, "Packet Switching Interconnection Networks for Modular Systems,", Computer, Vol. 14, No. 12, Dec. 1981, pp. 43-53.

[7] J. B. Dennis and D. P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor," Proc. 2nd Annual Symposium on Computer Architecture, Jan. 1975, pp. 126-132.

[8] D. M. Dias and J. R. Jump, "Analysis and Simulation of Buffered Delta Networks," IEEE Trans. Computers, Vol. C-30, No. 4, Apr. 1981, pp. 273-282.

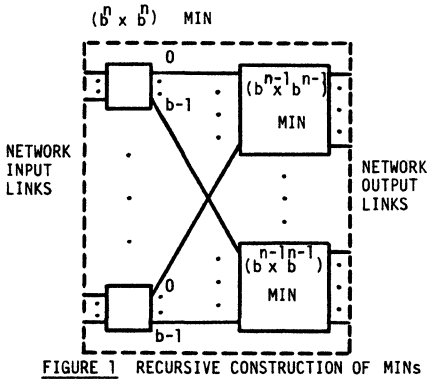


FIGURE 1 RECURSIVE CONSTRUCTION OF MINs

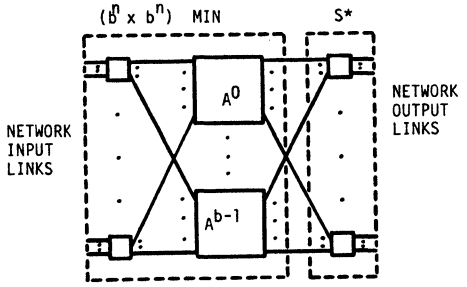


FIGURE 2 Constructing Augmented MINS

NOTE: $A^0 \ldots A$ are $(b^{n-1} \times b^{n-1})$ MINs

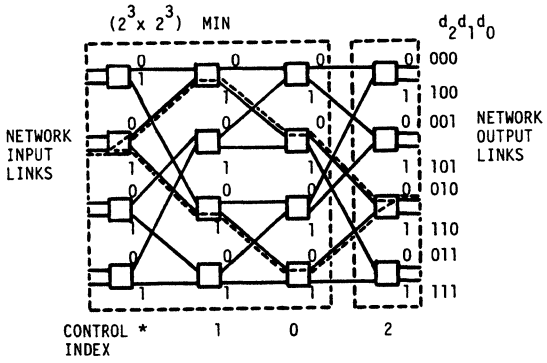


FIGURE 3 An Augmented MIN

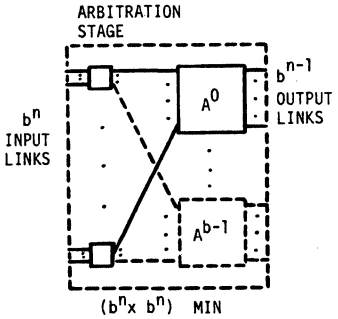


FIGURE 4 Constructing an Arbitration Stage

NOTE: $A^0 \ldots A^{b-1}$ are $(b^{n-1} \times b^{n-1})$ MINs

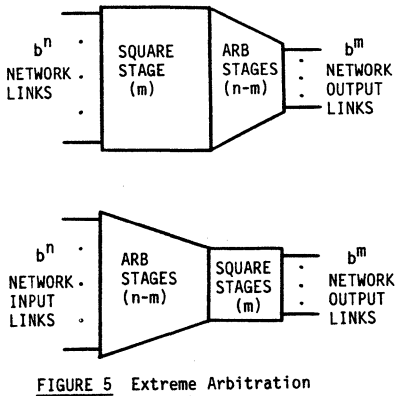


FIGURE 5 Extreme Arbitration Networks

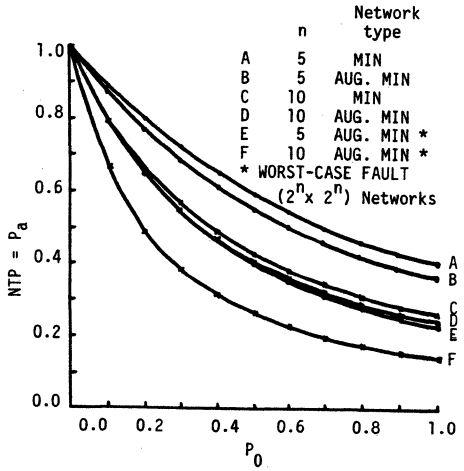


FIGURE 6 NTP versus P_0 for unbuffered augmented MINs, under normal operation and for a worst-case single fault.

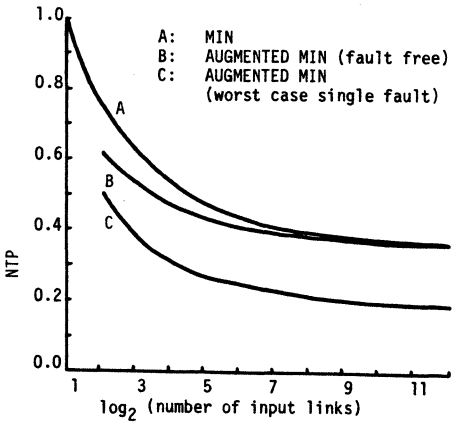


FIGURE 7 NTP versus network size for buffered augmented MINs, under normal operation and for a worst case single fault (Single buffer between stages; t-select = 1.0; t-pass=0.0; packets assumed to be always available at network input links.)

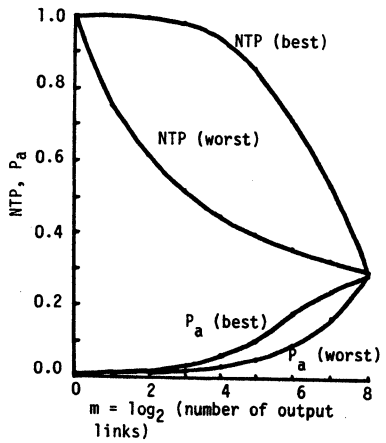


FIGURE 8 NTP and P_a versus the number of output links, for unbuffered arbitration networks (b=2, n=8, p_0=1)

Net-work	t_select=0 Structure	NTP	t_select=1 Structure	NTP
2^2X2^0	AA	0.75	AA	1.00
2^2X2^1	SA	0.56	AS	1.00
2^2X2^2	SS	0.38	SS	0.75
2^3X2^0	AAA	0.75	AAA	1.00
2^3X2^1	AAS	0.50	AAS	1.00
2^3X2^2	SSA	0.42	SAS	0.75
2^3X2^3	SSS	0.31	SSS	0.61
2^4X2^0	AAAA	0.75	AAAA	1.00
2^4X2^1	AAAS	0.50	AAAS	1.00
2^4X2^2	SAAS	0.45	SAAS	0.75
2^4X2^3	SSSA	0.36	SASS	0.65
2^4X2^4	SSSS	0.27	SSSS	0.53
2^5X2^0	AAAAA	0.75	AAAAA	1.00
2^5X2^1	AAAAS	0.50	AAAAS	1.00
2^5X2^2	SAAAS	0.48	SAAAS	0.75
2^5X2^3	SSAAS	0.42	SAASS	0.67
2^5X2^4	SSSSA	0.32	SASSS	0.57
2^5X2^5	SSSSS	0.25	SSSSS	0.48
2^6X2^0	AAAAAA	0.75	AAAAAA	1.00
2^6X2^1	AAAAAS	0.50	AAAAAS	1.00
2^6X2^2	SAAAAS	0.50	SAAAAS	0.75
2^6X2^3	SSAAAS	0.47	SAAASS	0.69
2^6X2^4	SSSAAS	0.41	SASASS	0.60
2^6X2^5	SSSSSA	0.30	SASASS	0.53
2^6X2^6	SSSSSS	0.24	SSSSSS	0.45
2^7x2^0	AAAAAAA	0.75	AAAAAAA	1.00
2^7x2^1	AAAAAAS	0.50	AAAAAAS	1.00
2^7X2^2	SAAAAAS	0.50	SAAAAAS	0.75
2^7X2^3	SSAAAAS	0.49	SAAAASS	0.70
2^7X2^4	SSSAAAS	0.47	SAASASS	0.62
2^7X2^5	SSSSAAS	0.40	SSAASSS	0.55
2^7X2^6	SSSSSAS	0.29	SSASSSS	0.49
2^7X2^7	SSSSSSS	0.23	SSSSSSS	0.43
2^8X2^0	AAAAAAAA	0.75	AAAAAAAA	1.00
2^8X2^0	AAAAAAAS	0.50	AAAAAAAS	1.00
2^8X2^2	SAAAAAAS	0.50	SAAAAAAS	0.75
2^8X2^3	SSAAAAAS	0.50	SAAAAASS	0.71
2^8X2^4	SSSAAAAS	0.49	SAASAASS	0.64
2^8X2^5	SSSSAAAS	0.46	SASAASSS	0.57
2^8X2^6	SSSSSAAS	0.39	SSASASSS	0.52
2^8X2^7	SSSSSSAS	0.28	SSSASSSS	0.47
2^8X2^8	SSSSSSSS	0.22	SSSSSSSS	0.41

TAble 1 Optimal buffered arbitration networks

PERFORMANCE OF SELF-ROUTING SHUFFLE-EXCHANGE INTERCONNECTION NETWORK IN SIMD PROCESSORS

Jamshed H. Mirza
Polytechnic Institute of New York
333 Jay Street, Brooklyn, NY 11201

Abstract : This is a study of the class of multi-stage Shuffle-Exchange (S/E) networks used in Single-Instruction-Stream-Multiple-Data-Stream (SIMD) processors, and their ability to realize interconnection requests which are arbitrary permutations on the set of Processing Elements(PEs). The network is made self-routing, and a recursive model is proposed which analyses a network in terms of smaller ones. Performance parameters such as bandwidth, blocking probability, and the number of passes necessary to realize an arbitrary permutation are obtained.

Introduction : There are several topologically equivalent multi-stage S/E networks that have been proposed [2-4]. We will use the Omega network [5] as a representative of the class of S/E networks. However the results we obtain are equally applicable to all S/E networks since they have been shown to be topologically equivalent [6,7].

In SIMD processors, the interconnection requests by the $N=2^n$ PEs are made synchronously, and the source-to-destination pairs usually represent a permutation on the set of PEs. Consequently, we will analyse the S/E networks with respect to their ability to realize arbitrary permutations.

It has been shown that any permutation can be realized in a maximum of three passes through the network[7].However, this requires a setup time that is time-consuming even on SIMD processors. We will assume instead that the network is self-routing so that the switch settings are determined dynamically and no setup time is necessary. Each data element carries with it the destination address as a tag and the setting of a switch in stage j is determined by the jth bit of the tags at its two inputs[5]. Of course symultaneous connection of more than one sourse-destination pairs may result in conflicts.

The Model : Let $N=2^n$. If $i \in [0,2^n-1]$, then $i = (i)_{n-1} \ldots (i)_1(i)_0$. Also, let Ω_n represent a $2^n x 2^n$ Omega network.

The Ω network in fig. 1 is drawn to highlight the recursive structure of such networks. This makes possible a recursive procedure for analysing a large S/E network in terms of smaller ones.

Assume that the interconnection request is a permutation specified by $D = (D_0, D_1, \ldots D_{N-1})$, where D is any arbitrary permutation of the set (0,1, ... ,N-1). It specifies that input i is to be connected to output D_i. Since D is a permutation, there are exactly N/2 tags with $(D_i)_{n-1} = 0$, and the other N/2 have $(D)_{n-1} = 1$.

In our self-routing network, no request is blocked or turned back mid-way through the network. If a conflict occurs at a switch, one of the requests is allowed to go to the right output, while the other is misdirected and forced to go to the wrong output. At the output of the network the requests that have correctly reached their destinations are filtered out, and only the misdirected ones are made to pass through the network again, starting from the destination they reached during the previous pass. The data elements carry with them two 1-bit flags, m and r, besides the n-bit tag identifying their destination. At any time, a request may be a correctly directed request (cdr, m=0) if it has been directed to the correct output at all switches it has encountered during the current pass, or a misdirected request (mdr, m=1) if it was misdirected at least once during the current pass.Also, since multiple passes through the network may be necessary, the r bit tells us whether a request has(r=1) or has not (r=0) already reached its destination during some previous pass.

Initially all requests start off with m=0 and r=0. The self-routing algorithm works as follows :

(i) if a mdr and a cdr meet at a switch, the switch is set by the cdr.

(ii) if 2 cdrs or 2 mdrs meet, the switch is set according to the tag at the upper input of the switch. If a conflict occurs, the lower input request is misdirected and is so marked by setting its m bit equal to 1.

At the output of the network the cdrs(m=0,r=0) are filtered out by the respective PEs. Also, if a request has m=1 and r=0, its m bit is reset to 0 for the next pass. For all other requests, m and r are both set to 1 to create "dummy" mdrs.The requests are then cycled again through the network. Thus during each pass, there exactly N requests - some cdrs and some mdrs. Since the self-routing algorithm always resolves conflicts in favour of a cdr, the mdrs can be considered to be "don't care" requests. We can assume that their tags have whatever value is necessary to justify our assumption that every Ω_k network in the recursive model is presented at its input with a permutation on the set $(0,1,\ldots,2^k-1)$.

Analysis : At stage(n-1) of the Ω_n network, D_i and $D_{i+N/2}$ meet at a switch. As long as their $(n-1)^{th}$ bits are different, conflict does not occur. If their (n-1)th bits are equal, a conflict occurs and $D_{i+N/2}$ is misdirected. Also, since exactly N/2 of the tags have the (n-1)th bit equal to 0 and the remaining have it equal to 1, conflicts will always occur in pairs. Then,

$$P(2c \text{ conflicts}) = \frac{\binom{N/2}{2c}\binom{2c}{c}\, 2^{N/2-2c}}{\binom{N}{N/2}} \qquad (1)$$

where $0 \leq 2c \leq N/2$ or $0 \leq c \leq N/4$.

Define for stage(n-1) of Ω_n, a $(2^n+1)x(2^n+1)$ Stage Transmission Matrix S_n, such that

$S_n(a,b) = P(b \text{ cdrs leave stage(n-1)} \mid a \text{ cdrs enter stage(n-1)})$

Using (1) we can find $S_n(N,N-k)$ for k even and $0 \leq k \leq N/2$. For k odd or $N/2 < k \leq N$, $S_n(N,N-k) = 0$.

Using $S_n(N,N-k)$, we can find $S_n(a,b)$ for all $0 \leq a < N$ and $0 \leq b \leq a$. For $b > a$, $S_n(a,b)=0$. We can write $S_n(a,b)$ as $S_n(N-i,N-(i+j))$, which gives the probability that we enter stage(n-1) with i

0190-3918/82/0000/0013$00.75 © 1982 IEEE

mdrs and j new mdrs are created at that stage.We have to calculate the contributions of the (N,N-k) cases to the (N-i,N-(i+j))cases. This represents a situation where there are i mdrs at the input of stage(n-1), there are conflicts at k switches and no conflicts a k''=N/2-k switches, and as a result of which j new mdrs are created ($j \leq k$) at the output of stage(n-1). Thus k'=k-j of the conflicts do not create a new mdr because at least one of the i input mdrs appear at each of those k' conflict switches.A little reflection will verify that at least L1=MAX(0,i-(2k''+k)) of these k' switches will have 2 input mdrs, and at most L2=MIN(i-k',k') of them can have two input mdrs. Then the contribution of the (N,N-k) case to the (N-i,N-(i+j))case is :

$$(N\text{-}N\text{-}k) \rightarrow (N\text{-}i,N\text{-}(i+j)) = \left[\frac{\binom{k}{k'}\sum_{p=L1}^{L2}\binom{k'}{p}2^{k'-p}\binom{2k''}{i-k'-p}}{\binom{N}{i}}\right] S_n(N,N\text{-}k) \quad (2)$$

We can then find the elements of S_n by :

$$S_n(N\text{-}i,Ni(i+j)) = \sum_{\substack{k \text{ even} \\ 0 \leq k \leq N/2}} \left[(N,N\text{-}k) \rightarrow (N\text{-}i,N\text{-}(i+j))\right] S_n(N,N\text{-}k) \quad (3)$$

Using these equations we can find S_k for any $k \geq 1$.

For any Ω_k, define the $(2^k+1)x(2^k+1)$ Network Transmission Matrix T_k, such that :

$T_k(i,j)$=P(j cdrs leave Ω_k | i cdrs enter Ω_k)

T_k is lower-triangular and gives the transmission through all stages of the network. Note that $T_1 = S_1 = I$ (where I is the unit matrix).

To determine T, we need to define for a network Ω_k, a $(2^k+1)x(2^k+1)$ matrix R_k such that :

$R_k(i,j)$ = P(j cdrs get through Ω_k | i cdrs get through stage(n-1) of Ω_k)

R_k is also lower-triangular and gives the transmission through stages (k-2) to 0 of Ω_k. Then,

$$T_k = S_k \times R_k \quad (4)$$

Looking at the recursive structure of the network (fig.2) we can verify that

$$R_k(i,j) = \sum_{(i1,i2)} \sum_{(j1,j2)} \left\{ \left[\binom{2^{k-1}}{i1}\binom{2^{k-1}}{i2} \Big/ \binom{2^k}{i}\right] \times T_{k-1}(i1,j1)T_{k-1}(i2,j2) \right\} \quad \text{if } j \leq i$$

$$= 0 \quad \text{if } j > i \quad (5)$$

Here the summations are over all distributions of i into (i1,i2) and of j into (j1,j2) such that

$MAX(0,i-2^{k-1}) \leq i1,i2 \leq MIN(i,2^{k-1})$, $j1 \leq i1$, $j2 \leq i2$.

Then, starting with $T_1 = I$, we can recursively find R_k from T_{k-1}, and then T_k from S_k and R_k, until T_n is determined for a Ω_n network.

Performance Parameters : Using the recursive model we can determine several important performance parameters associated with S/E networks.

The Bandwidth $B_n(i)$ of a NxN S/E network ($N=2^n$) is defined as the expected number of cdrs at the output, given that i cdrs entered the network. Then,

$$B_n(i) = \sum_{j=0}^{N} jT_n(i,j) \quad (6)$$

The Blocking Probability is the probability that a request that enters as a cdr is blocked or misdirected during its passage through the network because of conflicts. It depends on i, the number of cdrs at the input, and is given by :

$$p_n(i) = \sum_{j=0}^{N} (i\ j)T_n(i,j) = 1 - B_n(i)/i \quad (7)$$

Define the load L = i/N. Figures 3 and 4 show the variation ofthe Bandwidth and Blocking Probability with respect to the Load. Plotting the results with respect to L rather than i normalizes the plots for the different size networks. For $L < 1/2$ B increases almost linearly because at low loads mdrs are created mainly at the first stage and very few at latter stages since the chances of 2 cdrs meeting in a conflict at later stages is low. At higher loads ($L > 1/2$) there are more chances of 2 cdrs conflicting in later stages and the bandwidth increases at a slower rate. Fig. 3 also shows the results obtained by simulation and the two values are seen to agree closely. The plots also suggest that the bandwidth increases at a faster rate with increasing n. This is verified in fig. 5. Fig. 6 gives, for different size networks, the blocking probability at each of the n stages rather than the network as a whole. Its shape would serve to explain the shape of fig. 5. Since blocking probability falls more dramatically in later stages of larger networks, the bandwidth increases at a faster rate as the network size increases.

To determine the expected number of passes necessary to realize an arbitrary permutation, we define an absorbing Markov Chain (MC). The MC is in state i if i requests still need to be routed to their destinations. If X_n is the state transition matrix for the MC, then

$X_n(i,j)$ = P(j requests at the output still need to be routed | i requests at the input need to be routed)

$Xn(i,j) = T_n(i,i-j)$ for $j \leq i$

$= 0$ for $j > i$

Also, $X_n(i,i)=T_n(i,0)=0$ for all $i > 0$, and $X_n(0,0)=1$. Thus stste-0 is the only absorbing state, and all other states are transient and are visited at most once. If #passes(i) is the expected number of additional passes necessary if the MC is in state i, then :

$$\#passes(0) = 0,$$

$$\#passes(i) = 1 + \sum_{j=0}^{i-1} X_n(i,j)\#passes(j) \quad \text{for } i>0$$

Fig. 7 shows the variation of #passes(i) for different size networks at different loads. The plot for L=1 gives the expected number of passes to realize an arbitrary permutation. Fig. 8 shows that #passes(i)/n is more or less constant for different network sizes and is about (2/3)n. for L=1.

Conclusions : In this paper we presented a recursive model for a self-routing S/E network and used it to determine several performance parameters. Simulation results were found to agree closely with the analytical results.Although the network was used here, the results are equally applicable to all S/E networks. A more detailed treatement and discussion will be found in [1]. Further, in [1], it is shown that the class of Bit-Permute-Complement permutations [8] which include many of the permutations commonly encountered in parallel algorithms require only two passes through the self-routing S/E network to be realized.

References

1. J.Mirza, "Performance of Shuffle-Exchange Networks ," in preparation as Polytechnic Report.
2. L.Haynes et al,"A survey of highly parallel Computing,"Computer,Vol.15,No.1,Jan.1982.
3. H.Siegel,"A Model of SIMD machines and a comparision of various interconnection networks," IEEE Trans.Compt.,Vol.C-28,No.12,Dec.1979.
4. T.Feng,"A survey of Interconnection Networks," Computer,Vol.14,No.12,Dec.1981.
5. D.Lawrie,"Access and Allignment of data in an Array processor,"IEEE Trans.Compt.,Vol.C-24, No.12,Dec.1975.
6. C.Wu and T.Feng,"On a class of multi-stage Interconnection Networks,"IEEE Trans. Compt.,Vol. C-29, No.8, Aug.1980.
7. C.Wu,T.Feng,"The universality of Shuffle-Exchange network,"IEEE Trans.Compt.,Vol.C-30,May 1981.
8. D.Nassimi,S.Sahni,"A self-routing Benes network and parallel permutation algorithms,"IEEE Trans. Compt.,Vol.C-30,No.5,May 1981.

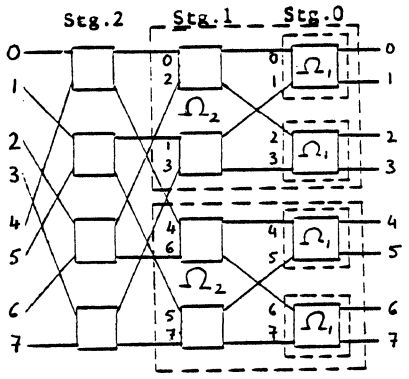


Fig.1: Omega Network (Ω_3)

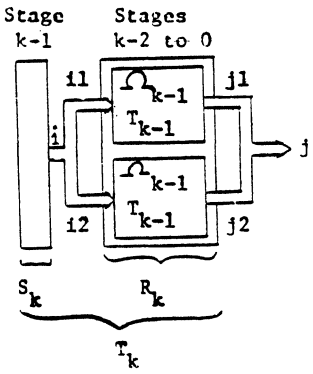


Fig. 2.

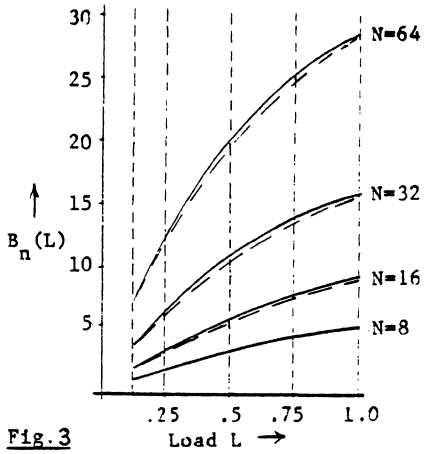


Fig.3

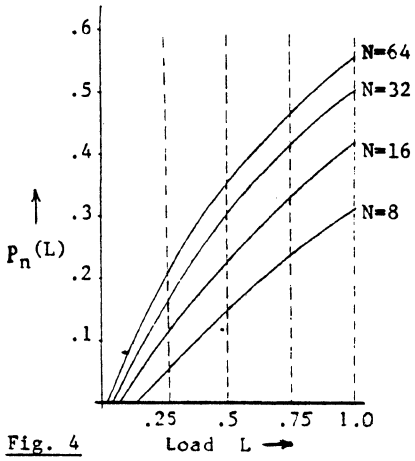


Fig. 4

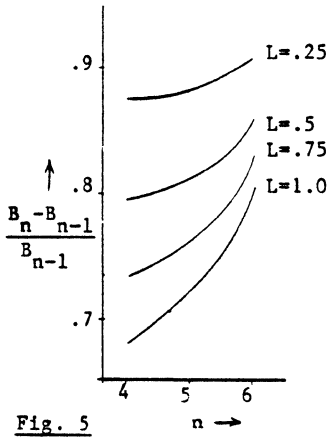


Fig. 5

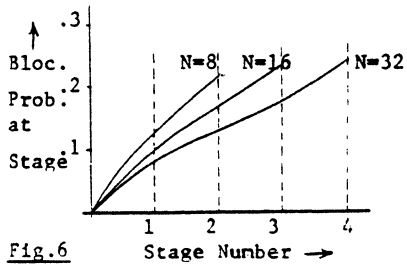


Fig.6

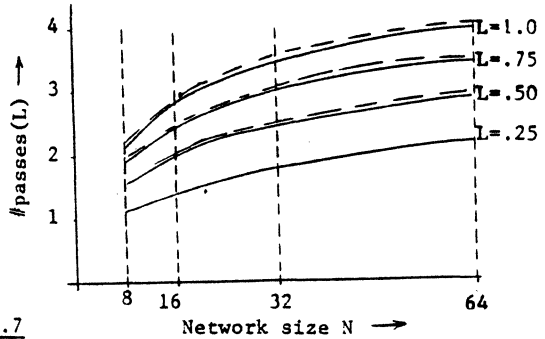


Fig.7

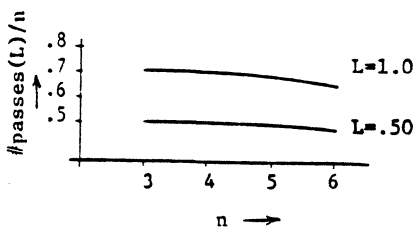


Fig. 8

(NOTE : Plots with broken lines were obtained by simulation)

SP2I INTERCONNECTION NETWORK AND EXTENSION OF THE ITERATION METHOD OF AUTOMATIC VECTOR-ROUTING

Wang Rong-Quan, Zhang Xiang, Gao Qing-Shi

Department of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
Beijing, People's Republic of China

Abstract -- In this paper the SP2I (Single-stage Plus 2^i) interconnection network, which is applicable to the CVCVHP with VCM (Cellular Vector Computer of Vertical-Horizontal Processing with Virtual Common Memory) and other multiprocessor systems, is discussed. Starting from the need for dynamic and parallel data alignment, we investigate various properties of conflict-free routing, describe the iteration method of automatic vector-routing which may be used to solve the conflict problem in the SP2I network. Furthermore, we extend the iteration method to the networks of ADM, Ω, δ, Indirect Binary n-Cube, Baseline, etc., which are usually seen in literature. Then the problem of routing conflict in these networks, which has not been well solved yet so far, may be solved efficiently. Finally, the implementation methods of several common data manipulation functions without conflict are given.

Introduction

The background of this paper is the problem of dynamic and parallel data alignment between vector processor and virtual common memory in the CVCVHP with VCM (Cellular Vector Computer of Vertical--Horizontal Processing with Virtual Common Memory). We investigate various properties of SP2I network and solve the routing conflict problem in SP2I and many other networks.

The CVCVHP system [1] consists of N cells. Suppose $N=2^n$. Every cell C_j has a processor P_j and a memory bank M_j with capacity of 2^m words, $j=0,1,\cdots,N-1$, $\{M_j | j=0,1,\cdots,N-1\}$ constitutes the VCM. The address $D=D'\cdot 2^n+D''$ points to the D'-th element of $M_{D''}$, D' is local address, D" is the memory bank number or cell number. Let $\vec{J}=(0,1,\cdots,N-1)$, $J_j=j$ is a binary constant number of n bits which is set up in C_j .

The SP2I network structure is used and it is really half of the PM2I single--stage network [10,11] . A routing register E_j is set up in C_j , $\{E_j | j=0,1,\cdots,N-1\}$ constitutes the vector routing register $\vec{E}$. The interconnection of the CVCVHP system is just the interconnection among elements of $\vec{E}$. Taking the subscript j of element of $\vec{E}$ as variable, the SP2I network has n interconnection functions [10,11] : $F_i(j)=j+2^i \pmod N$, $i=0,1,\cdots,n-1$; $j=0,1,\cdots,N-1$. Fig.1 shows the interconnection of the j-th cell with others. Fig.2 shows the information structure of E_j , these data items all participate in routing.

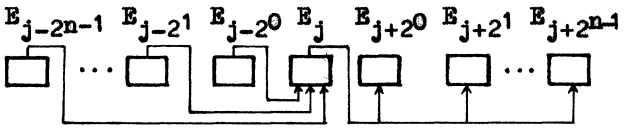


Fig.1, The interconnection of E_j with others

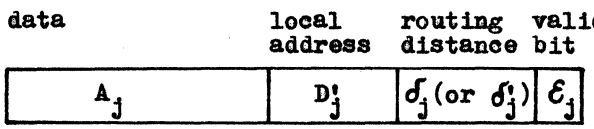


Fig.2, The information structure of E_j

Routing Rules of SP2I network

Let $x=x_{n-1}x_{n-2}\cdots x_1x_0$ be the binary notation of x, $(x)_i=x_i$, $L(k,x)=x_{k-1}x_{k-2}$

0190-3918/82/0000/0016$00.75 © 1982 IEEE

$\cdots x_1 x_0$, $H(k,x)=x_{n-1}x_{n-2}\cdots x_{n-k}$.

Suppose C_j produces an address $D_j=D'_j\cdot 2^n+D''_j$. In fetching operation, D'_j must be transmitted from C_j into $C_{D''_j}$, then the data A_j can be fetched out from $M_{D''_j}$. The fetched data A_j must be transmitted from $C_{D''_j}$ back into C_j, then C_j can use the data A_j. In storing operation, data and local address D'_j must be transmitted from C_j into $C_{D''_j}$, then the data can be stored into $M_{D''_j}$. This kind of transmission of data or address from C_j to $C_{D''_j}$ is called "routing", the transmission of A_j from $C_{D''_j}$ back to C_j is called "return-routing". Both routing and return-routing are executed in $\vec{E}$. Let δ_j be the routing distance, δ'_j be the return-routing distance. $\delta_j=D''_j-j \pmod N$, $\delta'_j=N-\delta_j \pmod N$, $0\le j\le N-1$. Both δ_j and δ'_j are binary numbers of n bits.

In the network controlling structure, routing is controlled by δ_j , return-routing is controlled by δ'_j. When the Memory Control Unit (MCU) sends out the command of "routing $+2^i$", the routing rule is:

(a) $\varepsilon_j=0$, this shows that E_j is invalid. So E_j has no effect on E_{j+2^i} . If the information of E_{j-2^i} must move to E_j, then E_j changes into new state and new value; otherwise ε_j remains unchanged.

(b) $\varepsilon_j=1$, this means E_j has valid information.Routing result depends on $(\delta_j)_i$:

(i) $(\delta_j)_i=1$, the content of E_j has to move to E_{j+2^i} . If the content of E_{j-2^i} must move to E_j , then E_j changes into new value; otherwise set $\varepsilon_j=0$.

(ii) $(\delta_j)_i=0$, this means E_j has valid information and does not need to move. E_j has no effect on E_{j+2^i} . If the content of E_{j-2^i} does not need to move to E_j, then E_j remains unchanged; otherwise E_j changes into new value, the old valid information of E_j is forced to disappear. This is the routing conflict.

The procedure of parallelly fetching a vector is:

(a) Compute the address vector $\vec{D}$ by all cells, then resolve $\vec{D}$ into $\vec{D}'$ and $\vec{D}''$. Compute $\vec{\delta}=\vec{D}''-\vec{J} \pmod N$. Send $\vec{D}'$, $\vec{\delta}$ and corresponding valid bit vector $\vec{\varepsilon}$ into $\vec{E}$ (i.e., $\vec{D}'\Rightarrow\overrightarrow{ED'}$, $\vec{\delta}\Rightarrow\overrightarrow{E\delta}$, $\vec{\varepsilon}\Rightarrow\overrightarrow{E\varepsilon}$, where $\overrightarrow{ED'}$ stands for $\vec{D}'$ of $\vec{E}$, $\overrightarrow{E\delta}$ for $\vec{\delta}$ of $\vec{E}$, $\overrightarrow{E\varepsilon}$ for $\vec{\varepsilon}$ of $\vec{E}$. This notation will be used below not only for $\vec{E}$ but also for $\vec{F}$ and $\overrightarrow{RI}$).

(b) Perform n steps of routing $+2^0$, $+2^1,\cdots,+2^{n-1}$.

(c)Fetch out corresponding data vector according to $\overrightarrow{ED'}$ under the control of $\overrightarrow{E\varepsilon}$ (i.e., every cell which has valid local address fetches a data from its private memory bank). Then the fetched data vector is sent into $\overrightarrow{EA}$. Compute $\delta'=N-\overrightarrow{E\delta}$. $\vec{\delta}'\Rightarrow\overrightarrow{E\delta}$.

(d) Perform n steps of return-routing $+2^0,+2^1,\cdots,+2^{n-1}$.

(e) $\overrightarrow{EA}$ is sent into $\vec{B}$ under the control of $\overrightarrow{E\varepsilon}$ (i.e., if $E\varepsilon_j=1$, then $EA_j\Rightarrow B_j$; otherwise $E\varepsilon_j=0$, then $EA_j\not\Rightarrow B_j$), where $\vec{B}$ is the vector buffer register for lookahead fetching. Then the data vector is in $\vec{B}$ and ready for use.

The procedure of parallelly storing a vector is similar with that of fetching operation, but the storing operation of a vector without conflict does not need the return-routing.

The SP2I network also may offer another kind of routing command: "broadcast-type routing $+2^i$", denoted as "routing $\underline{+2^i}$". The only difference of "$\underline{+2^i}$" from "$+2^i$" is: when the content of E_j moves to E_{j+2^i} and the content of E_{j-2^i} does not move to E_j , "$\underline{+2^i}$" makes E_j still remain its original state and value, but "$+2^i$" makes E_j become "empty".

Properties of Conflict-free Routing

For conflict-free vector routing,only 2n steps of routing treatment are needed for fetching a vector, n steps for storing a vector.

Theorem 1:For an address vector of N valid elements, n steps of routing treatment with arbitrary-order have no conflict if and only if the $\vec{\delta}$ satisfies the condition (*), i.e., for any integers i,j ($0 \le j < N$, $0 \le i < n$), there exists

$$\delta_{j+2^i} = \delta_j \pmod{2^{i+1}} \qquad (*)$$

Proof: Please see paper [2].

For a Δ-ordered vector $\vec{Y}$ whose address vector is $\vec{D}=(D, D+\Delta, D+2\Delta, D+3\Delta, \cdots)$, where D is initial address, constant Δ is address increment, if $K \le N/\gcd(\Delta,N)$, then any K consecutive addresses of $\vec{D}$ point to K different memory banks, where $\gcd(\Delta,N)$ is the greatest common divisor of Δ and N. So,if Δ is odd, then $\gcd(\Delta,N)=1$, therefore any N consecutive addresses of $\vec{D}$ point to N different memory banks. Generally speaking, pointing to N different memory banks is not sufficient enough to ensure routing conflict-free. But for odd Δ, according to Theorem 1, we have Corollary 1.

Corollary 1: The Δ-ordered address vector $\vec{D}$ of N valid elements with odd Δ is conflict-free in routing.

Proof: $\vec{D}=D+\vec{J}\cdot\Delta$, $\vec{\delta}=\vec{D}-\vec{J}=D+\vec{J}\cdot(\Delta-1) \pmod N$, $\delta_{j+2^i}-\delta_j=D+(j+2^i)(\Delta-1)-(D+j(\Delta-1))=2^i(\Delta-1)$. For odd Δ, $\Delta-1$ is even, $\therefore \delta_{j+2^i}-\delta_j=2^i(\Delta-1)=2^{i+1}\cdot\Delta'=0 \pmod{2^{i+1}}$, i.e., $\vec{\delta}$ satisfies the condition (*). According to Theorem 1, the Corollary is true.

In practice,the MCU issues n steps of routing command usually in certain order, such as $+2^0,+2^1,\cdots,+2^{n-1}$ or $+2^{n-1},+2^{n-2},\cdots,+2^0$.

Lemma 1: Suppose data A starts moving from E_j, routing distance is δ, routing command order is $+2^0,+2^1,\cdots,+2^{n-1}$. After k steps of routing ($1 \le k \le n$), A arrives at $E_{j+L(k,\delta)}$.

Proof: k steps of routing cause A to have moved a distance $L(k,\delta)$, so A must arrive at $E_{j+L(k,\delta)}$.

Lemma 2: Suppose data A starts moving from E_j, routing distance is δ, routing command order is $+2^{n-1},+2^{n-2},\cdots,+2^0$. After k steps of routing, A arrives at $E_{j+2^{n-k}H(k,\delta)}$, ($1 \le k \le n$).

Proof: k steps of routing cause A to have moved a distance $2^{n-k}\cdot H(k,\delta)$, so A must arrive at $E_{j+2^{n-k}\cdot H(k,\delta)}$.

Theorem 2: $\vec{E}$ has N valid elements, routing command order is $+2^0,+2^1,\cdots,+2^{n-1}$, then there will be no routing conflict if and only if for any integers i,j,k ($0 \le i < N$, $0 \le j < N$, $1 \le k \le n$), $i \ne j \longrightarrow i+L(k,\delta_i) \ne j+L(k,\delta_j) \pmod N$.

Proof:Conflict-free routing means for any E_i, E_j ($0 \le i < N$, $0 \le j < N$, $i \ne j$), after any k ($1 \le k \le n$)steps of routing,they do not arrive at any same routing register.Based on Lemma 1,it is obvious that Theorem 2 is true.

Theorem 3: $\vec{E}$ has N valid elements, routing command order is $+2^{n-1},+2^{n-2},\cdots,+2^0$. Then there will be no routing conflict if and only if for any integers i,j,k ($0 \le i < N$, $0 \le j < N$, $1 \le k \le n$), $i \ne j \longrightarrow i+2^{n-k}\cdot H(k,\delta_i) \ne j+2^{n-k}\cdot H(k,\delta_j) \pmod N$.

Proof: It follows from Lemma 2 and the concept of conflict-free routing.

Lemma 3: Suppose x,y move from E_{i_0}, E_{j_0} to E_{i_K}, E_{j_K}, respectively.If (i) $i_0 < j_0$, $i_K < j_K$, (ii) $j_0-i_0 \ge j_K-i_K$, (iii) $\delta_x \ge \delta_y$, (iv) routing command order is $+2^0,+2^1,\cdots,+2^{n-1}$, then x and y are conflict-free.

Proof: When $\delta_x = \delta_y$, x and y either both move or both do not move at the same time. This is called "parallel moving". Obviously, in this case x and y are conflict-free. Below we consider the case of $\delta_x > \delta_y$. Without losing generality, suppose

$$\delta_x = \underline{u_{n-1}u_{n-2}\cdots u_{p+1}1u_{p-1}\cdots u_0} \qquad (1)$$

$$\delta_y = \underline{u_{n-1}u_{n-2}\cdots u_{p+1}0u'_{p-1}\cdots u'_0} \qquad (2)$$

By the theorem conditions, it is easy to know that either $((i_K \ge i_0)\wedge(j_K \ge j_0))=1$ or $((i_K < i_0)\wedge(j_K < j_0))=1$, where $\wedge$ is logical AND. If $((i_K \ge i_0)\wedge(j_K \ge j_0))=1$, then $(j_0-i_0)-(j_K-i_K)=(i_K-i_0)-(j_K-j_0)=\delta_x-\delta_y$; if $((i_K < i_0)\wedge(j_K < j_0))=1$, then $(j_0-i_0)-(j_K-i_K)=(N-i_0+i_K)-(N+j_K-j_0)=\delta_x-\delta_y$.

Anyway, we obtain

$$(j_0-i_0)-(j_K-i_K)=\sigma_x-\sigma_y \quad (3)$$

From (1) and (2), we get

$$\sigma_x-\sigma_y=L(p+1,\sigma_x)-L(p+1,\sigma_y) \quad (4)$$

From (3), we get

$$j_0-i_0=j_K-i_K+\sigma_x-\sigma_y > \sigma_x-\sigma_y \quad (5)$$

If there exists an integer k ($1\leq k\leq n$), and a routing conflict occurs at the k-th step of routing, then from the Theorem 2, we have

$$i_0+L(k,\sigma_x)=j_0+L(k,\sigma_y) \pmod N \quad (6)$$

$$j_0-i_0=L(k,\sigma_x)-L(k,\sigma_y) \pmod N \quad (7)$$

(a) If $k\geq p+1$, from (7), (1),(2),(4), we obtain $j_0-i_0=\sigma_x-\sigma_y$. This is contradicted by the fomula (5).

(b) If $k\leq p$

(i) When $L(k,\sigma_x)\geq L(k,\sigma_y)$, then (7) becomes

$$j_0-i_0=L(k,\sigma_x)-L(k,\sigma_y) \quad (8)$$

Substitute (1) and (2) for σ_x and σ_y in (5), respectively, we get

$$j_0-i_0=(j_K-i_K)+(1u_{p-1}\cdots u_k-0u'_{p-1}\cdots u'_k)\cdot 2^k$$
$$+L(k,\sigma_x)-L(k,\sigma_y)>L(k,\sigma_x)-L(k,\sigma_y)$$

This contradicts the formula (8).

(ii) When $L(k,\sigma_x)<L(k,\sigma_y)$, from (6), since $i_0<j_0$, so $((j_0+L(k,\sigma_y)\geq N)\wedge(i_0+L(k,\sigma_x)<N))=1$. This must be the case of $((j_K<j_0)\wedge(i_K<i_0))=1$. Thus (6) must be $i_0+L(k,\sigma_x)=j_0+L(k,\sigma_y)-N$, then $N-j_0+i_0+L(k,\sigma_x)=L(k,\sigma_y)<2^k$, so

$$i_0+L(k,\sigma_x)<2^k \quad (9)$$

From $E_{i_0+L(k,\sigma_x)}$ to E_{i_K} , x has to cover the distance of $N-(i_0+L(k,\sigma_x))+i_K$. From (9), $N-(i_0+L(k,\sigma_x))+i_K>N-2^k+i_K\geq N-2^k$ (10)

On the other hand, according to (1), the distance from $E_{i_0+L(k,\sigma_x)}$ to E_{i_K} is $H(n-k,\sigma_x)\cdot 2^k$. But $H(n-k,\sigma_x)\cdot 2^k\leq N-2^k$. This contradicts the formula (10).

By all the above contradictions, Lemma 3 must be true.

Theorem 4: Suppose x_{i_k} moves from E_{i_k} to E_{j_k} , the routing distance of x_{i_k} is σ_{i_k}, ($k=1,2,\cdots,K$). If for $k=1,2,\cdots,K-1$,

(i) $0\leq i_k<i_{k+1}<N$, $0\leq j_k<j_{k+1}<N$

(ii) $i_{k+1}-i_k\geq j_{k+1}-j_k$

(iii) $\sigma_{i_k}\geq\sigma_{i_{k+1}}$

(iv) routing command order is $+2^0,+2^1,\cdots,+2^{n-1}$

then for $S=\{x_{i_k}\mid k=1,2,\cdots,K\}$, n steps of routing have no conflict.

Proof: For any p,q ($1\leq p<q\leq K$), by Lemma 3, x_{i_p} and x_{i_q} are conflict-free, so is S.

Lemma 4: Suppose x,y move from E_{i_0}, E_{j_0} to E_{i_K}, E_{j_K}, respectively. If (i) $i_0<j_0$, $i_K<j_K$, (ii) $j_0-i_0\leq j_K-i_K$, (iii) $\sigma_x<\sigma_y$, (iv) routing command order is $+2^{n-1},+2^{n-2},\cdots,+2^0$, then x and y are conflict-free.

Proof: The proof is similar to that of Lemma 3.

Theorem 5: Suppose x_{i_k} moves from E_{i_k} to E_{j_k}, the routing distance of x_{i_k} is σ_{i_k} , ($k=1,2,\cdots,K$). If for $k=1,2,\cdots,K-1$,

(i) $0\leq i_k<i_{k+1}<N$, $0\leq j_k<j_{k+1}<N$

(ii) $i_{k+1}-i_k<j_{k+1}-j_k$

(iii) $\sigma_{i_k}\leq\sigma_{i_{k+1}}$

(iv) routing command order is $+2^{n-1},+2^{n-2},\cdots,+2^0$

then for $S=\{x_{i_k}\mid k=1,2,\cdots,K\}$, n steps of routing have no conflict.

Proof: The proof is similar to that of Theorem 4.

Theorems 6,7, and 8 describe the ability of SP2I network to simulate the fundamental functions of other interconnection networks. The proof is simple and will be omitted here.

Vector $\vec{Y}=(Y_0,Y_1,\cdots,Y_{K-1})$ is called canonical if and only if for all k ($k=0,1,\cdots,K-1$), Y_k is in C_k, where $K\leq N$. Suppose $p=p_{n-1}p_{n-2}\cdots p_1p_0$ is the binary notation of the order number p of cell C_p .

Indirect Binary n-Cube network has n interconnection functions [10,11] :

$$Cube_k(\vec{J})=(C_k(0),C_k(1),\cdots,C_k(N-1))$$

where $C_k(p)=p_{n-1}p_{n-2}\cdots p_{k+1}\bar{p}_k p_{k-1}\cdots p_1 p_0$; $p=0,1,\cdots,N-1$; $k=0,1,\cdots,n-1$.

The basic interconnecting structure of Ω, δ ,Baseline, etc., is Shuffle-Exchange. The Shuffle function and Exchange function are two principal interconnecting functions [10,11] :

$\text{Shuffle}(\vec{J})=(S(0),S(1),\cdots,S(N-1))$

$\text{Exchange}(\vec{J})=(E(0),E(1),\cdots,E(N-1))$

where $S(p)=p_{n-2}p_{n-3}\cdots p_1 p_0 p_{n-1}$,

$E(p)=p_{n-1}p_{n-2}\cdots p_1\bar{p}_0$, $p=0,1,\cdots,N-1$. If one step of routing transmits a canonical vector $\vec{Y}$ into its destination with the address vector $F(\vec{J})$, then the network is said to have realized the $F(\vec{J})$ function.

Theorem 6: n-k steps of routing $+2^{j_1}$,$+2^{j_2}$, $\cdots$,$+2^{j_{n-k}}$, can realize the $\text{Cube}_k(\vec{J})$ function, where $(j_1,j_2,\cdots,j_{n-k})$ is any permutation of $(k,k+1,\cdots,n-1)$.

Theorem 7: n steps of routing with arbitrary order can realize the $\text{Exchange}(\vec{J})$ function.

Theorem 8: For a canonical vector of length N, n steps of routing with any command order can not realize the $\text{Shuffle}(\vec{J})$ function, but (2n-1) steps can.

Extension of the Iteration Method of Automatic Vector-Routing

Some vectors have routing conflict , such as Δ-ordered vector with even Δ. It has memory bank conflict, therefore has transmission route conflict. Even if the N elements of an address vector point to N different memory banks, there will still possibly exist routing conflict. For example, the bit reverse transmission of a canonical vector $\vec{Y}$ has no memory bank conflict,but has transmission route conflict, where $\vec{Y}=(Y_0,Y_1,\cdots,Y_{N-1})$, the bit reverse transmission means transmitting Y_p from E_p to $E_{p'}$, $p=p_{n-1}p_{n-2}\cdots p_1 p_0$, $p'=p_0 p_1\cdots p_{n-2}p_{n-1}$, $p=0,1,\cdots,N-1$.

In order to realize dynamic and parallel data alignment, a network must solve the conflict problem. The iteration method of automatic vector-routing which was presented in paper [2] is an efficient method to solve the routing conflict problem for SP2I network.The basic idea is as follows:

In addition to $\vec{E}$, we set up another vector register $\vec{F}=(F_0,F_1,\cdots,F_{N-1})$ for information reservation. F_j is in C_j . The information structure of F_j is the same as that of E_j (see Fig.2).

The whole procedure of accessing a vector consists of several routing iterations. Each iteration consists of n steps of routing and n steps of return-routing.

First,the information to be transmitted is sent into $\vec{F}$. Then perform routing iteration.

(i) $\vec{F}\Rightarrow\vec{E}$

(ii) $\vec{E}$ executes n steps of routing and n steps of return-routing. If conflict occurs, we allow some valid information to disappear from $\vec{E}$.

(iii) If $E\mathcal{E}_j=1$, then $EA_j\Rightarrow B_j$, $0\Rightarrow F\mathcal{E}_j$ (i.e., $\mathcal{E}_j$ of F_j); otherwise B_j and $F\mathcal{E}_j$ remain unchanged, $(j=0,1,\cdots,N-1)$.

(iv) If $\vec{F\mathcal{E}}\neq(0,0,\cdots,0)$, then go to (i); otherwise the fetching operation is finished.

The iteration method of automatic vector-routing may be extended to many multi-stage or single-stage interconnection networks to solve their routing conflict problems.For example,AIM [3] (Augmented Data Manipulator) network has 2n interconnecting functions $F_i(j)=j\pm 2^i$, $j=0,1,\cdots,N-1$; $i=0,1,\cdots,n-1$. Its controlling structure also uses the routing distance as controlling tag. Thus, the above iteration method may be directly used in the AIM network without any modification. When rerouting is used in the AIM network, the iteration number needed for AIM network is usually less than that needed for SP2I.

The iteration method may also be extended to such multi-stage Shuffle-Exchange-type networks as Indirect Binary n-Cube,

Ω, δ, Baseline, etc. [8,9]. In their controlling structure the destination address D_j is used as routing tag of the source address S_j ($j=0,1,\cdots,N-1$). In the i-th stage, if $(D_j)_i=1$, then the data is switched into the lower output of the switch element;otherwise into the higher output.When extending the iteration method to these networks to solve their conflict problems,the information to be transmitted consists of four parts as shown in Fig.3.

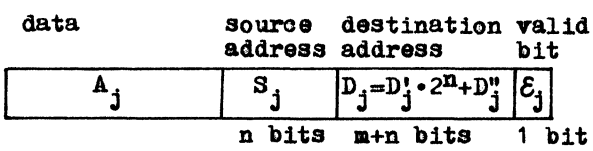


Fig.3, Information structure of RI_j and F_j

We set up an information input vector register $\overrightarrow{RI}$ and an information reservation vector register $\overrightarrow{F}$. RI_j and F_j are in C_j. Their information structure is shown in Fig.3 . Suppose the CVCVHP system uses the above Shuffle-Exchange-type multi-stage interconnection network to realize dynamic and parallel data alignment, we will take the fetching operation as example to describe the extended iteration method.

Suppose the MCU is going to fetch a vector $\vec{Y}$ of N elements from VCM and send it into vector buffer register $\vec{B}$, denoted as $\vec{Y}\Rightarrow\vec{B}$. At first,compute $\vec{Y}$'s address vector $\vec{D}=(D_0,D_1,\cdots,D_{N-1})$,where $D_j=D'_j\cdot 2^n+D''_j$. $\vec{J}\Rightarrow\vec{FS}$, $\vec{D}\Rightarrow\vec{FD}$, $\vec{\mathcal{E}}=\vec{F\mathcal{E}}$. Then perform routing iteration.

(i) Routing: $\vec{F}\Rightarrow\overrightarrow{RI}$. $\overrightarrow{RI}\Rightarrow$ network. D''_j of RI_j is used as routing tag. If at some k-th stage, a switch element has two valid inputs with source addresses i and j, and $(D''_i)_k=(D''_j)_k$,then the two input information have to output at the same higher (or lower) output port of the switch element, causing a conflict. In this case, we may choose only one input (such as always the higher input)to output and let the another to disappear (i.e., set its valid bit into zero). Thus for this switch element, its one output port sends out valid information,another output port sends out invalid information. Finally the output of the network returns to $\overrightarrow{RI}$.

Then parallelly perform a fetching operation, i.e., if $RI\mathcal{E}_j=1$, then fetch a data from M_j according to local address RID'_j and send it into RIA_j ; otherwise $RI\mathcal{E}_j=0$, then RIA_j remains unchanged, $j=0,1,\cdots,N-1$.

(ii) Return-routing: $\overrightarrow{RI}\Rightarrow$ network. This time, $\vec{S}$ is used as destination address vector, S_j becomes the routing tag. The transmission conflict is treated in the same way as (i). The output of the network returns to $\overrightarrow{RI}$.

If $RI\mathcal{E}_j=1$, then $RIA_j\Rightarrow B_j$, $0\Rightarrow F\mathcal{E}_j$; otherwise $RI\mathcal{E}_j=0$, then B_j and $F\mathcal{E}_j$ remain unchanged.

At last,examine $\overrightarrow{F\mathcal{E}}$.If $\overrightarrow{F\mathcal{E}}\neq(0,0,\cdots,0)$, then go to (i); otherwise the fetching operation is finished.

The procedure for storing operation is like that for fetching operation. But conflict-free storing operation does not need the return-routing.

If in Indirect Binary n-Cube, Ω, δ, or Baseline network, we only consider the permutation on the set of processor's addresses (i.e. memory bank addresses), then the destination address D_j only needs n bits. Data vector $\vec{A}$ moves from source address vector $\vec{S}=\vec{J}$ into vector register $\vec{R}$ according to destination address vector $\vec{D}$, that is just like a storing operation $\vec{A}\Rightarrow\vec{D}$. We can use the scheme described above to completely realize the permutation operation.

The iteration method is also applicable to other type single-stage interconnection networks. Using the method enables users to access to data vectors of the VCM without very carefully considering the various vector types. Vector access can be

realized automatically by machine. Conflict-free vector access only needs one routing iteration. The problem which is worth further studying is how to decrease the iteration number. For example, for Δ--ordered vector $\vec{Y}$ of N elements, if $\Delta=q\cdot 2^p$, q is odd, then the N elements distribute in $N/2^p$ memory banks. Every memory bank contains 2^p elements of $\vec{Y}$. Storing or fetching $\vec{Y}$ needs at least 2^p accesses to the VCM. In SP2I network, if the routing command order is always $+2^0,+2^1,\cdots,+2^{n-1}$ (both for routing and for return-routing), then the iteration number will be far greater than 2^p. But if we use the order $+2^{n-1},+2^{n-2},\cdots,+2^0$, for routing; the order $+2^0,+2^1,\cdots,+2^{n-1}$, for return-routing, then the iteration number will reach the lowest bound 2^p.

Conflict-free Implementation of Data Manipulation Functions

Conflict-free routing is one of the most important problems of interconnection network and has been well discussed in a number of papers [3,6,8,9, etc.]. Based on the properties of conflict-free routing discussed formerly, we will give some implementation methods in SP2I network for frequently used data manipulation functions.

1. Δ-Ordered Vector with Odd Δ and Reverse Vector

When $\vec{D}=D+\vec{J}\cdot\Delta$, and $\Delta=1 \pmod 2$, D and Δ are broadcast into all cells to compute $\vec{D}$. Then only one routing iteration is needed for realizing the access.

Suppose $\vec{Y}=(Y_0,Y_1,\cdots,Y_{K-1})$, $\vec{Y}'=(Y_{K-1},\cdots,Y_1,Y_0)$. Then $\vec{Y}'$ is called as the reverse vector of $\vec{Y}$. If $\vec{Y}$ is a Δ-ordered vector with odd Δ, then the $\vec{Y}'$ is also a Δ_1-ordered vector with odd Δ_1, where $\Delta_1=2^{m+n}-\Delta \pmod{2^{m+n}}$. The initial address of $\vec{Y}'$ is $D+(K-1)\Delta \pmod{2^{m+n}}$. Accessing to $\vec{Y}'$ is conflict-free. When $K\leq N$, and $\vec{Y}$ is canonical, we can get $\vec{Y}'$ through the following process:

(i) Compute the bit vector $\vec{\alpha}=(\alpha_0,\alpha_1,\cdots,\alpha_{N-1})$, where $\alpha_j=1$ (if $j<K$) or $\alpha_j=0$ (if $j\geq K$). This will be denoted as $\vec{\alpha}=[\vec{J}<K]$. Compute $\vec{\delta}=K-1-2\vec{J} \pmod N$.

(ii) $\vec{Y}\Rightarrow\overrightarrow{EA}$, $\vec{\delta}\Rightarrow\overrightarrow{E\delta}$, $\vec{\alpha}\Rightarrow\overrightarrow{E\xi}$. Perform n steps of routing. $\overrightarrow{EA}\Rightarrow\vec{B}$. Then $(B_0,B_1,\cdots,B_{K-1})=\vec{Y}'$.

2. Matrix Transposition

Suppose $A=(a_{i,j})_{K\times K}$, K is odd, the storage pattern makes every row vector be 1-ordered vector, every column vector be K-ordered vector. We want to get $B=A'=(b_{i,j})_{K\times K}$, where $b_{i,j}=a_{j,i}$, the storage pattern of B will be the same as that of A.

(a) Suppose A and B occupy the different memory space. Matrix transposition is to fetch a 1-ordered vector A[i,*] and to store it into the space of an K-ordered vector B[*,i], i=0,1,$\cdots$,K-1, i.e., to perform the following program[1]:

$0\Rightarrow i;[_K;[^*_K;A[i,*]\Rightarrow\vec{R}_1;\vec{R}_1\Rightarrow B[*,i];]^*;i+1\Rightarrow i;]$

(b) Suppose B will occupy the space of A. $\vec{R}_1$ and $\vec{R}_2$ are used as two vector working registers, the address of $a_{i,i}$ is used as initial address for both the i-th row vector and the i-th column vector to be fetched and stored. Perform the program:

$K\Rightarrow K';0\Rightarrow i;[_{K-1};[^*_{K'};A[i,*]\Rightarrow\vec{R}_1;A[*,i]\Rightarrow\vec{R}_2;$

$\vec{R}_2\Rightarrow A[i,*];\vec{R}_1\Rightarrow A[*,i];]^*;i+1\Rightarrow i;K'-1\Rightarrow K';]$

where the initial address of A[i,*] and A[*,i] is $D_i=D+iK+i$, D is the initial address of A.

(c) The case of $A=(a_{i,j})_{N\times N}$. As shown in Fig.4, transposition is done by exchanging of $\vec{a}_i$ and $\vec{b}_i$ which are the higher and lower subdiagonal vectors with the diagonal vector $\vec{a}_0=\vec{b}_0$ as the axis of symmetry.

If the global address of $a_{0,0}$ is $D'\cdot 2^n$, then $\vec{a}_0$ is canonical and has local address vector $\vec{D}'=D'+\vec{J}$. The corresponding elements of $\vec{a}_i$ and $\vec{a}_{i+1}$ have the same local address, but their memory bank addresses are different by 1. On the contrary, the corresponding elements of $\vec{b}_i$ and $\vec{b}_{i+1}$ have the same memory bank address, but their local addresses are different by 1. So we can

adopt the following transposition process:

(i)Broadcast D'. D'+$\vec{J}\Rightarrow\vec{\beta}_1$ and $\vec{\beta}_2$. 1$\Rightarrow$i.

(ii) $[\vec{J}\geq i]\Rightarrow\vec{\alpha}_1$. $[\vec{J}<N-i]\Rightarrow\vec{\alpha}_2$.

(iii) $\vec{\beta}_1\Rightarrow\overrightarrow{ED'}$. "Parallelly routing $+2^O$". $\overrightarrow{ED'}\Rightarrow\vec{\beta}_1$. Fetch out $\vec{a}_i$ (whose local address vector is just in $\vec{\beta}_1$) and send it into $\overrightarrow{EA}$."Parallelly routing +(N-i)".$\overrightarrow{EA}\Rightarrow\vec{R}_1$.

(iv) $\vec{\beta}_2+1\Rightarrow\vec{\beta}_2$. Fetch out $\vec{b}_i$ (its local address vector is just in $\vec{\beta}_2$) and send it into $\overrightarrow{EA}$. "Parallelly routing +i". $\overrightarrow{EA}\Rightarrow\vec{R}_2$.

(v) Under the control of $\vec{\alpha}_2,\vec{R}_1\Rightarrow(\vec{\beta}_2)$; Under the control of $\vec{\alpha}_1$, $\vec{R}_2\Rightarrow(\vec{\beta}_1)$.

(vi) i+1$\Rightarrow$i. If i$\geq$N, then end; otherwise go to (ii).

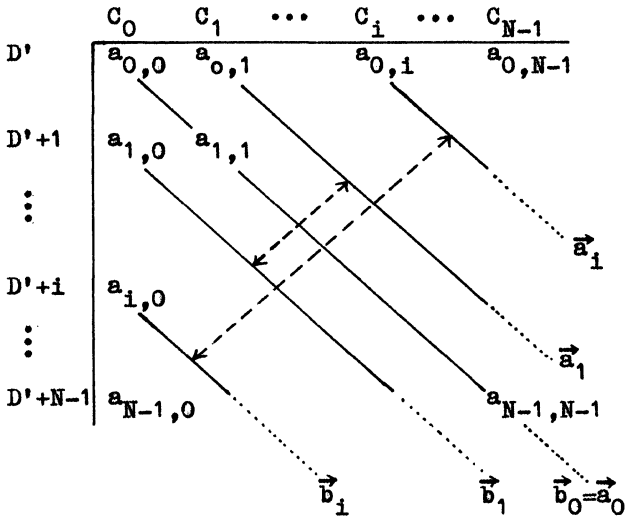


Fig.4, Transposition of $A=(a_{i,j})_{N\times N}$

3. Compressing and Spreading

Compressing is to save memory space. Only non-zero elements of a vector are stored into memory. Suppose $\vec{\beta}=(\beta_0,\beta_1,\cdots,\beta_{K-1})$ consists of the subscripts of non-zero elements of $\vec{Y}=(Y_0,Y_1,\cdots,Y_{N-1})$, K$\leq$N, $0\leq\beta_0<\beta_1<\beta_2<\cdots<\beta_{K-1}\leq N-1$.Compressing vector is denoted as $[\vec{\beta},\vec{Y}]=(Y_{\beta_0},Y_{\beta_1},\cdots,Y_{\beta_{K-1}})$. Spreading is the inverse operation of compressing,a spreading vector(sparse vector) is denoted as $(\vec{\beta},\vec{Z})=(0,\cdots,0,Z_0,0,\cdots,0,Z_1,0,\cdots,0,Z_{K-1},0,\cdots,0)$.

Suppose $\vec{\beta},\vec{Y}$, $[\vec{\beta},\vec{Y}]$ are all canonical. Because the SP2I network only has the "routing $+2^i$" function, i.e.,right-routing function,compressing to left is implemented by right-round-routing. If for i=0,1, ···,t-1, β_i=i; for k$\geq$t, β_k>k, this implies $Y_0,Y_1,\cdots,Y_{t-1}$ are non-zero elements, they do not need to move in compressing. But the moving of Y_{β_K} (k$\geq$t) may cause the Y_i (0$\leq$i$\leq$t-1) to disappear. To avoid this conflict,we first compress $\vec{Y}$ to right end, then parallelly transmit them to left end. As shown in Fig.5, to compress $\vec{Y}$ to right

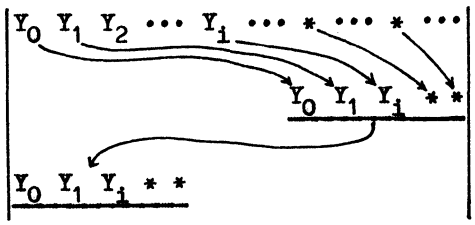


Fig.5, Compress routing

end satisfies the conditions of Theorem 4. On the command order of $+2^0,+2^1,\cdots,+2^{n-1}$, there is no routing conflict. Below is the compressing procedure:

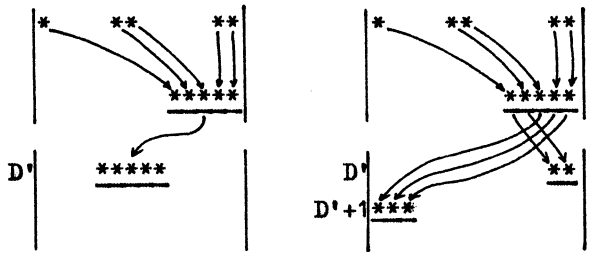
(a) Broadcast N-K. $[\vec{J}<K]\Rightarrow\vec{\alpha}$. Compute $\vec{\beta}-\vec{J}\Rightarrow\vec{\delta}^{**}$, N-K-$\vec{\delta}^{**}\Rightarrow\vec{\delta}^{*}$.

(b) $\vec{\delta}^{*}\Rightarrow\overrightarrow{EA},\vec{\delta}^{**}\Rightarrow\overrightarrow{E\delta},\vec{\alpha}\Rightarrow\overrightarrow{E\xi}$.Perform n steps of routing $+2^{n-1},+2^{n-2},\cdots,+2^0$, this is conflict-free according to Theorem 5.$\overrightarrow{EA}\Rightarrow\overrightarrow{E\delta}$.

(c) $\vec{Y}\Rightarrow\overrightarrow{EA}$. Perform n steps of routing $+2^0,+2^1,\cdots,+2^{n-1}$.

(d) "Parallelly routing +K" . $\overrightarrow{EA}\Rightarrow\vec{B}$. Then $(B_0,B_1,\cdots,B_{K-1})=[\vec{\beta},\vec{Y}]$.

If we want to store $[\vec{\beta},\vec{Y}]$ into the memory space D,D+1,D+2,···, where the initial address D=D'·2^n+D" , as shown in Fig.6, there are two cases.We only need to



(1) D"$\leq$N-K (2) D">N-K

Fig.6, Compressed into memory

modify the (d) of the above procedure as follows:

(After the process of (c))
"Parallelly routing +(D"+K)" . $D' \Rightarrow \overrightarrow{ED'}$. $[\vec{J}<D"] \Rightarrow \vec{\alpha}$. Under the control of $\vec{\alpha}$, $D'+1 \Rightarrow \overrightarrow{ED'}$. Then under the control of $\overrightarrow{E\mathcal{E}}$, $\overrightarrow{EA} \Rightarrow (\overrightarrow{ED'})$.

The implementation process of spreading vector $(\vec{\beta},\vec{Z})$ may be:

(a) $0 \Rightarrow \vec{B}$. $[\vec{J}<K] \Rightarrow \vec{\alpha}$. $\vec{\beta}-\vec{J} \Rightarrow \vec{\delta}$.

(b) $\vec{Z} \Rightarrow \overrightarrow{EA}$, $\vec{\delta} \Rightarrow \overrightarrow{E\delta}$, $\vec{\alpha} \Rightarrow \overrightarrow{E\mathcal{E}}$. Perform n steps of routing $+2^{n-1},+2^{n-2},\cdots,+2^{0}$. Under the control of $\overrightarrow{E\mathcal{E}}$, $\overrightarrow{EA} \Rightarrow \vec{B}$. Then $\vec{B}=(\vec{\beta},\vec{Z})$.

4. $\sqrt{N}\times\sqrt{N}$-Square-Block Vector, Fanout, Replicate

Here we suppose $N=2^{n}=2^{2t}$, $\sqrt{N}=2^{t}=s_1$. In order to thoroughly utilize the ability of parallel computation, in some cases, we can cut a matrix A into many $\sqrt{N}\times\sqrt{N}$-square-blocks,and every block as a "block vector" may be parallelly computed by N processors.

Suppose $A=(a_{i,j})_{K\times K}$, $\lceil K/s_1 \rceil = c$. Let $q=c$ (if c is odd) or $c+1$ (if c is even), and $\Delta=q\cdot s_1$. A is stored into memory in the following storage pattern: for $i=0,1,\cdots,K-1$, the address vector of the data vector $(a_{i,0},a_{i,1},\cdots,a_{i,K-1},0,0,\cdots,0)$ is $(D+i\Delta, D+i\Delta+1,\cdots,D+i\Delta+K-1,D+i\Delta+K,\cdots,D+i\Delta+\Delta-1)$, where D is the address of $a_{0,0}$, D may be arbitrary.Then the square block vector $\vec{A}_{I,J}$ is defined as $(a_{i,j},a_{i,j+1},\cdots,a_{i,j+s_1-1}, a_{i+1,j},a_{i+1,j+1},\cdots,a_{i+1,j+s_1-1},\cdots\cdots, a_{i+s_1-1,j},a_{i+s_1-1,j+1},\cdots,a_{i+s_1-1,j+s_1-1})$, whose address vector is $\vec{D}=D_{i,j}+H(t,\vec{J})\cdot\Delta +L(t,\vec{J})$, where $H(t,\vec{J})=(H(t,0),H(t,1),\cdots, H(t,N-1))$, $L(t,\vec{J})=(L(t,0),L(t,1),\cdots, L(t,N-1))$, $D_{i,j}$ is the address of $a_{i,j}$. It is easy to prove that $\vec{\delta}=\vec{D}"-\vec{J}$ satisfies the condition (*), so according to Theorem 1, $\vec{A}_{I,J}$ has no routing conflict.

Suppose $\vec{Y}=(Y_0,Y_1,\cdots,Y_{s_1-1})$.The fanout vector $\vec{F}(\vec{Y})$ is defined as $(Y_0,Y_0,\cdots,Y_0, Y_1,Y_1,\cdots,Y_1,\cdots\cdots,Y_{s_1-1},Y_{s_1-1},\cdots,Y_{s_1-1})$. The replicate vector $\vec{R}(\vec{Y})$ is defined as $(Y_0,Y_1,\cdots,Y_{s_1-1}, Y_0,Y_1,\cdots,Y_{s_1-1},\cdots\cdots, Y_0,Y_1,\cdots,Y_{s_1-1})$. Both $\vec{F}(\vec{Y})$ and $\vec{R}(\vec{Y})$ have $s_1^2=N$ elements.

Suppose $\vec{Y}$ is canonical. The $\vec{F}(\vec{Y})$ may be obtained as follows:

(a) $[\vec{J}<\sqrt{N}] \Rightarrow \vec{\alpha}$. $(\sqrt{N}-1)\cdot\vec{J} \Rightarrow \vec{\delta}$.

(b) $\vec{Y} \Rightarrow \overrightarrow{EA}$, $\vec{\delta} \Rightarrow \overrightarrow{E\delta}$, $\vec{\alpha} \Rightarrow \overrightarrow{E\mathcal{E}}$. Perform n steps of routing $+2^{n-1},+2^{n-2},\cdots,+2^{0}$.

(c) Perform t steps of broadcast-type routing $\underline{+2^{0},+2^{1},\cdots,+2^{t-1}}$. $\overrightarrow{EA} \Rightarrow \vec{B}$.Then $\vec{B}=\vec{F}(\vec{Y})$.

Suppose $\vec{Y}$ is canonical. $\vec{R}(\vec{Y})$ may be obtained as follows:

(a) $[\vec{J}<\sqrt{N}] \Rightarrow \vec{\alpha}$.

(b) $\vec{Y} \Rightarrow \overrightarrow{EA}$, $\vec{\alpha} \Rightarrow \overrightarrow{E\mathcal{E}}$. Perform t steps of broadcast-type routing $\underline{+2^{t},+2^{t+1},\cdots,+2^{n-1}}$. $\overrightarrow{EA} \Rightarrow \vec{B}$. Then $\vec{B}=\vec{R}(\vec{Y})$.

Besides, irregular fanout and replicate may also be implemented without conflict. Due to the limitation of space, these materials are omitted here.

Conclusion

This paper has discussed in detail the various properties of conflict-free routing of SP2I interconnection network, they are the basis to find out implementation methods of data manipulation functions.The routing step number of implementation methods presented in this paper is $O(\log_2 N)$, the time needed for computing control-information is usually less than the routing time. SP2I network needs $O(N\cdot\log_2 N)$ gates. Thus, SP2I has several advantages: network structure and controlling are simple, transmission rate is high, hardware devices are limited in a reasonable range. SP2I may efficiently realize the main data alignment functions faced by the CVCVHP system. The iteration method of automatic vector-routing, which is used especially for solving routing conflict problems, enables SP2I to easily realize various dynamic and parallel data alignments. The extension of the iteration method may practically and efficiently

solve the routing conflict problems in AIM, Indirect Binary n-Cube, Ω, δ, Baseline, and other networks.

Acknowledgements

The authors gratefully acknowledge the most helpful comments of referees. Ms. Zheng Ya-Xian's patient and skillful typing is also sincerely appreciated.

References

[1] Gao Qing-Shi, Zhang Xiang, "A general--Purpose Cellular Supercomputer---Cellular Vector Computer of Vertical and Horizontal Processing with Virtual Common Memory", Chinese Journal of Computers, Vol.2, No.1, January 1979, pp. 1-13.

[2] Zhang Xiang, Gao Qing-Shi, "Principle of Automatic Vector-Routing and Its Iteration", Chinese Journal of Computers, Vol.4, No.6, November 1981, pp. 459-467.

[3] R. J. McMillen and H. J. Siegel, "MIMD Machine Communication Using the Augmented Data Manipulator Network", 7-th Annual Symp, Computer Architecture, May 1980, pp. 51-58.

[4] Gao Qing-Shi, Zhang Xiang, "Another Approach to Making Supercomputer by Microprocessors---Cellular Vector Computer of Vertical and Horizontal Processing with Virtual Common Memory", 1980 Int'l. Conf. on Parallel Processing, August 1980, pp. 163-164.

[5] M.C. Pease, "The Indirect Binary n-Cube Microprocessor Array". IEEE Trans. Comput., Vol. C-26, May 1977, pp. 458-473.

[6] D. H. Lawrie, "Access and Alignment of Data in an Array Processor", IEEE Trans.Comput., Vol. C-24, December 1975, pp. 1145-1155.

[7] J. H. Patel, "Processor-Memory Interconnections for Multiprocessors", 6-th Int'l. Annual Symp. Computer Architecture, April 1979, pp. 168-177.

[8] C. Wu and T. Feng, "Routing Techniques for a Class of Multistage Interconnection Networks", 1978 Int'l. Conf. on Parallel Processing, August 1978, pp. 197-205.

[9] C. Wu and T. Feng, "The Reverse-Exchange Interconnection Network", 1979 Int'l. Conf. on Parallel Processing, August 1979, pp. 160-174.

[10] H. J. Siegel, "Analysis Techniques for SIMD Machine Interconnection Networks and the Effects of Processor Address Masks", IEEE Trans. Comput., Vol. C-26, February 1977, pp. 153-161.

[11] H. J. Siegel, "A Model of SIMD Machines and a Comparison of Various Interconnection Networks", IEEE Trans. Comput., Vol. C-28, December 1979, pp. 907-917.

DISTRUBUTED CIRCUIT SWITCHING STARNET*

Chuan-lin Wu, Woei Lin and Min-Chang Lin

Department of Electrical Engineering
Austin, Texas 78712

Abstract -- Starnet is a communication subnet which can cost-effectively connect hundreds or thousands of processors for distributed processing. It uses distributed control and circuit switching. Starnet's communication medium includes two major components: a multistage interconnection network and a set of interface units. The interconnection network uses a destination routing scheme with no central control. The interface unit provides handshaking between the computer/data node and the interconnection network under the control of a microporcessor. Detailed design of the communication medium is described. A model for comparing cost-effectiveness among Starnet, crossbar and multiple buses is included.

I. Introduction

Starnet is a distributed circuit switching local communication subnet which can provide flexibility required to cost-effectively solve general distributed processing problems. The area of local computer network architecture is concerned with interconnecting two or more computers within a restricted area such as a single building or a small cluster of buildings, to facilitate high-performance distributed processing. Although there are many local computer networks proposed or currently existing [1], increasingly sophisticated technology and enlarged problem domain have spawned a need for investigating networks which can provide higher-speed information processing in a new environment enhanced by technological advances and new processing requirements.

It is our goal to design a reconfigurable subnet which allows partitioning connected resources into any connection topology(ies). According to our need of efficient distributed processing the communication subnet consists of an interconnection network and a set of interface units (IU's). A block diagram of the communication subnet is shown in Fig. 1. Shown in Fig. 1 is also a multiple-IU assignment to system components, called nodes. The subnet realizes protocols specified in the first three layers of a hierarchical network architecture model [2]. In the first layer, the subnet specifies the mechanical, electrical and functional characteristics required to connect, maintain, and disconnect a physical circuit in the paths between interface units. The second layer breaks data up into frames, provides destination address, transmits the frames, processes the acknowledgement from the receivers and handles errors if they appear. The last layer of the subnet handles controls of the subnet operation. Its key design issues among others are routing and flow control. The architecture of the subnet is to implement these functions specified in the three layers in terms of its components: interconnection network and interface units.

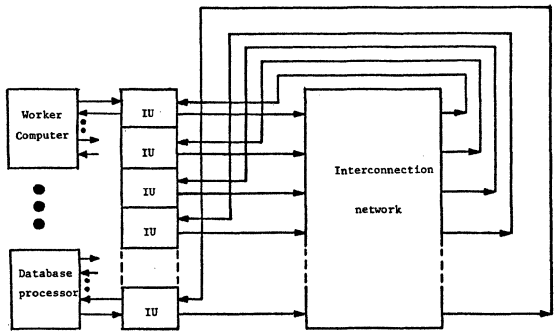


Fig. 1 A block diagram of Starnet

In section II, the design issues of the interconnection network are considered in two aspects: network topology and routing. Section III deals with switching elements of the interconnection network. The interface unit design is then exploited in section IV. An evaluation on the bandwidth and cost-effectiveness is presented in section V.

II. Network topology and routing

A modified topology of baseline network [3] is used. The baseline network is a multistage interconnection network which can provide a full-connection. By full-connection, we mean that there exists a direct connection for each input/output pair. Fig. 2 shows a generalized recursive process to generate baseline topology. In the recursive process, the first stage contains N/r switching elements of size r x t (i.e., r inputs and t outputs) where N is the number of inputs of the network. The second stage contains t subblocks: C_0, C_1,...., and C_{t-1}. The process can recursively be applied to the subblocks until each subblock can physically be realized by an off-the-shelf switching element. A topology

* This work was partially supported by Bureau of Engineering Research, The University of Texas, under grant, BER Research and Development 3074590780, and by University Research Institute, The University of Texas, under grant 20-7499-0868.

0190-3918/82/0000/0026$00.75 © 1982 IEEE

generated by setting r = t = 2 and N = 8 is shown in Fig. 3. The topology shows that there are three stages of 2 x 2 switching elements and there are 8 inputs and 8 outputs.

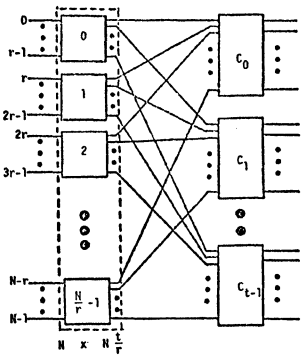


Fig. 2 A recursive process to generate baseline topology.

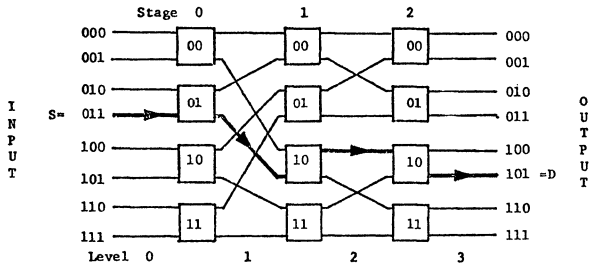


Fig. 3 A 8x8 baseline network topology

The label of the components (switching elements and links) of the interconnection network can be illustrated by the baseline network shown in Fig. 3. The stages are labelled in a sequence from 0 to n - 1 with 0 for the left most stage where $n = \log_2 N$. Similarly, the levels of links are labelled in a sequence from 0 to n. The switching element in stage i is labelled by $(S_{n-1}\ S_{n-2} \ldots S_1)_i$ where S_j, $1 \leq j \leq n-1$, is a base-r number. The input/output links of a switching element is labelled by $S_{n-1}\ S_{n-2} \ldots S_1\ S_0$ where $S_{n-1}\ S_{n-2} \ldots S_1$ denotes the label of the switching element from which the link spreads and S_0 denotes the relative location of the link in one side of the switching element. In Fig. 3, there are three stages (0, 1, 2), four levels (0, 1, 2, 3), four switching elements (00, 01, 10, 11) in each stage and 8 links (000, 001, ..., 111) in each level.

There is one and only one path existing between a source (input) and a destination (output). The unique path can be decided according to the destination address. Suppose that a source S is to be connected to a destination D whose address can be represented by a base-r number $D_{n-1}\ D_{n-2} \ldots D_1 D_0$. To connect the source to the destination, the switching element to which the source is attached, will take D_{n-1}, and connect source to the next stage in terms of its D_{n-1} th output link. The switching element in the next stage will then take the next number in the destination, D_{n-2}, and use its D_{n-2} th output link to connect the source to the next stage. The process is repeated until it reaches the destination. For example, assume that source S is to be connected to destination D, 101 in Fig. 3. Then switching element $(01)_0$ will take the left most bit of D, 1, and switches the source to switching element $(10)_1$ using the lower output link. Switching element $(10)_1$ will then take the next bit of D, 0, and switches the source to switching element $(10)_2$ using the upper output link. The last bit of D, 1, will then be used by switching element $(10)_2$ to connect the source to the destination.

For the purpose of fault diagnosis and subnet reconfiguration, it is necessary to know the routing path. The routing path can be derived from source and destination addresses. Let $S = S_{n-1}\ S_{n-2} \ldots S_1 S_0$ and $D = D_{n-1}\ D_{n-2} \ldots D_1 D_0$ as used in the above paragraph. Then the switching elements on the path from the source S to the destination D can be denoted by $(D_{n-1}\ D_{n-2} \ldots D_{n-i}\ S_{n-1}\ S_{n-2} \ldots S_{i+1})_i$ for $0 \leq i \leq n-1$. In other words, the links on the path can be denoted by $(D_{n-1}\ D_{n-2} \ldots D_{n-i}\ S_{n-1}\ S_{n-2} \ldots S_{i+1}\ D_{n-i-1})_{i+1}$ for $0 \leq i \leq n-1$. In the routing example shown in Fig. 4, S = 011 and D = 101 and switching elements $(01)_0$, $(10)_1$ and $(10)_2$ are on the path.

Since there is one and only one path existing for an I/0 pair, the I/0 pair can not be connected in case that there is a faulty element in the path or a conflict of using the switching elements and/or links occurs. In order to provide the fault tolerance and higher availability, an extra stage is added to the baseline network. A modified baseline topology is shown in Fig. 4. The extra stage added and the baseline network will allow two connection paths for an I/0 pair as shown in Fig. 4. In general, if the switching element used in the extra stage has t outputs, an I/0 pair can have t connection paths.

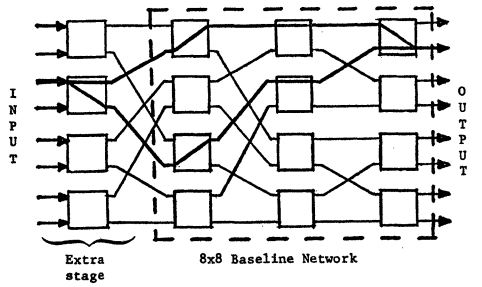


Fig. 4 A modified 8x8 baseline network.

The routing scheme is the same as the one described above except for the portion of the extra stage. Since any output link of the switching element to which the source is attached in the extra stage will lead to the destination, the source can select one of the output links according to the priority policy and/or the state of the subnet. After an output link of the extra stage is selected, the aforementioned routing can then be followed to establish a path to the destination.

III. Design of fault tolerant switching element

Interconnection network is designed in a way that it can be constructed modularly in terms of a single type of switching element. The switching element realizes communication protocols which specify control strategy and switching methodology [4]. In addition to the protocols, fault tolerance is justified by the fact that circuit complexity of the subnet can be at the same level as the complexity of the other part of the system. It is likely fair to say that a reliable subnet is even more critical than other reliability issues. Here we describe a 2 x 2 fault tolerant switching element which is to be used to modularly construct the interconnection network. The switching element uses distributed control and circuit switching. Its pin and gate count stays in the implementable range of VLSI technology.

A block diagram of the switching elements is shown in Fig. 5. The switching element comprises of two major parts: control plane and data plane, which deal with control and data respectively. The control plane does the handshaking process in establishing connection paths. It generates control signals which are fed into the data plane to connect data ports. The data plane is the part where the data communication actually occurs. Depending on the word length required, the number of the data planes to be connected can accordingly be adjusted.

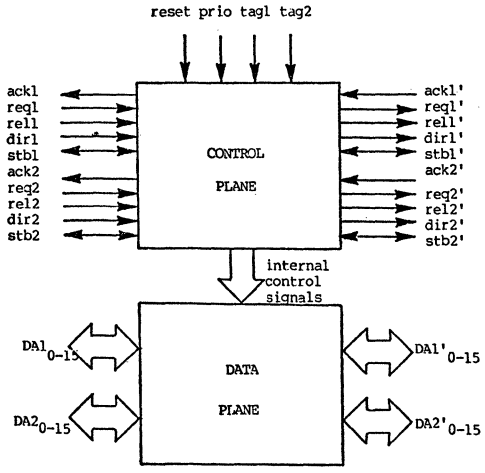


Fig. 5 A block diagram of a 2x2 switching element

The major function of the control plane is to set up the path between the source and the destination according to the routing scheme. As shown in Fig. 5, the control plane has the following input control lines: address tags of destinations (tag1, tag2), request lines (req1, req2), acknowledge signals (ack1', ack2') release signals (rel1, rel2), direction indicators (dir1, dir2), strobe lines (stb1, stb2), priority signal (Prio) and reset signal (reset). The output control lines are relayed request lines (req1', req2'), acknowledge signals (ack1, ack2), direction indicators (dir1', dir2'), and strobe lines (stb1', stb2'). The control circuit receives input signals from two switching elements in the previous stage, generates control signals for the data plane, modifies the input signals and relays them to the next stage. The control plane has four internal registers to record the current connection status of the switching element. It generates internal signals to facilitate conflict resolution and data plane control. With this design, a physical path of the modified baseline network can be established in one clock period, which include two clock phases ϕ_1 and ϕ_2. During phase ϕ_1, depending on the routing tag and the current state of the switching element, the request will be rippled down stage by stage.

If no conflict occurs along the requested path, an acknowledge signal will be returned from the receiver. During clock phase ϕ_2, each switching element along this allocated conflict-free path would update their internal registers and set up the related connection in the data palne. At the end of phase ϕ_2, a physical path is actually established between the source and the destination if there is no conflict. The established path can remain still as long as the source wishes. The source can issue the release signal to disconnect the path. Before a better fault tolerance scheme can be finalized, Triplicated Modular Redundance, in which the circuit is triplicated, is used to enhance the reliability.

The data plane is composed of a data circuit and a test circuit. The data circuit comprises of a number of duplicated copies each of which can have a general 2 x 2 crossbar connection for full-duplex communication. The test circuit always monitors the response to on-line data and perform self-diagnosis asynchronously. If a fault occurs, the test circuit will adopt a recovery procedure to reconfigure the data circuit. The test circuit comprises of three parts: test data generator, match detector, and recovery control logic. Test data generator provides the idle link with auxiliary test data in addition to on-line data for fault detection. Match detectors compare input data (either on-line data or auxiliary test data) to the associated outputs. If there exists a mismatch, the corresponding error flag will be raised which will then trigger recovery control logic to replace the current copy of the data circuit with an error-free copy.

The switching element can perform bidirectional fault-tolerant communication and broad-

casting. The data propagation delay per switching element is estimated to be 30 ns. 88 pins are required for a switching element which allows 16-bit parallel half-duplex communication. The design is viable for VLSI implementation. The gate-level design details can be found in [5].

IV. Interface unit

The system component is attached to the interface unit which in turn connects to multiple ports of the interconnection network. The functions of the interface unit can be divided into two parts. Part 1 provides mechanisms for enabling communication between the system component and the interface unit. Its design depends on the system component. Part 2 facilitates accesses to the interconnection network. The design of the second part concerns the interaction to the interconnection network and is independent of the attached system component.

Part 2 contains active and passive connection ports. The active port is connected to the input of the interconnection network while the passive port is connected to the output. Only the active port can initiate connection request. However, both active and passive ports can transmit data after a connection path is established. When an active ports needs to access a passive port, it places the address bits of the destination ports on proper data lines and raises the request signal. Through the routing procedure (in a network clock cycle), the request is either accepted or rejected as signaled by the acknowledge line. If rejected, try again or the alternate path. If accepted, a path is already established and the interface unit then starts the data communication using the handshaking process and performs error checking and possible error correction. Input and output buffers are provided for each port. The main function of this part is to provide a reliable data communication. The higher level protocols are implemented in the first part of the interface unit.

As shown in Fig. 6 for a connection between a master computer node and a slave computer node, the input/output lines of the IU can be divided into two groups:

(1) IU to interconnection network - This side is directly connected to the interconnection network. An interface unit has two kinds of connection to the interconnection network: the upper links represneting the active port where the master node initiates the request to the slave node; the lower links representing the passive port where the slave node sends the reply to the master node.

(2) Computer nodes to IU - The IU can be treated as an I/O device of computer nodes. Node will direct IU by passing orders via hardwired interrupt lines and control codes contained on data bus. An IU requests nodes for service in a similar fashion.

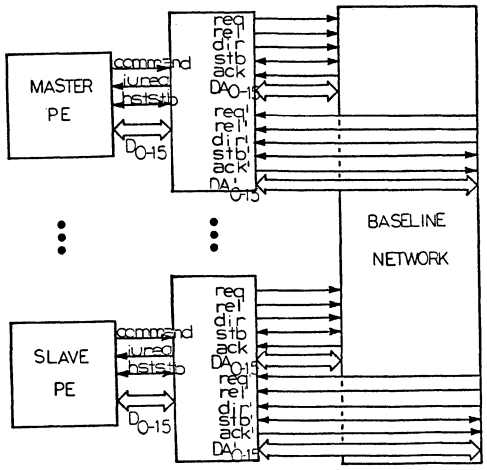


Fig. 6 Connection of interface units

In this section we present an interface unit design, which is based on the 2900 family [6]. The configuration of the bit-slice based IU design is illustrated in figure 7. The 2900 family components employed in this design include (1) CPU-ALU and scratchpad register units, Am2901; (2) microprogram sequencer and controller, Am2910; (3) bipolar memory, Am2960 Series; (4) interrupt controller and support devices, Am2914; and (5) condition code multiplexer, Am2922.

The IU internal structure as illustrated in Fig. 7 is elaborated as follows.

(1) ALU (Am2901). With the 9 bits of microword, ALU is capable of selecting source operands, functions, destination registers and providing various status outputs.

(2) Microprogram Controller (Am2910): This microprogram controller is an address sequencer that is intended for controlling the sequence stored in the microprogram memory. Beside the capability of sequential access, it provides conditional branch and five levels of nesting microroutines.

(3) Pipeline Register: Pipeline register is used to improve the execution speed. It is added at the PROM output to allow the overlap of ALU operation and memory fetch process.

(4) Interrupt Controller (Am2914): The Am2914 interrupt controller may be connected to provide the capability of microprogram level interrupt. The occurrence of an interrupt causes a branch address, which is provided by mapping PROM, to be fed into microprogram controller. Such a vectored interrupt suspends the current routine and activates a specific interrupt service microroutine. After the interrupt service routine is finished, the suspended routine will be resumed.

(5) Microprogram Memory (Am2960): These PROM's

are used to store a number of interrupt-driven microroutines, which handle various stimuli from outside.

(6) Interrupt Mapping PROM (Am29751 and Am2913): These PROM's supply the 12 bit starting address of a specific microroutine according to the type of interrupt.

(7) Condition Code multiplexer (Am2922): This multiplexer selects the desired status and feeds it into microprogram controller.

(8) Control Register (CR): This 16-bit register is used to control data flow inside IU.

(9) Output Data Register (ODR): This 16-bit register holds the data to be exported to active port and/or passive port.

(10) Input Data Register (IDR): This 16-bit register holds the data imported from active port or passive port.

(11) Control Signal Generator (CSG): The CSG, composed of an 8-bit register and an 8-bit driver, drives handshaking signals, strobe signals, and request signals which are generated from pipeline register.

(12) Active Port Interface and Passive Port Interface (API and PPI): These two transceivers interface the internal output data bus to the external data buses DA_{0-15} and DA'_{0-15} respectively.

(13) Data Bus Driver DBD): This transceiver interfaces the internal input data bus to the external data bus D_{0-15} which is connected to the host.

(14) Bypass Switch (BS): This switch is used to bypass the 16-bit data transferred to IU without microprogram interferring.

(15) Strobe Switch (SS): As shown in Figure 4-7, this switch is used to bypass the strobe signals transferred to the interface unit without microprogram interferring.

In the microprogram storage, there are a number of microroutines, which are invoked by the interrupt from outside. Each microroutine consists of a series of microinstructions that contain the control information over some hardware elements. Every time an interrupt occurs, a specific microroutine would be activated and executed by gating succesive microinstructions into pipeline register. Via the dedicated connections to the corresponding hardward components, these microinstructions initiate a sequence of hardware activities and make the interface unit be able to react properly. In correspondence to the hardware components of the interface unit, the microinstruction can be divided into 11 different fields. Each field is in charge of a specific hardware element. The width, 64 bits, is suitable for a Am2900 family based design.

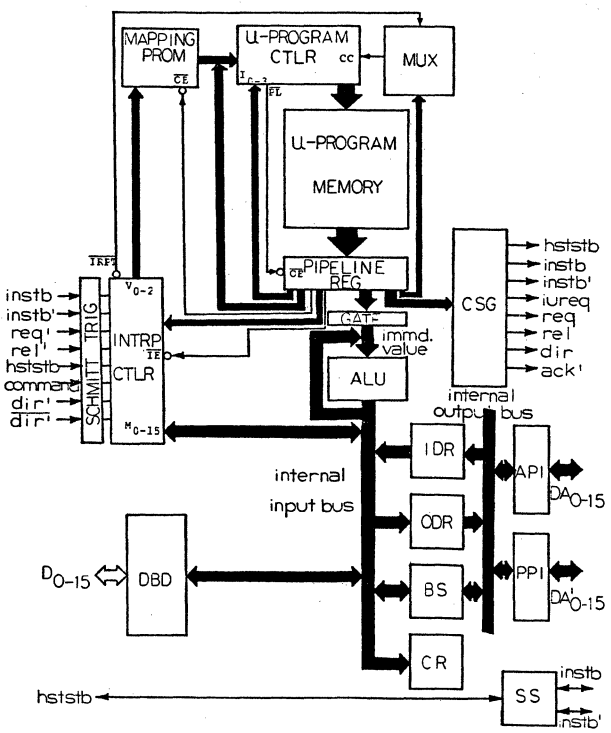


Fig. 7 Configuration of interface unit

V. Performance Evaluation

In this section, two performance factors: bandwidth and cost-effectiveness (bandwidth/cost), are examined to compare bus-structure network [7], Starnet, and crossbar switch. Both asynchronous and synchronous modes of operations are considered here. Following assumptions are made for these two modes of operations.

Asynchronous mode:

(1) Poisson arrival and exponential distribution service time for messages are assumed; each input link has the same arrival rate λ, average message transmission time is $1/\mu$.

(2) The time to setup a path is assumed to be small compared to message transmission time and can be neglected.

(3) Unbuffered systems are assumed; a request will be discarded if it is blocked of if it arrives at a busy input link.

Synchronous mode:

(1) Message length is fixed; the time to transmit a message plus the time to set up a path is defined as a cycle.

(2) Each input link generates requests for message transmission randomly and independently; the destinations of requests are uniformly distributed over all output links.

(3) The requests are generated synchronously; at the beginning of each cycle, each input link generates a request with the same probability Po.

(4) The requests being blocked are discarded; the requests generated at next cycle are assumed to be independent to the previous ones.

1. Bus-structure network analysis

A network with $\log_2 N$ buses is used for comparison, where N is equal to the number of computer nodes to be connected. Since the complexity of a unibus system can be approximated as of order N, the complexity or $\log_2 N$-bus system is about $O(N*\log_2 N)$. For simplicity, we assume that under both asynchronous and synchronous modes of operations, the $\log_2 N$-bus system is able to acheive its perfect condition which has bandwidth:

m, when the number of requests of the system at a particular time is m, $m<\log_2 N$;
$\log_2 N$, when $m>\log_2 N$.
where $(0\leq m\leq N)$

2. Analysis for baseline network and crossbar switch

Crossbar switch can be thought as a special case of baseline network with one stage and an NxN switching element. The analysis of baseline network thus can be applied to crossbar switch with slight modification.

Asynchronous operation:

The asynchronous operation of unbuffered baseline network is assumed to have the continuous time Markovian behavior, which means that the transitions of system states are timely continuous and the rate of transition to the next state depends on current state only.

Fig. 8 shows the Markov chain of this model, the number of currently accepted requests is chosen as the state parameter. A new arrival of acceptable request changes state i to state i+1, with rate $(N-i)\cdot\lambda\cdot PP_N(i)$, i=0,1,....,N-1; a departure of currently accepted request changes state i to state i-1, with rate $i\cdot u$, i=1,2,....,N where $PP_N(i)$ is the probability that a new request will be accepted when a size-N network currently has i accepted requests (i paths currently exist). If all $PP_{N/2}(i)$'s are known, $PP_N(i)$ can be found in Eq. (1) where $R_{n,m}$ is the total number of possible arrangements of m paths in a size-N network.

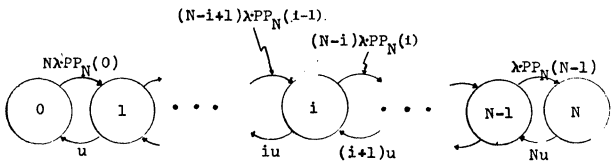


Fig. 8 Markov chain for asynchronous baseline operational model

To find $PP_N(i)$'s, we can start from the boundary condition $PP_2(0)=1.0$, $PP_2(1)=0.5$, $PP_2(2)=0.0$. Once the $PP_N(i)$'s are found, the equilibrium probabilities, ST(i)'s, can be solved easily, and the bandwidth of the system is

$$\sum_{i=1}^{N} i\cdot ST(i).$$

Crossbar switch in asynchronous mode is analyzed with the same model. The only change is that $PP_N(i)=(N-i)/N$ for i=0,1,....,N-1.

Synchronous operation:

The synchronous operation of baseline network and crossbar switch are modeled by assuming each input link generate a request with probability P_o in every cycle. The probability that i requests are accepted at one cycle can be found in Eq. (2).

$$PP_N(i) = \frac{\sum_{\substack{2j+k=i \\ j+k+\lambda=N/2}} \binom{N/2}{j,k,\lambda} 2^{j+k} \sum_{r=0}^{k} \binom{k}{r}\left[1-\frac{r}{2\lambda+k}\right]\cdot PP_{\frac{N}{2}}(j+r)\cdot \frac{R_{N/2,j+r}}{\binom{N/2}{j+r}} \frac{R_{N/2,j+k-r}}{\binom{N/2}{j+k-r}}}{\sum_{\substack{2j+k=i \\ j+k+\lambda=N/2}} \binom{N/2}{j,k,\lambda} 2^{j+k} \sum_{r=0}^{k} \binom{k}{r} \frac{R_{N/2,j+r}}{\binom{N/2}{j+r}} \frac{R_{N/2,j+k-r}}{\binom{N/2}{j+k-r}}} \qquad \text{Eq. 1}$$

$$PA(i) = \sum_{m=i}^{N} \binom{N}{m} P_o^m (1-P_o)^{N-m} \cdot \sum_{r_1+r_2+\ldots r_i=m-i} PP_N(0)\cdot PP_N(1)\cdots PP_N(i-1)(1-PP_N(1))^{r_1}(1-PP_N(2))^{r_2}\cdots(1-PP_N(i))^{r_i} \qquad \text{Eq. 2}$$

The first summation term is the probability that m requests are generated at one cycle, the second summation term is the conditional probability that i requests are accepted when m requests are generated at one cycle, note that $PP_N(0)$ is equal to 1 for every N.

The bandwidth of the system is $\sum_{i=1}^{N} i \cdot PA(i)$, crossbar switch has a simplified form $N \cdot [1-(1-\frac{P_o}{N})^N]$.

3. Comparison

Bandwidth:

The bandwidths of the three networks are shown in Fig. 9. The analysis of baseline network and corssbar switch is verified by simulation for N=4,8,16. We can see from the figure that the bus network is not suitable when N is large, its bandwidth is limited by the number of buses. Increasing the number of buses is not practical since the increasing of one bus not only increases the cost of order N, but also incurs more scheduling problems. The bandwidths of both baseline and crossbar networks are of order N, the bandwidth of baseline network falls down slowly as N increases, the bandwidth of crossbar switch reaches a lower limit as N increases to infinity.

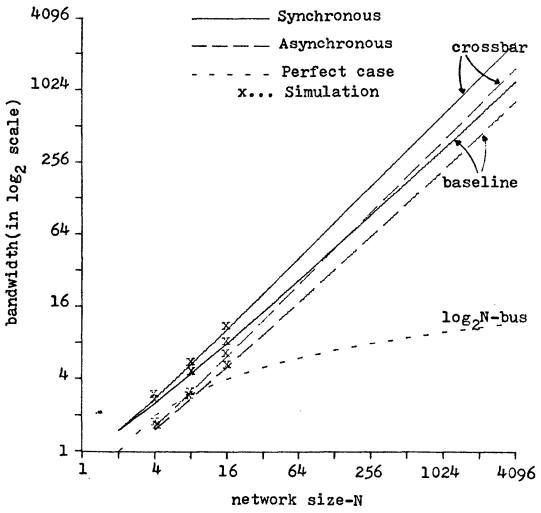


Fig. 9(a) Bandwidth of size-N networks, P_o=1.0 for synchronous mode, λ/u=1.0 for asynchronous mode.

Cost effectiveness:

To compare the cost-effectiveness, we assume that the control mechanisms of all three networks are implemented by hardware. The number of crosspoints of each network system is chosen as the cost index: bus system has $N \cdot \log_2 N$ crosspoints, baseline network has $2 \cdot N \cdot \log_2 N$ crosspoints and crossbar switch has $N \cdot N$ crosspoints.

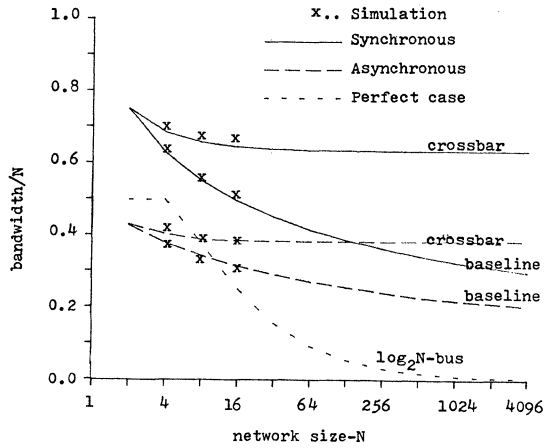


Fig. 9(b) Bandwidth of size-N networks, P_o=1.0 for synchronous mode, λ/u=1.0 for synchronous mode.

This choice of cost index favors bus and crossbar networks since the control logic for each crosspoint of these two networks is more complex than that of baseline network. The result is shown in fig 10. It can be found that baseline is the most cost-effective when $N \geq 64$. The bus structure network is not able to support a large system for both reasons: few bandwidth available and not cost-effective. The crossbar has poor cost-effectiveness when N is large, also the tremendous complexity makes it very difficult to implement a crossbar switch with size over one hundred.

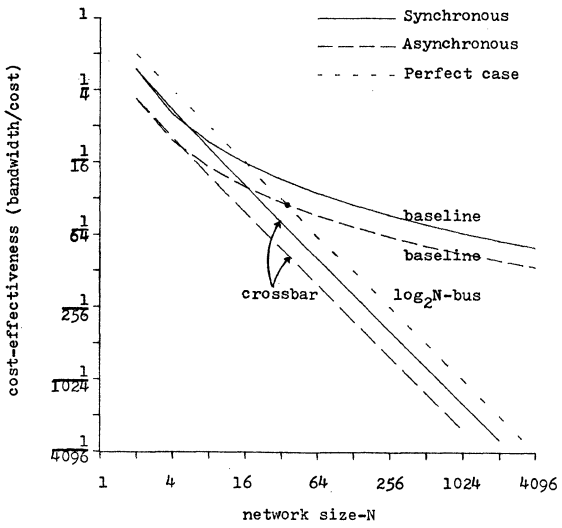


Fig. 10 Cost-effectiveness of size-N networks, P_o=1.0 for synchronous mode, λ/u=1.o for asynchronous mode.

VI. Conclusion

This design shows that Starnet can provide a data access time of 1 microsecond in a local computer network with over one thousand computer/

data nodes. The adequate transport mechanism can thus provide better coupling among its nodes, compared to some contemporary multiprocessing systems. With its enchanced reliability and reconfigurability, Starnet has potential in being used for various applications including large scale real-time computation and office information systems.

References

[1] K.J. Thurber and H.A. Freeman, "Architecture considerations for local computer networks," Proc. 1st International Conference on Distributed Computing Systems, 1979, pp. 131-142.

[2] H. Zimmerman, "OSI reference model - The ISO model of architecture for open systems interconnections," IEEE Trans. Commun., Vol. COM-28, April 1980, pp. 425-432.

[3] C. Wu and T. Feng, "On a class of multistage interconnection networks," IEEE Trans. Comput., Vol. C-29, Aug. 1980, pp. 694-702.

[4] T. Fend, "A survery of interconnection networks," Computer, Dec. 1981 pp. 12-27.

[5] W. Lin and C. Wu, "Design of a 2x2 fault-tolerant switching element," Proc. 9th Annual Symp. on Computer Architecture, 1982.

[6] J. Mick and J. Brick, Bit-slice microprocessor design, McGraw-Hill, Inc. 1980.

[7] R.M. Metcalfe and D. R. Boggs, "Ethernet-Distributed packet switching for local computer networks," CACM, July 1976, pp. 395-404.

COMPARATIVE STUDY OF THE EXPLOITATION OF DIFFERENT LEVELS OF PARALLELISM ON DIFFERENT PARALLEL ARCHITECTURES

R.H. Barlow, D.J. Evans & J. Shanehchi
Department of Computer Studies
Loughborough University of Technology
Loughborough, Leicestershire,
U.K.

Abstract

This paper considers various levels of parallelism obtainable from sequential solutions for locating the eigenvalues of real symmetric tridiagonal matrices based on the bisection algorithm coupled with Sturm sequence evaluation. Three levels of parallelism are identified and the implementation of these three levels on three different parallel computer architectures is described. The three computer systems are a vector processor (the CRAY-1), an array processor (the ICL Distributed Array Processor) and an asynchronous multiprocessor consisting of 4 minicomputers linked through shared memory. Results presented confirm the theoretical analysis and show that one of the levels of parallelism, based on converting a standard linear recurrence relation is of use only for locating small numbers of eigenvalues using large number of processors. The other two levels, when combined, yield an effective algorithm for locating any number of eigenvalues on all three types of computer.

0. Introduction

The application considered here is the determination of the eigenvalues of a real symmetric tridiagonal matrix.

This is an extremely important problem as standard sequential eigenvalue solvers first transform a real symmetric matrix to tridiagonal form by similarity transformations: the eigenvalues of the resultant tridiagonal matrix are identical to those of the original matrix.

The original problem of N eigenvalues on interval R yields on evaluation of the associated Sturm sequences at m interior points of R, up to m+1 similar problems on smaller intervals. Repeated application of the technique isolates the eigenvalues onto smaller and smaller intervals until eventually the user required minimum size is reached.

Parallelism can be introduced into the problem at three levels. Firstly since multiple independent subintervals are generated parallelism over interval processing can be utilised. This solution has been implemented by Barlow and Evans (1978) but is inefficient when the number of intervals is small. Secondly, parallelism can be exploited within the interval by sampling in parallel a number of points in the same interval. Barlow et al (1981a) have reported on the implementation of a mixture of these two methods on two different parallel computers. Finally parallelism can be introduced within the Sturm sequence evaluation. Thus, the Sturm sequence is defined by a linear recurrence relation and well known methods (see for example Kuck, 1978) are available to transform this apparently sequential relation into parallel form.

Which level of parallelism is the most efficient to exploit depends upon the balance between the demand for various parallel resources from the different versions of the algorithm, and the availability and cost of these resources on a given computer system. To analyse this balance the paper first specifies the problem and then analyses the parallel properties of the three potential solutions. Sections 2,3 and 4 then report on the implementation of these solutions on three different types of parallel computer: an array processor (the ICL Distributed Array Processor), a vector processor (the CRAY-1) and finally an asynchronous multiprocessor based on four Texas 990/10 minicomputers linked via shared memory.

1. Problem Specification and Analysis

The solutions are all based on the classic method (Barth et al, 1967) of counting the negative signs of Sturm sequences derived from the matrix. Thus given a symmetric tridiagonal matrix, of size n, with real eigenvalues lying between λ_{min} and λ_{max} then counting the number of negative signs of the Sturm sequences at a point λ_c gives the number of eigenvalues lying below λ_c. Since the interval can be sampled at an arbitrary number (m) of interior points the interval fragments itself into up to m+1 smaller intervals containing one or more eigenvalues. Application of the method to the new set of smaller intervals isolates the eigenvalues further and the process is repeated until the interval size is less than some user specified size.

Sequential solutions sample each interval at only one interior point (the bisection point). Parallelism can be introduced at the three levels mentioned in the introduction:

a) Processing some or all of the current set of known intervals in parallel: each individual interval is processed as in the sequential solution. This level of parallelism requires the same number of samples as the sequential solution but has the deficiency of having idle processors in the initial iterations of the algorithm when the number of intervals is small. The maximum degree of parallelism is limited to the number of distinct eigenvalues (N) to be located

0190-3918/82/0000/0034$00.75 © 1982 IEEE

and thus, assuming no overheads associated with controlling parallelism, the solution has a potential speedup (S) in the range $1<S<N$.

b) Evaluation of many sample points from one interval in parallel. This solution can exploit an arbitrary number of processors but is relatively inefficient. Consider an interval containing eigenvalues that on being sampled at m (equally spaced) interior points fragments itself into only one interval that contains all the eigenvalues. Thus, multisection has reduced the interval size by $1/(m+1)$ whereas sequential bisection can reduce the accuracy by this amount using only $\ell n_2(m+1)$ samples. It follows that the potential speedup of this solution lies between $\ell n_2(m+1)$ and m: the latter reflects the fragmentation of a single interval into m+1 intervals containing eigenvalues.

c) Parallelism within the evaluation of the Sturm sequence for a single point. The sequential nonlinear recurrence relation is

$$p_i = c_i - b_i/p_{i-1},\ p_0 = 1,\ p_1 = c_1,$$
$$\text{for } i=2,3,\ldots,n, \qquad (1.1)$$

where $c_i=a_i-x$, with x being the sample point and a_i the i^{th} diagonal element of the tridiagonal matrix: b_i is the square of the i^{th} diagonal element of the tridiagonal matrix. The number of negative signs of p_i yields the number of eigenvalues below x. Thus evaluation of this recurrence relation requires 3n operations.
This relationship can be transformed into the parallel recursion relation

$$q_0 = 1,\ q_1 = c_i\ ,$$
$$\begin{bmatrix} q_i \\ q_{i-1} \end{bmatrix} = \begin{bmatrix} \sum_{j=1}^{i} S_j \end{bmatrix} \begin{bmatrix} q_1 \\ q_0 \end{bmatrix}\ , \qquad (1.2)$$

where,

$$S_j = \begin{bmatrix} c_j & -b_j \\ 1 & 0 \end{bmatrix} \quad i=2,3,\ldots,N,$$

where

$$p_i = \frac{q_i}{q_{i-1}}$$

This relation involves $\ell n_2(n)$ sequential stages, each stage consisting of between n/2 and n parallel subprocesses each of complexity 12 operations: thus stage 1 forms all products $S_j.S_{j-1}$ for $j=2,\ldots,n$, stage 2 combines these results to give all products $(S_j.S_{j-1}).(S_{j-2}.S_{j-3})$, for $j=4,\ldots,n$ etc. (Lambiotti, 1975). Since each p_i can be reconstructed by a single operation the number of eigenvalues below x can be computed in $12\ell n_2 n+2$ parallel operations using n processors. Thus the speedup of the parallel version over its sequential counterpart is $\sim n/(4\ell n_2 n)$.

Already from this simple examination of the schemes it can be seen that for the separation of large numbers of eigenvalues (N) the first solution offers the best potential. For small numbers of eigenvalues this solution has little parallel potential and the other two solutions are better. Since the first two solutions both apply parallelism over the evaluation of Sturm sequences of different sample points it is a relatively simple task to combine these two solutions to yield a single parallel solution with a broader range of application than either of its two parts.

A further distinguishing feature between the solutions is the amount of synchronisation that the solutions require. Synchronisation is required in the third solution to ensure that all subprocesses of a stage are complete before the next stage starts, that is every 12 operations. In solutions one and two synchronisation is required at most every 3n operations; that is not more frequently than once for each Sturm sequence evaluation.

Finally, there is the question of data communication. Thus for the first two solutions sample points must be provided to the processors and the results in some way compared. For the third solution equation 1.2 shows that between each stage the s_i must be moved between the processors before the next stage can begin.

The effects of synchronisation and data communication will be more fully discussed in the following section.

2. Implementation on an Array Processor

It is assumed that the array processor consists of a single control processor with p slave processors all of which execute the same instruction on different data. Any required set of slaves can be set inactive (masked out) on any instruction. The control processor can broadcast the same data to all of the slaves or pick out an item of data from one of the slaves. Slaves are assumed to be linearly interconnected so that all slaves in parallel can move an item of data to either their left or right-hand neighbour. Synchronisation is automatic on these systems. The ICL Distributed Array Processor (Flanders et al, 1977) on which the solutions were implemented has 4096 slave processors (each of which can process one bit at a time).

Solution three, based on recursive doubling, can use at most n processors and thus for $n < p$ some processors in the array are idle. Parallelism here involves the evaluation of the q_i of equation 1.2. This is done by evaluating different partial

products of the S_j at different processors. Thus if a_j and b_j are stored at processor j; then all the S_j can be evaluated locally once the sample point has been broadcast. Each stage i of the $\ell n_2(n)$ stages involves shifting the previous partial products (of S) 2^{i-1} places along the array followed by the combination of the shifted results with the previous results. The first i-1 of the shifted results are filled in with the old results. Although the arithmetic operation count is reduced significantly from 3n of the sequential scheme to $12\ell n_2(n)$ for this parallel form, the number of shift operations at (n-1) is linear in the system size. While data communication paths other than nearest neighbour are generally available on array processors it is clear that the cost of moving results will tend to dominate the processing for large n.

While solution two can use any number of processors it is more efficient to combine it with solution one so that the maximum speedup potential is realised while at the same time utilising all the processors. While this may require some increase in data communication to allocate sample points to processors the results show that this overhead is small compared to the cost of evaluating the Sturm sequence.

Each iteration of the combined method consists of allocating a distinct sample point to each of the slave processors followed by the independent evaluation of the sample function at each of the processors.

Allocation of the new sample points starts by detecting the set of non-empty intervals arising from the previous iteration. This is done by nearest neighbour comparison of the sample point results for points interior to an interval. For points adjacent to interval boundaries this is done by comparing the sample point result with the boundary point result carried over from the previous iteration. The full treatment of boundaries is described by Barlow et al, 1981, and it is sufficient here to note that new intervals adjacent to boundaries can be treated in exactly the same manner as new internal intervals. At this stage intervals containing eigenvalues but lying outside a user defined range of interest can also be marked as empty. Using parallel add and shift operations the intervals are numbered, in monotonic increasing order, and their total (N') obtained. Using the new multisection factor m=INT(p/N') the intention is to move the data of new interval i (centre point and because of boundaries its result) to the m processors j=(i-1)m+1 to j=im. This is done by either sequentially broadcasting the interval data and masking out all but m processors each broadcast, or by shifting the data of all the intervals in parallel. The latter involves first shifting all data leftwards until no interval needs be shifted further and then repeating the process for right shifts. The efficiency of the parallel shift can be grasped by noting that if the number of intervals is a constant between iterations, then, at most m-1 right or left shifts are required. In the implementation dynamic choice between the two options is made.

Each slave processor independently evaluates the sample function for its sample point. This Sturm sequence evaluation is identical for all points, except for the possible incrementation of the eigenvalue count: thus the test for a negative sign sets a mask that is then used to mask out processors for the incrementation operation.

2.1 Results

It can be noted that each individual processing element of the ICL DAP has an arithmetic (logical) power about 100 times less (equal) to that of its host.

Table 1 indicates the power of the combined method in locating large numbers of eigenvalues: the theoretical results indicate a maximum expected speedup of $\sim N\ell n_2(1+(4096/N))$, offset by the limited power of the processing elements. Table 2 shows that for a matrix of size n=1024 the parallel recursive method outperforms the combined method only when searching for up to two eigenvalues. For smaller (larger but $n \lesssim 4096$) matrices the recursion method performs relatively worse (better). Theory predicts that for locating one eigenvalue of a matrix this size results in speedups of $\ell n_2(4096)=12$ and $1024/(4\ell n_2 1024)=25$ for respectively the combined and recursion methods: in practice the parallel versions were 7 and 3 times slower than the host computer due to the low power of the processing elements.

The cost of data communication for the recursion relation was 15% for the example above while for the combined multisection it was 17% for N=64 decreasing steadily to 1.5% for N=4096.

3. Implementation on a Vector Processor

Vector processors achieve their power by introducing pipelining into the various computational processes. For our purpose it is sufficient to note that the time taken to compute operations on vectors of length n is

$$T(n) = A + B(n-1), \text{ where } p=A/B \text{ and } p>>1.$$

In practice this simple formulae may only apply up to limiting value of n corresponding to some vector register size of the computer concerned. Scalars are assumed to take the same time to process as a unit length vector: that is T(1)=A. This is a simplification since on vector processes that require complex data alignment networks to be set in order to process vectors the scalar operation time will be significantly less than A. Implementations were on the CRAY 1S for which p ~ 30-50 (Peterson, 1979).

Consider now the implementation of solution three: recursive doubling. The ordinary sequential recursion relation of 3n operations

requires,

$$Ts(n) = 3nA \qquad (3.1)$$

since no vector processing is possible. The parallel recursive doubling solution requires $\ell n_2(n)$ stages of 12 operations with between n/2 and n processes in each stage. In fact approximately $9n\ell n_2(n)$ operations are required. The time to execute these instructions on a vector processor can be bounded below by making the assumption that all these operations can be put into a single vector operation: this would result in a time of

$$Tr(n) = 9n\ell n_2(n)B \qquad (3.2)$$

Comparison of these equations shows that recursive doubling cannot yield results faster than the sequential solution if $p<3\ell n_2(n)$. For the CRAY 1 this implies that parallel recursion cannot be faster than sequential recursion if $n>2^{10}$ assuming p=30.

This is an interesting limit on all parallel algorithms that increase the combined computational complexity by a factor $\ell n_2(n)$ in order to generate parallelism of order n.

Let us now consider a combined solution one and two. Imagine that we are searching for only one eigenvalue. Then it requires k iterations of bisection to reduce the interval size by 2^k. Since only one interval is available the vectors are of length 1 and the time taken is

$$Tb = k2n(A) \qquad (3.3)$$

If multisection is now introduced so that $m=2^j-1$ samples are taken in the single interval then k/j iterations are required and the time taken is

$$Tm = k/j\ 2n(A+mB) \qquad (3.4)$$

since the vectors are of length m. Before we proceed to minimise this equation with respect to j we can introduce the effect of multiple sub-intervals (N) into this last formulae on the assumption that sampling produces no more sub-intervals. Thus 3.4 becomes

$$Tm = k/j\ 2n(A+N(2^j-1)B) \qquad (3.5)$$

Minimising this with respect to j one obtains

$$A/B = N(2^j(j*\ell n_e(2)-1)+1) \qquad (3.6)$$

3.1 Results

Table 1 shows the CRAY 1 time to locate all the eigenvalues of some large matrix. The speedup comparison is with respect to the optimal sequential version (Barth, 1967) which since it evaluates only 1 sequential recursion relation at a time uses only the scalar functional units of the CRAY.

Table 3 illustrates the improvement that can result from using multisection when searching for small numbers of eigenvalues. The minimum times occur for a value of the multisection factor that is in rough agreement with equation 3.6. The times of the sequential algorithm are given under the multisection factor of zero. For the case of locating 1 eigenvalue using bisection (multisection factor=1) the parallel algorithm has no parallelism and carries out exactly the same operation as the sequential version. However the parallel version uses the vector functional units and it can therefore be seen that scalars can be processed ~2½ times as fast as vectors of length 1.

Table 2 compares parallel recursion with parallel multisection: the latter using multisection factors determined dynamically in the program. Discrepancies between the results of Tables 2 and 3 arise from using different terminating accuracies.

4. Implementation on an Asynchronous Multi-Processor

Asynchronous multiprocessor computer systems are composed of processors capable of independent operation. On these systems similar operations may take different amounts of time to execute. Thus termination of one or more operations cannot be guaranteed by a hardware clock as in array or vector processors. Signalling the termination of parallel processes is thus an overhead on such systems. Furthermore it involves communicating information between the processors which requires that the processors must share physical and/or logical resources. Limited access constraints to shared resources imply the speedup is bounded by saturation of shared resources availability (see for example Barlow, 1982 or Barlow et al, 1982). Finally, synchronisation requires that on sub-processes of equal complexity faster processors must wait on slower ones to finish.

The system to be considered consists of four Texas Instruments 990/10 minicomputers linked through shared memory (Barlow, et al, 1981). The cost of synchronising the termination of paths is equivalent to 40 integer operations at a minimum. The synchronisation resource itself can only be accessed by one processor at a time and since this resource has a cycle time of approximately 20 integer operations the speedup obtainable is limited by S=(equivalent integer operations between access)/20. In addition there is an overhead associated with access to the data communication system (the shared memory) of 100% compared to accesses to local data. The shared memory being a shared resource also limits the maximum obtainable speedup. A deficiency of the current system is that it has no floating point hardware and thus these operations take ~40 times longer to execute than integer operations. To supplement our analysis we include values (in brackets) that would result from a floating point to integer operation execution time ratio of 5.

For the solution based on parallelism within the Sturm sequence evaluation the cost of synchronisation is extremely high as it is required once per 12 operations from each processor. Thus, the overhead due to synchroni-

sation is 8% (67%) and the limit to speedup due to saturation of the synchronisation resource is 24 (3): and this ignores the extra operations carried out in the parallel form of the solution. For small numbers of processors and large matrices it is possible to reduce the synchronisation by grouping subprocesses within a stage together: so that each processor takes n/p subprocesses. However it is clear that a minimum of $\ell n2(n)$ more synchronisations are required than in solutions one and two. Furthermore from equations 1.1 and 1.2 it can be seen that the expected speedup from this solution can be at most

$$s = p/(c\ell n2(n)) \text{ where } 2 \leq c \leq 4$$

and thus for this system this solution has nothing to offer.

The results of a straightforward parallel implementation of a combined solution one and two are shown in Table 4. This implementation is almost identical with the array processor version of Section 2. Thus parallelism over intervals and possibly within intervals is exploited. The results show that a significant amount of processor time is wasted (8%) either by some processors completing before others or by an imbalance in the number of subprocesses to processors (a single multisection factor cannot always achieve an equal allocation of work to processors).

Since waiting arising from synchronisation can be a significant cost on asynchronous systems various authors (Kung 1976, Baudet 1977) have redesigned algorithms so that they require no synchronisation. The algorithms do however sometimes require coordination (mutual exclusion) to ensure the integrity of certain program data structures shared by the processors: this coordination imposes overheads on access and limits to speedup for exactly the same reasons as the shared synchronisation resource.

Following these ideas it is of interest to develop a form of solutions one and two that eliminates the synchronisation that forces fast processors to wait on slower ones. For solution one (bisection) this is simple since a processor sampling at one point of an interval can generate new intervals by carrying forward from the previous iteration the Sturm sequence results of the interval boundaries. The asynchronous algorithm can be formulated as:

a) A list of intervals giving their centre point, size, left and right boundary number of eigenvalues.

b) A process that collects the next interval from the list, evaluates the Sturm sequence at its centre point and then adds new smaller intervals to the list.

Coordination between the processors is then only required to ensure one processor at a time access to the list.

Multisection (solution two) can be incorporated into this solution by extending the structure of the list so that each interval becomes represented by a tree structure. This tree structure is headed by the bisection interval information. This level (0) and lower levels then point to nodes that represent multisection points: 1/2 at level 1, 1/4 and 3/3 at level 2 etc., each node consisting of a pointer to its parent, space for two pointers to children and finally space for the result of the Sturm sequence evaluation for the point it represents.

Pointers to nodes are only built when the point of that node has been taken to be sampled. Processors search the list/tree structure on a level by level basis, starting at level 0, so that the amount of multisection as opposed to bisection is always minimised. After completion of the evaluation the result is returned to the relevant node. If the node is at level 1 the tree splits to represent two new intervals, with each of the old level two nodes pointing to one of these intervals. Following a splitting operation the processor must try to recursively split the newly generated intervals since other processors have earlier completed sampling level two nodes from the old interval. Further details are given in Barlow et al (1981a).

Results for this asynchronous solution shown in Table 4 indicate some improvement over the synchronous solution. However the cost of tree processing is severe (6.8%) and since only one processor at a time can access this list there is a limit (p=n/16 see Barlow et al 1982) to the potential speedup.

For both synchronous and asynchronous solutions the rate of access to shared data is low because after having obtained the interval data the 2n operations of the Sturm sequence evaluation make no reference to shared data. Thus without floating point hardware the losses were too small to measure: they can be expected to be ~1% for matrix sizes of ~256 with floating point hardware.

Conclusion

The most striking feature of the results is the failure of the solution based on parallel recursion to yield any significant improvement over the sequential recursive solution. Our analysis shows that this failure is to be expected for large size systems.

This result has ramifications for all parallel solutions that for a system of size n convert the original sequential algorithm of complexity $c_1 n$ into a parallel algorithm that consists of $F(n)$ steps with each step containing n subprocesses of complexity c_2. For the resulting solution to run faster than its sequential counterpart then the number of processors that are utilised must satisfy $p > (c_2/c_1)F(n)$. This is a severe limitation for processes based on pipelining as there is a limit to the number of stages and thus the effective number of processors.

TABLE 1: ICL DAP and CRAY 1 Timings for Locating all the Eigenvalues Using Combined Multisection and Bisection

SIZE	ICL DAP		CRAY 1	
	Time (secs.)	Speedup*	Time (secs.)	Speedup***
64	0.24	4	0.028	3.8
256	1.15	12	0.27	5.5
1024	6.66	27	3.14	6.2
4096	65.15	>46(15)**	49.26	6.8

* *Speedup calculated with respect to ICL 2980 (the DAP host)*

** *ICL 2980 version ran out of time (CDC 7600 comparison in brackets)*

*** *Compared to CRAY sequential solution*

TABLE 2: ICL DAP and CRAY 1 Timings for Locating a Small Number of Eigenvalues of a Matrix of Size 1024

No. of Eigenvalues	ICL DAP		CRAY 1	
	Solution 1+2	Solution 3	Solution 1+2	Solution 3
1	2.85 secs.	1.1 secs.	0.034 secs.	0.145 secs.
4	2.85	2.29	0.0485	0.312
16	2.85	6.8	0.0929	1.128

TABLE 3: Effect of Varying the Multisection Factor on the CRAY 1 (Matrix Size 1024)

Factor(m)	Times (in seconds) to locate Eigenvalues						
Eigenvalue	1	2	4	8	16	32	64
0*	0.034	0.044	0.071	0.129	0.256	-	-
1	0.084	0.085	0.085	0.092	0.107	0.147	0.228
2	0.054	0.055	0.059	0.067	0.091	0.138	0.253
4	0.038	0.040	0.046	0.060	0.090	0.162	0.316
8	0.034	0.036	0.046	0.064	0.101	0.208	-
16	0.034	0.037	0.049	0.080	0.134	-	-

* *Sequential solution time.*

TABLE 4: Results for an Asynchronous Multiprocessor (N=16,n=256)

Method	Speedup with 2 processors*	3 processors*	4 processors*	Idle Time**	Synchronisation Overhead**(inc. lockout)	Tree Processing Overhead**(inc. lockout)
Synchronous	1.8	2.6	3.4	8%	0.5%	-
Asynchronous	2.0	2.9	3.7	-	-	6.8%

* *Compared to sequential bisection on one processor*

** *For the case of 4 processors*

This conclusion has been previously pointed out by Lambiotti (1975). It has led Sameh and Brent (1977) and Chen, Kuck and Sameh (1978) to develop alternative parallel recursive solutions that have less parallelism but are more effective with small numbers of processors. These solutions are currently being investigated.

Another interesting point that was discovered was that, firstly, inspite of the low potential gain on introducing parallel multisection within an interval and secondly the increase in processing time on the vector processor the method did yield improvements even when locating small numbers of eigenvalues. The failure of this method for very small numbers of eigenvalues on the ICL DAP reflects the extremely low processing power of the array elements.

Finally, although the asynchronous solutions involving both parallelism within and between intervals yielded only a slight improvement over its synchronous counter-part the former solution can yield significant gains when the speed of the processors differ significantly. Thus in the synchronous version all processors are slowed down to the speed of the slowest processor.

References

[1] R.H. Barlow, D.J. Evans, "A Parallel Organisation of the Bisection Algorithm", Computer Journal, 22, 1978, pp.267-269.

[2] R.H. Barlow, D.J. Evans, J. Shanehchi, "Parallel Multisection Applied to Locating Eigenvalues of Symmetric Tridiagonal Matrices", to appear in Computer Journal, 1981a.

[3] R.H. Barlow, D.J. Evans, I.A. Newman, M.C. Woodward, "The NEPTUNE Parallel Processing System", Internal Report, Loughborough University of Technology, 1981b.

[4] R.H. Barlow, D.J. Evans, I.A. Newman, J. Shanehchi, M.C. Woodward, "Performance Analysis of Parallel Algorithms on Asynchronous Parallel Computers", Internal Report, L.U.T. (1982).

[5] R.H. Barlow, "Performance Measures for Parallel Algorithms", in 'Parallel Processing Systems', ed. D.J. Evans, (Cambridge University Press), 1982.

[6] W. Barth, R.S. Martin, J.H. Wilkinson, "Calculation of the Eigenvalues of a Symmetric Tridiagonal Matrix by the Method of Bisection", Numer.Math. 9, 1967, pp. 386-393.

[7] S.C. Chen, D.J. Kuck and A.H. Sameh, "Practical Parallel Band Triangular System Solvers", ACM Transactions on Mathematical Software, Vol.4, No.3, 1978, pp.270-277.

[8] P.M. Flanders, D.J. Hunt, S.J. Reddaway, D. Parkinson, "Efficient High Speed Computing with the Distributed Array Processor", in 'High Speed Computer and the Algorithm Organisation', ed. D.J. Kuck, D.H. Lawrie, A.H. Sameh, 1977, Academic Press.

[9] D.J. Kuck, "The Structure of Computers and Computations", 1978, Wiley & Son.

[10] J.J. Lambiotte, Jr., "The Solution of Linear Systems of Equations on a Vector Computer", Ph.D. Thesis, School of Engineering and Applied Science, University of Virginnia, 1975.

[11] W.P. Peterson, "Basic Linear Algebra Subprograms for CFT Usage", CRAY Research Inc., Publication Number 2240208.

[12] A.H. Sameh, R.P. Brent, "Solving Triangular Systems on a Parallel Computer", SIAM J. Numer.Anal. Vol.14, No.6, pp.1101-1113.

A MESH COLORING METHOD FOR EFFICIENT MIMD PROCESSING IN FINITE ELEMENT PROBLEMS

Ph. Berger, P. Brouaye, J.C. Syre

Department of Computer Science
O.N.E.R.A. - C.E.R.T.
B.P. n° 4025
31055 - TOULOUSE (FRANCE)

Abstract -- Solving finite element problems on SIMD or MIMD systems raises implementation questions due essentially to non conflict free accessing to data structures as they are commonly handled in finite element programs. These difficulties may be overcome by redesigning algorithms and partitioning the mesh into non connected subsets. After a graph modelization of the problem, the decomposition is related to a graph coloring algorithm, yielding the elementary tasks and their corresponding data which are allowed to run concurrently in a multiple processor system. The study is implemented on a general hardware and software MIMD simulator supporting a high level language and performance evaluation tools.

Introduction

The emergence of multiple processor systems raises numerous questions in many fields of computer science : processor-memory organization, data allocation, task scheduling, programming languages. Other questions are of great concern when one wants to write real programs for those new systems. More specifically in numerical analysis, we are conducting a study of parallelization in Partial Differential Equations problems using finite element techniques. The environment consists in a high performance, MIMD system currently under specification, where algorithms and data organization have to be redesigned to achieve efficiency, since speed is based on concurrency and independence rather than on vectorizing or pipelining techniques [1][7]. Data access conflicts occurring during the algorithmic step of discretization are solved by partitioning the whole grid into non connected subdomains.

The efficiency of this partitioning method over conventional ones is evaluated by utilizing a general MIMD simulator currently under development in companion teams. The MIMD system can be configured by choosing appropriate parameters for the number of CPUs, local memory and secondary memory banks, the behavior of two communication networks linking CPUs and local memories on one hand, local memories and secondary memories on the other hand. A simulation language allows the high level expression of tasks, data organization and allocation, and the expression of control among tasks which will be interpreted by the simulator's supervisor. Almost every part of the hardware and software for the MIMD system is user-definable, thus many strategies can be rapidly set up and compared.

Conflict free data accesses in multiprocessor systems

Finite element algorithms often require a step of linearization. During this step, a usually large matrix is assembled. Its size equals the number of unknowns in the mesh and varies with the problem approximation [2], [3], [4]. Once fixed the grid geometry, the assembly step yields two closely interacting phases, for each element :

- compute an elementary matrix representing the local nodal contribution of an element in the mesh to the global matrix (figure 1),
- accumulate the matrix elements into the global matrix supporting the linear system coefficients (figure 2).

Let NE be the number of elements in the mesh, C(p) the elementary contribution matrix for element p, and A the global matrix. Obviously, C(p) and C(q), for any p and q $\in \{1,NE\}^2$ may be computed concurrently (phase one). Unfortunately, as seen in figure 2, they will eventually alter a same line of A during the second phase accumulating their terms into A.

The conventional step consists in computing one C(p), updating A and repeating it until the last element. An SIMD or MIMD system (MIMDS) could perform phase one in parallel, but updatings would have to be sequential, or strongly sequenced to avoid access conflicts to A. MIMD systems would achieve better performance than SIMD ones for phase one, since data are usually accessed through other arrays. Indirect addressing is known to be tedious for SIMDs, while independent processors (MIMDs) should accomodate it naturally.

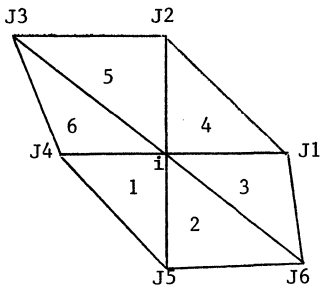
However, an MIMDS would be in trouble for phase two, since two or more updating tasks running on different processors will possibly perform load-add-store operations on identical elements of A. Thus a control must be defined in order to avoid conflicts. It will sequence the updating tasks such that at any time there should not be simultaneous operations on a given subset of A.

The partitioning technique

To solve this problem, we are searching a partition of elements into subsets S_i i=1,n, such that :

- S_i = {set of elements in the mesh}

0190-3918/82/0000/0041$00.75 © 1982 IEEE



$$C(3) = \begin{bmatrix} a_{ii}^{(3)} & a_{ij1}^{(3)} & a_{ij6}^{(3)} \\ a_{j1i}^{(3)} & a_{j1j1}^{(3)} & a_{j1j6}^{(3)} \\ a_{j6i}^{(3)} & a_{j6j1}^{(3)} & a_{j6j6}^{(3)} \end{bmatrix} \qquad C(p) = \begin{bmatrix} a_{kk}^{(p)} & a_{k\ell}^{(p)} & a_{kh}^{(p)} \\ a_{\ell k}^{(p)} & a_{\ell\ell}^{(p)} & a_{\ell h}^{(p)} \\ a_{hk}^{(p)} & a_{h\ell}^{(p)} & a_{hh}^{(p)} \end{bmatrix}$$

For triangle 3

For triangle p with nodes (k,ℓ,h)

FIGURE 1 : contribution matrices.
(For P1. Type triangulation with one unknown per point)

$$A \text{ (ith line)} = \left[0 \diagdown a_{ij4} \cdots a_{ij3} \cdots a_{ij5} \cdots a_{ii} \cdots a_{ij2} \cdots a_{ij6} \cdots a_{ij1} \diagdown 0 \right]$$

Then, for the triangles given above :

$$a_{ij5} = a_{ij5}^{(1)} + a_{ij5}^{(2)}$$

$$a_{ii} = a_{ii}^{(1)} + a_{ii}^{(2)} + a_{ii}^{(3)} + a_{ii}^{(4)} + a_{ii}^{(5)} + a_{ii}^{(6)}$$

(superscripts denote the triangle number)

FIGURE 2 : Accumulation of contribution matrices into the final linear system.
(P1 triangulation).

- $\bigcup_{i=1}^{n} S_i$ = domain of integration
- for any E, element in the mesh, there exists one $i \in [1,n]$, $E \in S_i$
- for any $i \in [1,n]$, any couple E_1, $E_2 \in S_i^2$, $E_1 \cap E_2 = \phi$.

For all elements belonging to S_i, the set of operations consisting of computing the contribution matrices (phase one) and assembling them into A (phase two) can be performed fully concurrently. By construction of the partition, no conflict occurs during phase two which is now mixed with phase one. Passing from S_i to S_{i+1} will still be synchronized, with the capability of anticipating S_{i+1} phase one during final stages of S_i processing.

The partition is determined by identifying each element in the mesh with a node in a graph. Two nodes are adjacent if they correspond to a couple of neighbour elements, i.e. E_1 and E_2 such that $E_1 \cap E_2 \neq \phi$. The construction of subsets S_i of elements in the mesh is equivalent to the problem of coloring the corresponding non planetary graph [5], [6].

One way to achieve optimal coloring is to find the chromatic number γ of graph G. Practically, this algorithm would be too much time expensive if one wants to include it into the normal processing of the global matrix computation. Instead, we used another one, derived from Powell and Welsh's theorem [8]. Let us denote d_i the degree (or valency) of a point v_i in G, i.e. the number of lines incident with v_i, since G is not oriented. Then the algorithm produces a number of colors $N \leq \text{Max}\ (d_i)+1$.

The coloring algorithm can be stated as follows :

- Step 1 : nodes in the graph are re-ordered according to their decreasing valency. Let $\{p_i\}$ i=1,..,n the newly ordered list of elements. Thus p_1 has the largest valency. Let j = 1.
- Step 2 : take element $p_{j_1} = p_j$, and find all elements p_{jk} in the ordered list such that p_{jk} is not adjacent to any $p_{j\ell}$, $\ell = 1,\ldots,k-1$. This forms partition S_j.
- Step 3 : suppress all elements of S_j from the list. If the list is empty, halt the process, otherwise let j be the new first element and iterate step 2.

Fig. 3 shows two applications of the algorithm.

This basic coloring method is applicable to any mesh type and, so far, has been experimented on three different regular domain triangulation (P1-type). The results are shown on Table I for different grid sizes.

Additionally, the basic algorithm can be improved in two ways : firstly, the difference between the optimal chromatic number and the number of colors can be attenuated, and secondly, the number of elements for each color can be balanced.

We chose the last solution, thus from a parti-

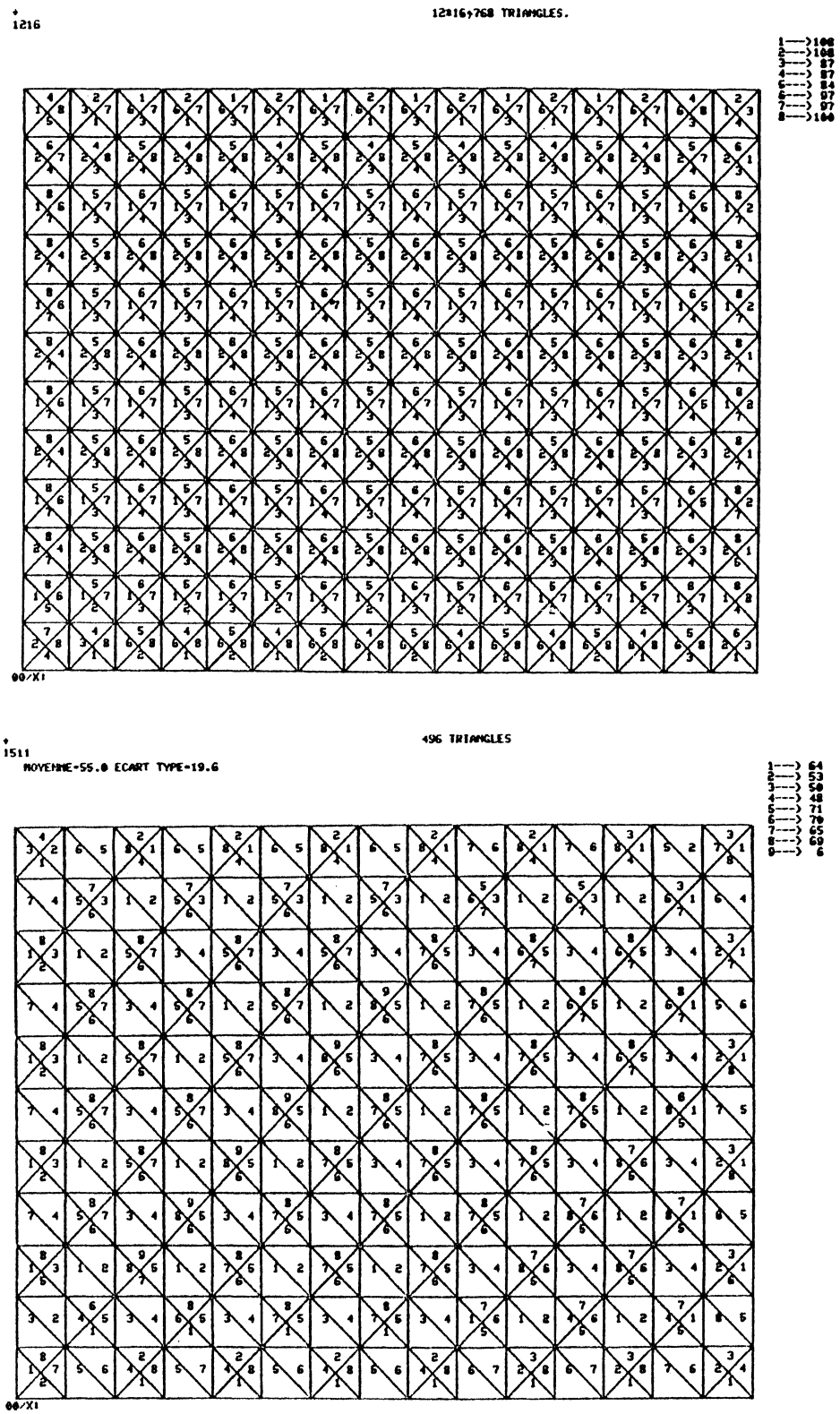
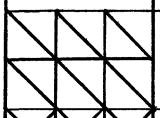
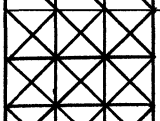
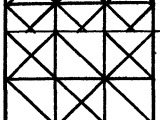


FIGURE 3 : Coloring two meshes (type 2, type 3)

MESH TYPE	# Elements	γ	# colors	σ
	50	6	7	2.1
	200	6	8	8.17
	800	6	8	13.06
	100	8	8	2.92
	160	8	8	3.35
	400	8	8	5.61
	768	8	8	8.75
	106	8	9	5.7
	232	8	9	10.3
	496	8	9	19.6
	756	8	9	30.96

σ = standard deviation of number of elements per color.

TABLE 1

tion, C1, set up by the basic algorithm, we now build an improved one, C2, by performing the following changes :

Step 1. The subsets of C1 are ordered according to their decreasing number of elements. Let S_i i=1,...,n be this list.

Step 2. Some elements in S_1 are shifted into S_n when possible. The shifting process halts for S_1 and S_n when :

. S_1 has now NE/n elements (balanced number). Then step 2 is done with S_2 and S_n.

. S_n has now NE/n elements. Step 2 is done with S_1 and S_{n-1}.

. No element in S_1 can be transferred into S_n, and S_n has less than NE/n elements. Step 2 is done with S_2 and S_n.

We thus obtain a pseudo-uniform number of elements in each color, close to NE/n. An illustration of this optimization gives, for the third mesh type in Table 1 whose deviation is rather large, the following improvements :

- for 106 elements = 0.7 (instead of 5.7)
- for 232 elements = 0.7 (instead of 10.3)
- for 496 elements = 4.4 (instead of 19.6)
- for 756 elements = 8.4 (instead of 30.96).

Simulation and results

Our work, oriented to parallel numerical methods, is part of a joint study of multiprocessor systems at ONERA-CERT. Other people in our group developped a simulator of general MIMD architectures. Before giving a comparison of performance between several assembling methods, it may be helpful to define the overall characteristics of the simulator.

The class of MIMD systems simulated corresponds to figure 4 :

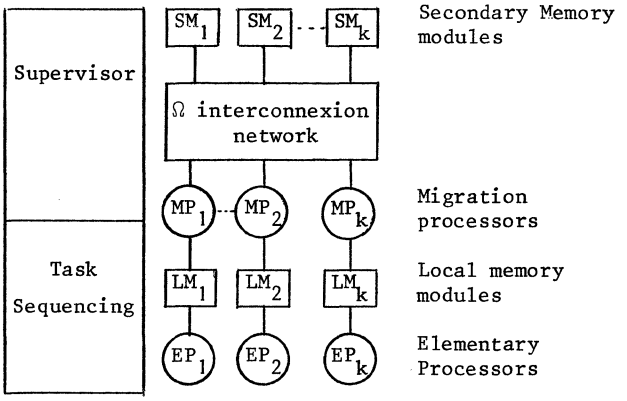


FIGURE 4 : MIMD architectures

Data and code are initially located in Secondary Memory modules. A Migration Processor can access an SM module via an Omega-type asynchronous network, to build up data or code blocks and transfer them between SMs and LMs (Local Memory modules). Each Local Memory is attached to one MP and one Elementary Processor (EP).

A program is made of a collection of tasks specified by their actual input/output data on one side, and their algorithmic part on the other side. Migration Processors execute the in/out part of a task, while Elementary Processors execute the algorithmic part and can be considered as conventional pipelined machines.

Task sequencing is expressed in a parallel language describing concurrency and synchronization by a set of independent, non sequential formulas. The supervisor interprets formulas, decides which ones are firable and executes actions corresponding to activable tasks. It sends control signals to MPs for data management and to EPs for code execution. Although the simulator is fully parametrized, the results given further are obtained from the following specifications :

- SMs : Random access memories, cycle time 300 ns,
- Omega Network : Routing time 100 ns, percentage of conflicts 30/100,
- MPs : Machine cycle time 300 ns,
- LMs : 128 K 64 bit word RAMs, cycle time 100 ns,
- PEs : cycle time 200 ns (5 Mflops),
- k will range from 1 to 16, giving a peak rate of 80 Mflops.

We are now considering an application, taken from aerodynamics or structural analysis problems. A two-dimensional mesh is composed of 25600 triangular elements (and 19200 points). A single unknown in each point leads to four non zero coefficients in one line in the upper semi band of the final matrix (assuming it symmetric). This matrix is implemented as a sparse structure, where only non zero terms are stored, with a privileged access to lines. The elementary contribution matrices have six relevant coefficients. The geometry, i.e. the coordinates of nodes, is duplicated and included into the connectivity matrix which gives the correlation

between the double numbering of elements and the nodes in the mesh. The execution time for one contribution is evaluated to 25 microseconds. We can now compare several assembling methods.

First method :

use sequential algorithm

One processor is used. Its LM is loaded with 800 lines of matrix A and local data for 200 contribution matrices. The EP iteratively computes one contribution and assembles it into A. The 800 lines of A are re-written into SM and another bulk of data is loaded again. There are 128 such iterations, leading to the results in Table 2.

# EPs	1
Exec. time (in seconds)	5.2
Percentage MP usage	20
Percentage EP usage	80
M Flops	4

TABLE 2 :

Simulation results (Sequential algorithm)

Second method :

Parallelize computation of contributions

The set of all contribution matrices is now a data structure by itself, and lies in SM. We can split the first step of computation into parallel tasks performing a subset of elementary matrices. The second step of assembling them into A is still sequential. Table 3 summarizes simulation results :

# EPs	1	2	4	8	16
Exec time (s)	5,2	3,2	2,2	1,7	1,5
Percentage MP usage	21	17	13	8	5
Percentage EP usage	79	65	47	30	18
M Flops	4	6,4	9,3	12	13,9

TABLE 3 : Simulation Results (parallelize contributions)

This method shows a little amount of parallelism in the first part of the program, however performance is not considerably improved when the number of processors is increased.

Third method :

Adding buffers for concurrency

We keep the usual numbering of elements, and still distinguish the computation of contributions and their assembly into A. The first step can be considered as a producer process, while the second will consume contribution matrices. The mesh is now divided into 2ℓ subsets $\{L_i\}$ $i=1,\ldots,2\ell$. We can exploit the following concurrency : while elementary matrices in subset L_{2j} $j=1,\ldots,\ell$ (respectively L_{2j-1}) are computed and stored into a buffer BUF1 (respectively BUF2), those already computed in buffer BUF2 (respectively BUF1) can be assembled into A, corresponding to subset L_{2j-1} (respectively L_{2j-2}). Thus two levels of parallelism appear here :

- between computation and assembling,
- within computations.

This version of the algorithm is a representative trade-off of parallelization without dramatic algorithmic changes from the initial method. Simulation results are given below in Table 4.

As can be seen, introducing buffers does not improve performance quite significantly. This is due to several factors :

- the management of buffers, which is explicitely expressed by the programmer, is time expensive and increases the overhead in the supervisor.

- the consuming process, assembling the elementary matrices, is slow and is in fact dominant in the total execution time. Adding more buffers is therefore needless.

# EPs (k)	1	2	4	8	16
Exec time (seconds)	5.3	3	2	1.6	1.3
Percentage MP usage	21	19	15	9	6
Percentage EP usage	79	70	54	34	20
M Flops	3.9	7	10.7	13.4	16

TABLE 4 : Adding buffers for concurrency

Fourth Method :

Use of coloring algorithm

The mesh is now divided into eight colors with 3200 elements in each subset. The coloring phase is a pre-processing step which prepares the computations of elementary matrices and their assembling. Note that this step is not part of the normal processing, hence it is not included into the simulation times. The overhead induced by the preprocessing is expected to be small, and may be minimized by parallelizing the coloring algorithm itself.

The producer/consumer mechanism is maintained, however we now exploit an additional level of parallelism, since the assembling phase is made fully parallel for all elementary matrices belonging to the same partition. Table 5 gives the simulation results.

# EPs (k)	1	2	4	8	16
Total time (sec.)	5.2	2.5	1.2	0.65	0.35
Percentage MP usage	29	29	30	29	26
Percentage EP usage	75	77	81	76	71
M Flops	3.8	7.8	16.2	30.2	56.7

TABLE 5 : Use of coloring algorithm

Finally, the following diagram shows the performance of the different methods. Needless to say, the coloring one exhibits a notable gain over the others.

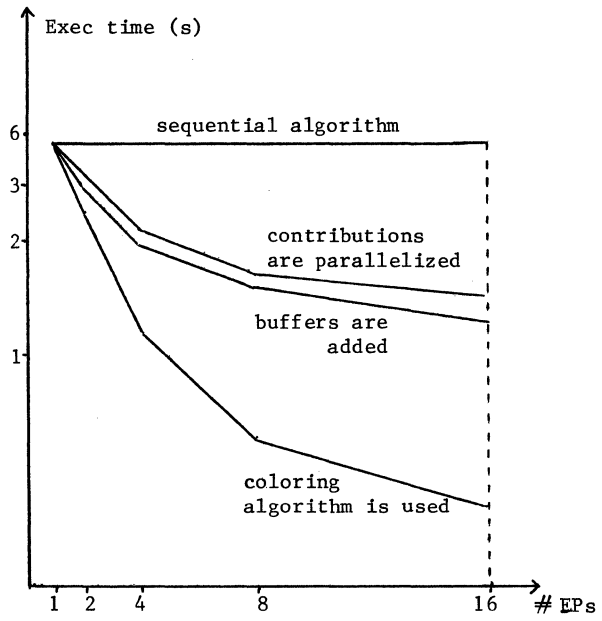


FIGURE 5 : Relative performance of assembling algorithms

Conclusion

As far as parallelization of finite element problems is concerned, various possibilities can be considered on multiprocessor systems. It must be pointed out that applying a given method to a given physiçal problem introduces peculiarities which can improve the initial algorithm resolving the partial differential equations. However, almost all methods, in aerodynamics as well as in structural analysis, resort to some common features useful to be dealt with :

- the discretization of the integration domain is complex and irregular enough to lead the designer to describe his geometry with a connectivity matrix,
- the initial step in those methods consists in computing some data local to elements, and projecting them onto a global data structure representing the state of physical system,
- when this assembling step leads to a matrix-type data structure, the next thing to do is a matrix inversion. Problems arise for this job, since that matrix is very large, very sparse with non zero terms more or less concentrated around main diagonal, giving a sparse or profile implementation.

The simulation of the first phase let us hope that high performance could be achieved on a multiprocessor system. As for the second one, if one uses the current widely used methods for direct inversion (like Gauss, Choleski), simulations did not yet reach sufficiently interesting rates to show that these methods are acceptable with minor changes. Thus new directions have to be discovered. The development of our simulator will allow us to interpret more intricated synchronization graphs of tasks, corresponding to new methods well adapted to MIMD machines.

References

[1] Ph. Berger, D. Comte, A. Maldonado, J.C. Syre, Etude et conception d'un système informatique spécialisé, DERI, ONERA-CERT, technical report n° 1/3147 (january, 1981)

[2] T.J. Chung, Finite element analysis in fluid dynamics, Mc Graw Hill (1978)

[3] A.J. Davies, The finite element method : a first approach, Clarendon Press, Oxford (1980)

[4] E. Hinton, D.R.J. Owen, Finite element programming, Academic Press, London (1977)

[5] B. Roy, Algèbre moderne et théorie des graphes, Ed. Dunod (1969)

[6] F. Harary, Graph Theory, Ed. Addison Wesley, (1969)

[7] A. Maldonado, Etude et évaluation d'architectures multiprocesseurs de type MIMD, ENSAE, Ph. D. Thesis (december, 1981)

[8] D.J.A. Welsh, M.B. Powell, "An upper bound for the chromatic number of a graph and its application to timetabling problems", Computer Journal 10 (1967), pp. 85-87

[9] Ph. Berger, Algorithmes et méthodes numériques parallèles pour multiprocesseurs MIMD, Department of Computer Science, ONERA-CERT, Technical report number 4/3168, (December, 1981)

This work was supported by DRET Grant number 82/1056/DRET/DS/CR.

AN EFFICIENT PARALLEL BLOCK CONJUGATE GRADIENT METHOD FOR LINEAR EQUATIONS

J. S. Kowalik
Department of Computer Science
Washington State University
Pullman, Washington 99164-1222

and

S. P. Kumar
Department of Mathematics and Computer Science
University of Miami
Coral Gables, Florida 33124

Abstract -- The block conjugate gradient method for linear equations is implemented to run on an MIMD parallel computer. The speedup of the parallel version of the method is approximately equal to the number of processors used, thus the method is well suited to run on a multiprocessor computer. Experiments have been performed on the Heterogeneous Element Processor manufactured by Denelcor, Inc. to validate the analysis and the code.

1. Introduction

"Plato taught that we do not learn new things; we merely remember things we have forgotten. For parallel processing, Plato's point is well taken" [7]. Indeed, the ideas of parallel computation have been around for a long time, but only recently have we begun to design efficient parallel algorithms, write executable codes and experiment with real parallel machines.

The subject of this paper is a block conjugate gradient (BCG) method for solving linear equations on an MIMD (multiple-instruction multiple-data) parallel computer. Originally, the method was developed for sequential computing by Jennings and Malik [2], who tested it and found that it was more efficient than the standard conjugate gradient algorithm. The reader interested in the method's numerical performance and its comparison with other iterative methods should consult Jennings and Malik. The block conjugate gradient algorithm attracted our attention because of its structure which naturally lends itself to parallel implementation on MIMD processors. To test numerically the parallel block conjugate gradient (PBCG) method we have used the HEP (Heterogeneous Element Processor) computer manufactured by Denelcor, Inc. [5]. This computer represents a departure from traditional computer architecture in that it supports multiple instruction streams (subroutines) executing cooperatively and in parallel to solve a single problem. An important feature of the computer is that these concurrent streams of instructions need not be identical. Moreover, there are means to synchronize the solution process, i.e., enforce temporal precedence constraints which are imposed by the nature of implemented algorithms. An extended Fortran language allows us to create concurrent subroutines and synchronize execution of the method. From the user viewpoint the HEP processor we used can be regarded as a collection of $1 \le p \le 9$ independent processors connected to a common main memory.

Let us now consider a set of linear equations,

$$A\underset{\sim}{x} = \underset{\sim}{b} \qquad (1)$$

where A is an nxn positive definite matrix, and $\underset{\sim}{x}$ and $\underset{\sim}{b}$ are vectors of the variables and right-hand sides, respectively. The standard conjugate gradient algorithm for solving (1) is as follows:

Initial step.

Set k = 0

$$\underset{\sim}{p}^0 = \underset{\sim}{r}^0 = \underset{\sim}{b} - A\underset{\sim}{x}^0$$

where $\underset{\sim}{x}^0$ is an initial approximation to the solution, usually taken to be null.

Iterative steps.

(i) $\underset{\sim}{u}^k = A\underset{\sim}{p}^k$

(ii) $\alpha^k = \dfrac{(\underset{\sim}{r}^k)^T\underset{\sim}{r}^k}{(\underset{\sim}{p}^k)^T\underset{\sim}{u}^k}$

(iii) $\underset{\sim}{x}^{k+1} = \underset{\sim}{x}^k + \alpha^k\underset{\sim}{p}^k$

(iv) $\underset{\sim}{r}^{k+1} = \underset{\sim}{r}^k - \alpha^k\underset{\sim}{u}^k$

(v) Test convergence, and stop or continue.

(vi) $\beta^k = \dfrac{(\underset{\sim}{r}^{k+1})^T\underset{\sim}{r}^{k+1}}{(\underset{\sim}{r}^k)^T\underset{\sim}{r}^k}$

(vii) $\underset{\sim}{p}^{k+1} = \underset{\sim}{r}^{k+1} + \beta^k\underset{\sim}{p}^k$

(viii) Set k:=k+1 and return to (i).

A suitable convergence criterion is $\|\underset{\sim}{r}^k\|/\|\underset{\sim}{b}\| \le \varepsilon$ where ε is a small number.

To develop a block conjugate gradient version of this algorithm we assume that the variables $\underset{\sim}{x}$ are divided into subvectors $\underset{\sim}{x}_i$, $\underset{\sim}{x}^T = (\underset{\sim}{x}_1^T, \underset{\sim}{x}_2^T, \ldots, \underset{\sim}{x}_m^T)$ and equation (1) is subdivided accordingly,

0190-3918/82/0000/0047$00.75 © 1982 IEEE

$$\begin{bmatrix} A_{11} & A_{22} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdots & \cdot \\ A_{m1} & A_{m2} & \cdots & A_{mm} \end{bmatrix} \begin{bmatrix} \underset{\sim}{x}_1 \\ \underset{\sim}{x}_2 \\ \cdot \\ \cdot \\ \underset{\sim}{x}_m \end{bmatrix} = \begin{bmatrix} \underset{\sim}{b}_1 \\ \underset{\sim}{b}_2 \\ \cdot \\ \cdot \\ \underset{\sim}{b}_m \end{bmatrix} . \qquad (2)$$

The diagonal submatrices are positive definite and their Choleski factors are denoted by L_{ii}.

Let L denote a lower triangular block diagonal matrix such that

$$L = \begin{bmatrix} L_{11} & & & O \\ & L_{22} & & \\ & & \ddots & \\ O & & & L_{mm} \end{bmatrix} .$$

Introducing a new set of variables

$$\underset{\sim}{z} = L^T \underset{\sim}{x} \qquad (3)$$

we obtain from (1)

$$AL^{-T}\underset{\sim}{z} = \underset{\sim}{b}$$

and multiplying both sides of this equation by L^{-1} we get

$$L^{-1}AL^{-T}\underset{\sim}{z} = L^{-1}\underset{\sim}{b}$$

or

$$B\underset{\sim}{z} = \underset{\sim}{d} \qquad (4)$$

where

$$B = L^{-1}AL^{-T} \qquad (5)$$

and

$$\underset{\sim}{d} = L^{-1}\underset{\sim}{b}.$$

The block conjugate gradient method is the standard conjugate gradient method applied to equation (4).

Since

$$B = L^{-1}AL^{-T} = \begin{bmatrix} I & C_{12} & \cdots & C_{1m} \\ C_{21} & I & \cdots & C_{2m} \\ \cdot & \cdot & \cdots & \cdot \\ C_{m1} & C_{m2} & \cdots & I \end{bmatrix}$$

where

$$C_{ij} = L_{ii}^{-1}A_{ij}L_{jj}^{-T},$$

the expression $\underset{\sim}{y}^k = A\underset{\sim}{p}^k$ yields subvector expressions of the form

$$\underset{\sim}{y}_i^k = \underset{\sim}{p}_i^k + L_{ii}^{-1}\Big(\sum_{j\neq i} A_{ij}L_{jj}^{-T}\underset{\sim}{p}_j^k\Big).$$

The vectors $\underset{\sim}{y}_i^k$ can be calculated in three steps:

(a) $L_{jj}^T\underset{\sim}{w}_j^k = \underset{\sim}{p}_j^k$

(b) $L_{ii}\underset{\sim}{v}_i^k = \sum_{j\neq i} A_{ij}\underset{\sim}{w}_j^k$

(c) $\underset{\sim}{y}_i^k = \underset{\sim}{p}_i^k + \underset{\sim}{v}_i^k.$

These three steps replace the calculation $\underset{\sim}{y}^k = A\underset{\sim}{p}^k$ in the standard conjugate gradient method. Also, step (iii) is replaced by

$$\underset{\sim}{z}^{k+1} = \underset{\sim}{z}^k + \alpha^k\underset{\sim}{p}^k.$$

The original variables $\underset{\sim}{x}$ are calculated using transformation equations (3).

Now we can state the block conjugate gradient method in terms of the vectors $\underset{\sim}{z}$, $\underset{\sim}{p}$, $\underset{\sim}{r}$, $\underset{\sim}{d}$, $\underset{\sim}{w}$, $\underset{\sim}{v}$ and $\underset{\sim}{y}$ which are subdivided in the same way as $\underset{\sim}{x}$.

The block conjugate gradient method.

Initial step.

Calculate L_{ii} such that

1. $A_{ii} = L_{ii}L_{ii}^T$, $\quad i = 1,2,\ldots,m$

   and solve for $\underset{\sim}{d}_i$.

2. $L_{ii}\underset{\sim}{d}_i = \underset{\sim}{b}_i$, $\quad i = 1,2,\ldots,m.$

   Set

   $k = 0$

   $\underset{\sim}{z}^0 = 0$

   $\underset{\sim}{p}^0 = \underset{\sim}{r}^0 = \underset{\sim}{d}.$

Iterative steps.

Perform the following sequence of calculations

3. $L_{jj}^T\underset{\sim}{w}_j^k = \underset{\sim}{p}_j^k$, $\quad i = 1,2,\ldots,m$

4. $L_{ii}\underset{\sim}{v}_i^k = \sum_{j\neq i} A_{ij}\underset{\sim}{w}_j^k$, $\quad i = 1,2,\ldots,m$

5. $\underset{\sim}{y}_i^k = \underset{\sim}{p}_i^k + \underset{\sim}{v}_i^k$, $\quad i = 1,2,\ldots,m$

6. $\alpha^k = \dfrac{(\underset{\sim}{r}^k)^T\underset{\sim}{r}^k}{(\underset{\sim}{p}^k)^T\underset{\sim}{y}^k}$,

7. $\underset{\sim}{z}_i^{k+1} = \underset{\sim}{z}_i^k + \alpha^k\underset{\sim}{p}_i^k$, $\quad i = 1,2,\ldots,m$

8. $\underset{\sim}{r}_i^{k+1} = \underset{\sim}{r}_i^k - \alpha^k\underset{\sim}{y}_i^k$, $\quad i = 1,2,\ldots,m$

9. $\beta^k = \dfrac{(\underset{\sim}{r}^{k+1})^T\underset{\sim}{r}^{k+1}}{(\underset{\sim}{r}^k)^T\underset{\sim}{r}^k}$.

10. If convergence is achieved then go to 12, otherwise continue. Our convergence test is $\dfrac{\|\underset{\sim}{r}^{k+1}\|}{\|\underset{\sim}{d}\|} \leq \varepsilon$, where ε is a suitable accuracy.

11. $\underset{\sim}{p}_i^{k+1} = \underset{\sim}{r}_i^{k+1} + \beta^k \underset{\sim}{p}_i^k \qquad i = 1,2,\ldots,m.$

Set k:=k+1 and go to 3.

Finishing step.

Solve for $\underset{\sim}{x}_i$

12. $L_{ii}^T \underset{\sim}{x}_i = \underset{\sim}{z}_i, \qquad i = 1,2,\ldots,m$

and stop.

The BCG method exhibits a remarkable degree of parallelism. Note that steps 1, 2, 3, 4, 5, 7, 8, 11 and 12 decompose naturally and can be computed concurrently by m processors. Steps 6, 9 and 10 can be implemented in a parallel fashion but they constitute a very small portion of the entire computational effort and have been implemented in our program on one processor.

2. The Parallel Algorithm.

For the purpose of our further analysis we make the following simplifying assumptions:

a. n/m is integer and each block in the partition of A is of the same size n/m.

b. every multiplication or additive arithmetic operation constitutes one computational step and all steps are equal in time length.

c. all processors are identical.

d. the matrix A is fully dense.

e. the number of partitions m, and the number of used processors p are equal, p = m.

The first assumption is simplifying but not essential. If n/m is not an integer then we assume that some submatrices A_{ii} are of the size $\lfloor n/m \rfloor$ and the remaining diagonal submatrices are of the size $\lfloor n/m \rfloor + 1$. The second and third assumptions are correct for the HEP processor. Sparsity is not considered in this paper. However, the presented PBCG method can be used to solve equations with sparse matrices and would run efficiently on multiprocessor computers for some classes of structured matrices, e.g., banded matrices.

The last assumption is not restrictive since the number of processors p is usually given a priori, and we can always use the number of partitions m = p.

To design a parallel version of BCG, we break the BCG algorithm into a set of computational tasks, denoted by T. A task is a collection of computational activities (operations) and our concurrently running subroutines will consist of sequences of tasks. The tasks of the BCG method are shown in Figures 1 and 2 which present this method as a pseudocode. Table 1 gives the operation count for each task.

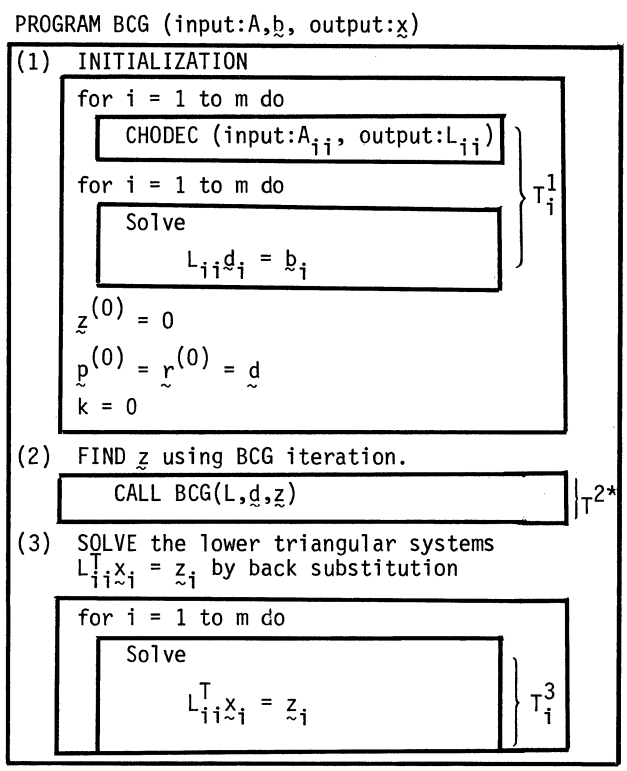
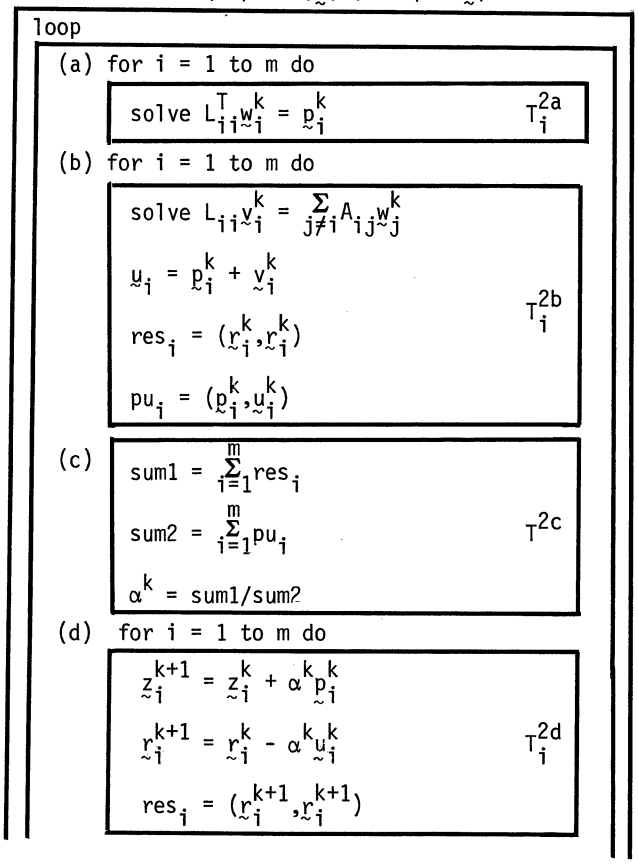


FIGURE 1.



(e) $sum2 = \sum_{i=1}^{m} res_i$
$\beta^k = sum2/sum1$ — T^{2e}

Convergence test. Exit or continue.

(f) for i = 1 to m do
$p_i^{k+1} = r_i^{k+1} + \beta^k p_i^k$ — T_i^{2f}

(g) k:=k+1, go to loop.

FIGURE 2.

TASK	OPERATION COUNT
T_i^1	$n^3/3m^3 + 3/2n^2/m^2 + O(n/m)$
T_i^{2a}	n^2/m^2
T_i^{2b}	$2n^2/m - n^2/m^2 + 4n/m-2$
T^{2c}	$2m - 1$
T_i^{2d}	$6n/m - 1$
T^{2e}	m
T_i^{2f}	$2n/m$
T_i^3	n^2/m^2

TABLE 1.

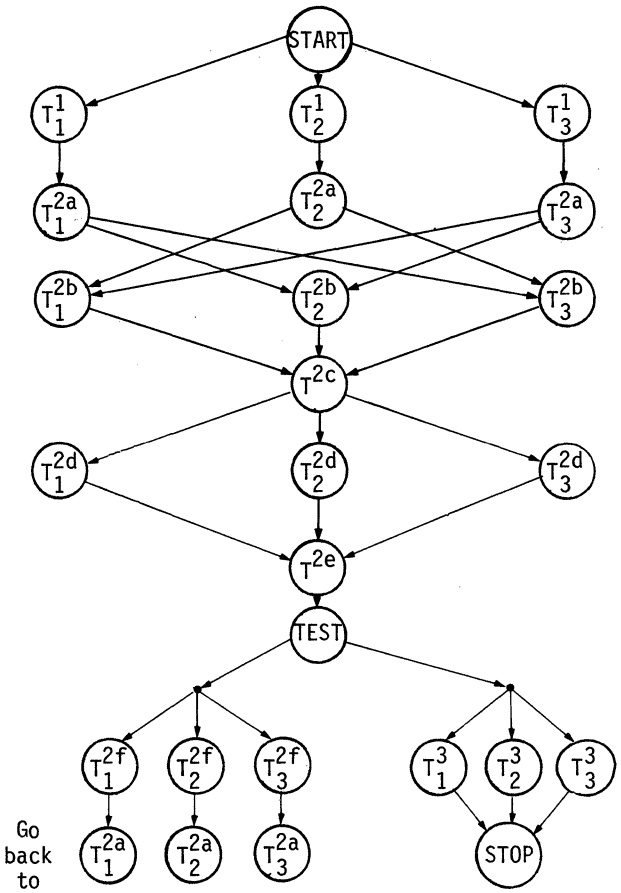


FIGURE 3. Task execution precedence graph G for m = 3.

Furthermore, we need to identify the time precedence constraints relating execution of the tasks. With each new task T there are associated two, possibly overlapping, ordered sets of memory cells, the domain D_T and range R_T. When the task T is initiated it reads the values stored in its domain and writes values into its range cells. We say that two tasks, T and $\hat{T}$ are noninterfering if either:

(i) T is a predecessor or successor of $\hat{T}$, or

(ii) $R_T \cap R_{\hat{T}} = R_T \cap D_{\hat{T}} = D_T \cap R_{\hat{T}} = \emptyset$ (empty).

The pair of set of computational tasks and the partial order representing time precedence constraints is called a task system and can be conveniently represented by a directed, acyclic graph, without redundant (transitive) arcs. The task system of mutually noninterfering tasks of the PBCG method, for m = 3, is shown in Figure 3 as the graph G. Note that if we execute various tasks T of G in parallel but follow the precedence constraints (execution of each task T commences only after all immediate predecessor tasks of T are completed) then the intermediate and final results of the computation will be exactly the same as the results of the sequential program.

Since we assumed that the number of processors and partitions are the same we can easily schedule the task execution. Figure 4 shows a schedule for m = 3. The shaded areas indicate idle periods for processors.

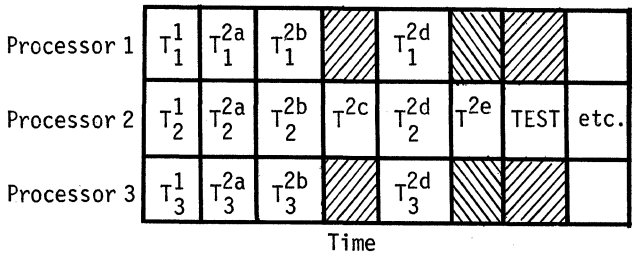


FIGURE 4. Task schedule for m = 3.

Giving weights to the nodes of the graph G according to Table 1 we obtain a weighted graph which has m maximum length paths from START to STOP. One of them is, for instance, START, T_1^1, (ITR-1)-times the path (T_1^{2a}, T_1^{2b}, T^{2c}, T_1^{2d}, T^{2e}, TEST, T_1^{2f}), T_1^{2a}, T_1^{2b},

T^{2c}, T_1^{2d}, T^{2e}, TEST, T_1^3, STOP. The weights of the nodes START and STOP are zeros. The weight of the TEST task depends on the selected convergence criterion and is not included in our operation count.

The sum of weights along this path, which is the maximum path length in the weighted graph, is:

$$t_m = n^3/3m^3 + \frac{5}{2}n^2/m^2 + ITR(2n^2/m + 12n/m) - 2n/m + ITR(3m-4)$$

The toal number of steps required by the BCG method is:

$$t_1 = m(n^3/3m^3 + \frac{5}{2}n^2/m^2 + ITR(2n^2/m + 12n/m) - 2n/m) - ITR$$

where ITR is the number of used iterations. t_1 is the length of execution time for BCG on a uniprocessor, measured in steps. Thus the speedup of the parallel algorithm for the chosen schedule with m processors is:

$$S_m = \frac{t_1}{t_m} = \frac{m(n^3/3m^3 + \frac{5}{2}n^2/m^2 + ITR(2n^2/m + 12n/m) - 2n/m - ITR}{n^3/3m^3 + \frac{5}{2}n^2/m^2 + ITR(2n^2/m + 12n/m) - 2n/m + ITR(3m-4)}.$$

Assuming that: (i) we solve sufficiently large systems, (ii) $n > m$, and (iii) $ITR << n$, the value of S_m is very close to m, which is the maximal speedup achievable under ideal conditions. In reality, there is some loss of speedup due to the overhead in the parallel computing process. We have to create parallel subroutines, and synchronize their progress. Additional time may be needed for data transfer and potential memory contention. Also we have taken into account the arithmetic work but ignored other instructions, such as do-loop controls.

Kumar [3] estimated that the time required to create and synchronize parallel subprograms in the PBCG method is:

$$t_m' = 2m(7\ ITR + 2) - 10\ ITR.$$

Thus the total execution time for the PBCG method is at least

$$t_m'' = t_m' + t_m$$

and the corresponding speedup is at most

$$S_m = \frac{t_1}{t_m''} \qquad (6)$$

3. Numerical Results.

To test the PBCG method the following two types of problems have been solved:

a. randomly generated positive definite systems. The matrices A and the vectors $\underset{\sim}{x}$ have been generated randomly and then A has been made diagonally dominant. The right-hand side vector $\underset{\sim}{b}$ has been calculated from $\underset{\sim}{b} = A\underset{\sim}{x}$.

b. using the five-point-star finite difference formula an elliptic boundary value problem has been converted to the problem of solving linear equations [1]. The problem is:

$\nabla^2 u - 2u = g(x,y)$ inside a unit square R, $0 \le x \le 1$, $0 \le y \le 1$, and $u = 0$ on boundary of R,

where

$$g(x,y) = x^2 + y^2 - x - y - xy(xy - x - y + 1).$$

The problem has the solution

$$u = 1/2xy(x-1)(y-1).$$

The matrix for the second problem is highly sparse for large n, but our code does not take advantage of sparsity in storing or manipulating the elements of A.

A sample of our computational results for several values of n and m is shown in Tables 2 and 3. To predict speedup we used equation (6). The actual execution times t_1^A (one processor) and t_m^A (m processors) are in seconds of the HEP computer.

n	m	t_1^A	t_m^A	S_m	
				Predicted	Actual
10	2	0.0367	0.0196	1.9822	1.8724
24	2	0.1157	0.0581	1.9961	1.9914
	6	0.1426	0.0281	5.7833	5.0747
36	4	0.2328	0.0607	3.9734	3.8353
	6	0.2504	0.0460	5.8952	5.4435
48	4	0.3718	0.0947	3.9845	3.9261
	8	0.3331	0.0466	7.8447	7.1481
64	4	0.4409	0.1128	3.9910	3.9085
	8	0.4777	0.0635	7.9091	7.5228

Table 2. Random matrix.

n	m	t_1^A	t_m^A	S_m	
				Predicted	Actual
16	2	0.0366	0.0189	1.9923	1.9365
	4	0.0542	0.0157	3.8877	3.4522
25	5	0.1505	0.0335	4.8888	4.4925
36	6	0.3099	0.0562	5.8945	5.5142
64	8	1.0508	0.1387	7.9083	7.5761

Table 3. Boundary value problem.

4. Conclusions.

The computational results support our expectation that the PBCG method is very efficient. The efficiency of a parallel method can be measured by the value of

$$E_m = \frac{S_m}{m} \le 1 \qquad (7)$$

and the efficiency of the PBCG is close to the optimal value $E_m = 1$. The idle periods for m - 1 processors are very short as compared with the total execution time. This compares favorably with performance of the parallel LU decomposition and Givens transformation methods for linear equations [4].

In our implementation the PBCG method is bimodal, i.e., either all processors are busy or only one. Of course, it is possible to use more processors for the computation of α^k and β^k but the resulting overhead could eliminate potential advantages. Bimodal methods have been considered by Ware [6] and Worlton [7] who have pointed out that even a small amount of sequential processing can significantly reduce the effectiveness of a multiprocessor if the number of processors p is large. Assume, for instance, the p = 100 and only s = 1/100 of the entire computational work is done sequentially on one processor. The ideal speedup of 100 is reduced to

$$\hat{S}_{100} = \frac{1}{s + (1-s)/p} = 50.25. \qquad (8)$$

On the other hand for p = 10 we have only a small loss since

$$\hat{S}_{10} = 9.17. \qquad (9)$$

Hence, if the execution units of the multiprocessor with p = 10 are ten times faster than the execution units of the multiprocessor with p = 100 we have $t_{10} \cong 1/2 t_{100}$. This led Worlton to the conclusion that there is less risk in the use of multiprocessors having a small number of fast processors than there is in the use of multiprocessors having a large number of slow processors. Our experimentation with the PBCG method is an example illustrating the point.

5. Acknowledgements.

The authors would like to thank Denelcor, Inc. for making available to them the HEP computer and for the research grant. We also thank the reviewers for their helpful suggestions.

6. References.

[1] W. Cheney, and D. Kincaid, Numerical Mathematics and Computing, Brooks/Cole Publishing Company, (1980).

[2] A. Jennings, and G. M. Malik, "The Solution of Sparse Linear Equations by the Conjugate Gradient Method," Numerical Methods in Engineering (12, 1978), pp. 141-158.

[3] S. P. Kumar, Parallel Algorithms for Solving Linear Systems on MIMD Type Computer, Department of Computer Science, Washington State University, (1981).

[4] R. E. Lord, J. S. Kowalik, and S. P. Kumar, "Solving Linear Algebraic Equations on an MIMD Computer," to appear in the Journal of ACM (1982).

[5] B. J. Smith, Architecture and Applications of the HEP Multiprocessor Computer System, proceedings of the 1981 Society for Photographic and Instrumentation Engineering, (August, 1981).

[6] W. H. Ware, "The Ultimate Computer," IEEE Spectrum, (March, 1972) pp. 84-91.

[7] J. Worlton, "Supercomputers - the Philosophy Behind the Machines," Computerworld, (November, 1981), In Depth pp. 1-14.

A MULTI-COLOR SOR METHOD FOR PARALLEL COMPUTATION

L. Adams and J. Ortega
Department of Applied Mathematics and Computer Science
University of Virginia
Charlottesville, Virginia 22901

Abstract*

This paper considers a generalization of the classical red/black ordering of grid points for finite difference or finite element discretizations of elliptic partial differential equations. These "multi-color" orderings are shown to be effective in the implementation of the SOR iteration method on vector or parallel computers. Examples are given of various orderings for different discretizations and implementation on the CDC Cyber 203/205 and the Finite Element Machine is discussed.

Introduction

We are concerned in this paper with the solution of a sparse nxn linear system of equations

$$Ax = b \qquad (1.1)$$

by iterative methods, especially SOR type methods, on parallel arrays or vector computers. As opposed to the Jacobi iteration, which has rather ideal properties for parallel computation, the SOR method is essentially a sequential method. However, several authors (e.g. Hayes [1974], Lambiotte [1975]) have observed that if (1.1) arises from a five-point finite difference discretization of Poisson's equation and the equations are ordered according to the classical Red/Black partitioning of the grid points then an SOR sweep may be carried out, in essence, by two Jacobi sweeps, one on the equations corresponding to the red points and one for the equations corresponding to the black points. Thus, in this case, the SOR method can be effectively implemented on vector or parallel computers.

This strategy does not work, however, for higher order finite difference or finite element discretizations or for more general elliptic equations which contain mixed partial derivative terms. In these cases, it is necessary to generalize the Red/Black partitioning of the grid points to a "multi-color" partitioning; for example, a three color partitioning, say Red/Black/Green, might give the desired result. In general, the number of colors necessary will depend on the connectivity pattern of the grid points. If p colors are used, an SOR sweep can be implemented by p Jacobi sweeps, one for each set of equations associated with a given color. For vector computers, this reduces the effective vector length to O(n/p) while for parallel arrays it is necessary that each processor hold a multiple of p equations. This multiple will be determined by the particular discretization. Clearly, there will be a point of diminishing returns as p increases but for most differential equations and discretizations of interest it seems that no more than 6 colors will suffice and for the size of n we have in mind ($n \simeq 10{,}000 +$), the multi-color strategy can be very effective.

We note that multi-color orderings for SOR have been used before (see Young [1971]) but, to the best of our knowledge, have not been used in the context of parallel computation.

In the next section, we describe the method in more detail and in Section 3 we discuss some of the implementation questions for both vector computers and parallel arrays. We do not address the many other problems in the successful use of the SOR iteration, especially the problem of determining an optimum relaxation parameter.

Multi-Color Orderings

For concreteness, we consider first an elliptic equation of the form

$$u_{xx} + au_{xy} + u_{yy} = f \qquad (2.1)$$

on the unit square with Dirichlet boundary conditions where a is a given constant and f is a given function of x and y. We discretize (2.1) with the usual second-order finite difference approximations (see, e.g., Forsythe and Wasow [1960]) which give the difference equations

$$u_{i+1,j}+u_{i-1,j}+u_{i,j+1}+u_{i,j-1}-4u_{ij}+\frac{a}{4}[u_{i+1,j+1} - u_{i-1,j+1}+u_{i-1,j-1}-u_{i+1,j-1}] = h^2 f_{ij} \qquad (2.2)$$

where h is the spacing between grid points, i,j=1..N where h(N+1)=1, u_{ij} denotes the approximate solution at the i,jth grid point, and $f_{ij}=f(ih,jh)$. Now partition the grid points by the Red/Black scheme, as indicated by Figure 1, and then number the grid points in each class from left to right, bottom to top.

R B R B
B R B R
R B R B
B R B R

Figure 1. Red/Black Ordering

*This research was sponsored by the National Aeronautics and Space Administration under grant number NAG1-46. The work of the second author was partially supported under NASA grant number NAG1-16394 while he was in residence at ICASE.

0190-3918/82/0000/0053$00.75 © 1982 IEEE

If a=0, so that (2.1) is just Poisson's equation, then it is well-known (see e.g. Young [1971]) that the difference equations (2.2) may be written in the partitioned matrix form

$$\begin{bmatrix} D & B \\ B^T & D \end{bmatrix} \begin{bmatrix} u_r \\ u_b \end{bmatrix} = \begin{bmatrix} f_r \\ f_b \end{bmatrix} \tag{2.3}$$

where D is a diagonal matrix and u_r and u_b denote the vectors of unknowns associated with the red and black grid points respectively. The Gauss-Seidel iteration for (2.3) may be written as

$$\begin{aligned} Du_r^{k+1} &= -Bu_b^k + f_r \\ Du_b^{k+1} &= -B^T u_r^{k+1} + f_b \end{aligned} \tag{2.4}$$

and each part of (2.4) can then be effectively implemented in a parallel fashion, with the introduction of the SOR parameter causing no problem.

If $a \neq 0$, the form (2.3) of the difference equations is still valid although D is no longer a diagonal matrix and the Gauss-Seidel step (2.4) is no longer easily implementable in a parallel fashion. The problem is that unknowns corresponding to red points are coupled to each other in (2.3) (and black points to each other also) whereas when a=0 , they completely uncouple. Thus we wish to introduce another partitioning of the grid points for which unknowns within each subset of the partitioning are uncoupled. If we consider the grid point stencil for (2.2), shown in Figure 2,

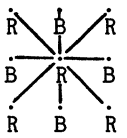


Figure 2. Stencil for (2.2)

with the Red/Black ordering, we see that the center Red point is connected to the Red points at the four corners. If, however, we use four subsets of grid points, labeled red, black, white, orange, we can ensure that each center point connects with only points of different colors. A suitable coloring pattern for this is illustrated in Figure 3.

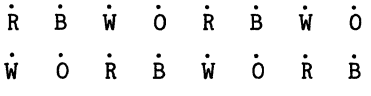


Figure 3. Four color ordering of the gridpoints

In this case, the system (2.2) can be written in a partitioned form analogous to (2.3) as

$$\begin{bmatrix} D_1 & B_{12} & B_{13} & B_{14} \\ B_{21} & D_2 & B_{23} & B_{24} \\ B_{31} & B_{32} & D_3 & B_{34} \\ B_{41} & B_{42} & B_{43} & D_4 \end{bmatrix} \begin{bmatrix} u_r \\ u_b \\ u_w \\ u_o \end{bmatrix} = \begin{bmatrix} f_r \\ f_b \\ f_w \\ f_o \end{bmatrix} \tag{2.5}$$

where D_1, D_2, D_3, and D_4 are diagonal matrices. The Gauss-Seidel iteration in terms of (2.5) is then

$$\begin{aligned} D_1 u_r^{k+1} &= -B_{12}u_b^k - B_{13}u_w^k - B_{14}u_o^k + f_r \\ D_2 u_b^{k+1} &= -B_{21}u_r^{k+1} - B_{23}u_w^k - B_{24}u_o^k + f_b \end{aligned} \tag{2.6}$$

with similar equations for u_w^{k+1} and u_o^{k+1}. Since the D_i are diagonal, (2.6) is easily implementable on vector or parallel architectures.

A variety of other connectivity patterns arise from either finite difference or finite element discretizations. Two of the more common are illustrated by their stencils in Figure 4,

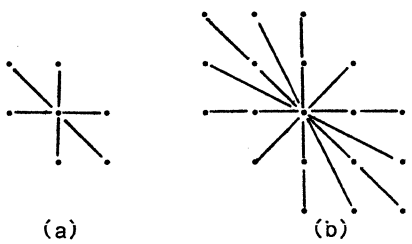


Figure 4. Common finite element stencils

in which (a) arises, for example, from finite element discretization by piecewise linear functions over triangular subregions and (b) by piecewise quadratic functions. In case (a), three colors are necessary and sufficient to achieve the desired decoupling while in case (b) six colors are required. The coloring patterns for the two cases are illustrated in Figure 5.

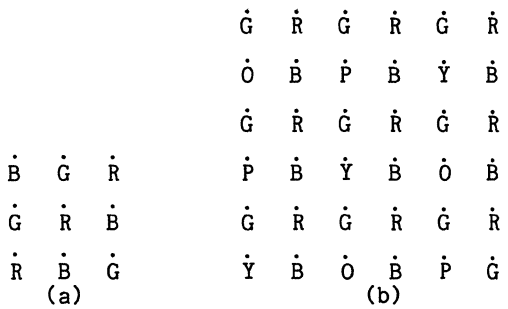


Figure 5. Three and six color partitions

In both cases, the color patterns repeat beyond the subregions illustrated.

A variety of other examples could be given. Provided that the domain of the differential equation is a rectangle or other regular two or three dimensional region and the discretization stencil is repeated at each grid, it is usually evident how to color the grid points to achieve

the desired result. However, for arbitrary discretizations and/or irregular regions there is at present no algorithm to carry out the coloring.

Implementation Considerations

We discuss briefly in this section some of the implementation considerations of the multi-color SOR method on vector computers and parallel arrays. For concreteness, we will use the CDC CYBER 203/205 as an example of the former and the Finite Element Machine at NASA's Langley Research Center as an example of the latter.

On the CYBER 203/205, vectors consist of contiguous storage locations and the efficiency of the vector operations is strongly dependent on vector length. Maximum efficiency is achieved for very long vectors. For vectors of length 1000 around 90% efficiency is obtained, but this drops to approximately 50% or less for vectors of length 100 and less than 10% for length 10. Hence, we would like to keep vector lengths on the order of 1000 or more whenever possible.

Consider, for example, the difference equations (2.2) and suppose that h=.01 so that N=99 and $n=N^2\approx 10^4$. The implementation of Jacobi's method on this problem can be done in a straightforward way using vectors of length N, corresponding to the unknowns in each row of grid points. It is desirable, however, to work with vectors of length order n and it is possible to achieve this by considering the boundary values to be unknowns and ordering all the grid points, including the boundary points, from left to right, bottom to top and then applying the Jacobi iteration to the corresponding vector of length $(N+2)^2$ of unknowns. The boundary values, of course, cannot be changed by the iteration and this is prevented by use of the control vector feature on the 203/205 which allows suppression of storage of updated values into the boundary locations. (See, e.g. Lambiotte [1975] or Ortega and Voigt [1977] for more details on this procedure.) Since the calculation of new values corresponding to the boundary points is superfluous, this introduces an inefficiency of approximately 4% for N=99 but allows almost full efficiency of the vector operations.

For the Gauss-Seidel or SOR method for (2.2) we use the four-color ordering of Figure 3. and order the unknowns into four vectors corresponding to the grid points associated with the four colors. The matrix-theoretic description (2.6) of the Gauss-Seidel iteration is then implemented by four separate Jacobi sweeps, one for each color. As above, the boundary values are considered as unknowns and then updated values suppressed on storage. Since the vector lengths are now on the order of 2500, the corresponding vector operations will run at about 95% efficiency. The introduction of the SOR parameter causes no difficulty.

We turn now to parallel arrays. The Finite Element Machine (FEM) is a prototype array of 36 microprocessors, arranged in a 6x6 grid. Each processor is connected to eight nearest neighbors, as illustrated in Figure 6.

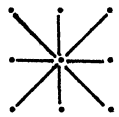


Figure 6. Processor Interconnections on FEM

and there is also a global bus that connects all processors. Further details, which do not concern us here, may be found in Jordan [1978] and the references therein.

Our primary goal in the implementation of the multi-color SOR method on the Finite Element Machine, or on a similar array with perhaps many more processors but limited processor to processor interconnections, is to keep as many processors as possible running at a given time. This, in turn, requires maximum use of the processor interconnections and minimum use of the global bus since contention for the bus will tend to introduce delays which cause processors to be idle.

Perhaps the primary consideration in the implementation is to ensure that each processor holds at least as many unknowns as a certain multiple of the number of colors where this multiple is the number of rows above the center point in the gridpoint interconnection stencil. Thus, for example, if we consider the gridpoint interconnection stencil of Figure 4(a) and the corresponding three color ordering of Figure 5(a), we would assign a minimum of 3 unknowns to each processor as illustrated in Figure 7(a). Similarly, for the stencil of Figure 4(b) and the corresponding six color ordering of Figure 5(b), we would assign a minimum of 12 unknowns to each processor as illustrated in Figure 7(b).

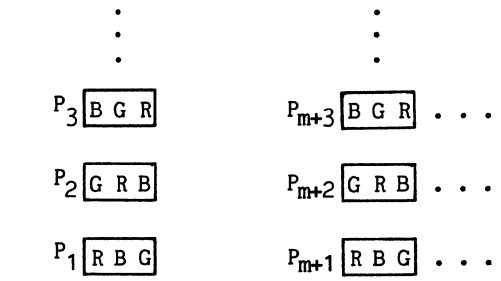


Figure 7(a). Processor Assignment

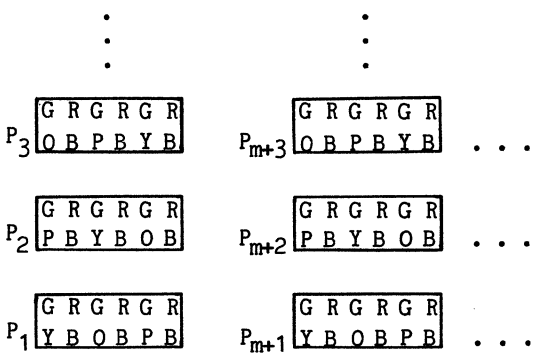


Figure 7(b). Processor Assignment

In the simplest case of 36 processors and 108 grid points, with 108 corresponding unknowns, the assignment scheme of Figure 7(a) would be sufficient and the SOR method would be implemented by Jacobi operations, first on all the Red points, then the Black, then the Green. To carry out these Jacobi operations, current values of neighboring unknowns would be obtained either from the processor itself or a neighbor processor and no use of the global bus is necessary. Known boundary values would be stored in the processors which needed them. In any problem of interest, however, there would almost certainly be many more grid points and unknowns than processors. For the situation discussed above with three colors, we would assign unknowns in multiples of three to the processors. Similarly, for the grid point stencil of Figure 4(b) and corresponding six color pattern of Figure 5(b), we would assign unknowns in multiples of 12 to each processor.

The above assignment strategy would allow each processor to run without waiting except for two problems, synchronization and convergence. Consider a Jacobi operation on all the unknowns of a given color. The processors may complete their work on this operation in different times due to a number of factors: slightly different clock times in the processors; different memory access times, especially for those processors containing unknowns connected to boundary values; different numbers of unknowns assigned to processors and so on. To compensate for these possible differences in processing times, the computation can be synchronized by having each processor set a flag when it is done with its calculation on the current Jacobi operation and then wait for all other processors to complete. This synchronization, of course, introduces delays. Alternately, the processors can run asynchronously. In this case, the numerical iterations will tend to deviate from the true mathematical iteration, although the consequences of this may even be beneficial. (See, e.g. Baudet [1978] and the references therein for further discussion of asynchronous iterative methods.)

It is, of course, necessary to check for convergence of the iterative process. At the end of each SOR iteration, each processor can monitor the convergence of the unknowns assigned to it, probably by comparing the current and previous iterates. When the convergence criterion has been satisfied for all unknowns assigned to a given processor, that processor must continue the iteration until the convergence criterion is satisfied in all processors. Hence, the whole process will not terminate until all unknowns have satisfied the convergence criterion and towards the end of the process a portion of the processors may be doing unnecessary work. This seems to be an unavoidable inefficiency.

Summary and Conclusions

The multi-color SOR method described herein seems promising for vector and array processors although practical experience to date has been limited to a few numerical experiments on a four-processor version of the Finite Element Machine. It faces the usual difficulty with the SOR method of obtaining suitably good values of the overrelaxation parameter and for most applications of current interest for which a vector computer or large array would be used, there is little theory to help in this choice. For irregular regions, there is also the problem of processor assignment and coloring of the grid points; the processor assignment problem has been addressed by various authors (see, e.g. Bokhari [1979] and Gannon [1980]) but not in conjunction with the coloring problem.

References

Baudet, G. [1978]. "Asynchronous Iterative Methods for Multiprocessors," J. Assoc. Comp. Mach. 25, pp. 226-244.

Bokhari, G. [1979]. "On the Mapping Problem," Proc. Int. Conf. on Par. Proc, pp. 239-248.

Forsythe, G. and Wasow, W. [1960]. Finite Difference Methods for Partial Differential Equations, John Wiley, New York.

Gannon, D. [1981]. "On Mapping non-uniform P.D.E. Structures and Algorithms onto Uniform Array Architectures." Proc. 1981 Int. Conf. Par. Proc., pp. 100-105.

Hayes, L. [1974]. "Comparative Analysis of Iterative Techniques for Solving Laplace's Equation on the Unit Square on a Parallel Processor," M.S. Thesis, Department of Mathematics, University of Texas, Austin.

Jordan, H. [1978]. "A Special Purpose Architecture for Finite Element Analysis," Proc. 1978 Int. Conf. on Par. Proc., pp. 263-266.

Lambiotte, J. [1975]. "The Solution of Linear Systems of Equations on a Vector Computer," Ph.D. Dissertation, University of Virginia.

Ortega, J. and Voigt, R. [1977]. "Solutions of Partial Differential Equations on Vector Computers", Proc. 1977 Army Num. Anal. Conf., pp. 475-526.

Young, D. [1971]. Iterative Solution of Large Linear Systems, Academic Press, New York, pp. 427-428.

A PARALLEL ALGORITHM FOR FINDING THE ROOTS OF A POLYNOMIAL

Thomas A. Rice
Leah J. Siegel

Purdue University
School of Electrical Engineering
West Lafayette, Indiana 47907

Abstract -- In many applications, it is necessary to perform the computationally intensive task of extracting the roots of a high order real polynomial. Parallel approaches to the root-finding problem are summarized. A new SIMD (single instruction stream - multiple data stream) algorithm is described. The algorithm is a parallel implementation of Graeffe's method. It can employ a number of processors less than or equal to the degree of the polynomial. The p-processor algorithm achieves an O(p) speedup over the corresponding serial algorithm. This compares favorably with other iterative parallel root-finding algorithms, which have typically used fewer processors than the SIMD Graeffe's method, and which have exhibited O(number of processors) speedup.

1. Introduction

In many applications, including digital signal processing and automatic control, it becomes necessary to extract the (possibly complex) roots of a high order real polynomial equation. Polynomials of degree 10 to 25 are not uncommon; polynomials with degree as high as 100 are sometimes encountered. In this paper, the application of parallel processing to the root-finding problem is examined. Proposed techniques are summarized, and a new parallel algorithm is described.

Since the conventional techniques of root-finding usually involve variable length iterations and repetitive root extraction, they generally do not map immediately to the parallel domain. The main concerns thus become: can the problem be fairly partitioned among a large enough number of processors to gain a reasonable speed-up, and can the interprocessor communications be sufficiently minimized? In addition, methods that have been discarded for serial computation need to be reconsidered for parallel computation if they are easily partitionable.

For a given application, a number of properties of the root-finding method must also be taken into consideration. These include the following: Can the method extract only real or both real and complex roots? Can the method handle multiple roots at the same location? Does the method encounter problems of numerical stability or overflow under some conditions? Does the method require a good initial estimate of a root's location in order to converge?

In Section 2, approaches to using parallelism to extract the roots of a polynomial are discussed. In Section 3, a specific parallel algorithm is presented. The particular root-finding method described is one which has the required properties for both parallel implementation and scientific (in particular, signal processing) application. The attributes required in a parallel machine to implement the algorithm and the computational characteristics of the parallel algorithm are discussed.

2. Approaches to Parallel Root-Finding

There are two principal ways in which root extraction is performed. The first of these is domain division. This consists of partitioning the domain over which roots may occur and then searching for roots in the individual subdomains. For example, after the domain is partitioned, Muller's method [6] could then be used to find the roots in each subdomain. Some parallel methods of solving partial differential equations [e.g., 4] may be applicable to parallel root-finding algorithms based on domain division.

The second principal method of root-finding is the iterative approach, in which successive approximations to the roots are obtained. Examples of this approach are illustrated in [5, 7, 8]. Parallelism can be applied to such algorithms either (1) to attempt to reduce the number of iterations performed or (2) to reduce the execution time per iteration. Parallel methods by Miranker [9, 10], Feldstein and Firestone [10], Shedler [11], and Winograd [15] have attempted to reduce the number of iterations in such methods as Lagrange extrapolation [10], Hermite interpolation [10], and Newton-Raphson [11]. At each step in the iteration for finding a given root, p processors obtain p different, independent approximations for the root. From these, the best approximation to use in the next step is derived. In such methods, it has been shown that the number of iteration steps is reduced by a factor of log p when p processors are used; values of p considered (e.g., in [11]) have typically been small.

3. A Highly Parallel Root-Finding Algorithm

In this section, an iterative root-finding algorithm is presented in which parallelism is used to reduce the execution time per iteration. The algorithm is a parallel implementation of Graeffe's method [6]. This method is not commonly used with serial processors since it is slow in comparison with other serial algorithms. The method can find both real and complex roots and can be adapted to find roots with multiplicity greater than one.

This material is based on work supported by the National Science Foundation under Grant ECS-790916.

0190-3918/82/0000/0057$00.75 © 1982 IEEE

Graeffe's Method

Graeffe's method is based upon forming a sequence of equations whose zeros are the squares of the zeros of the previous equation in the sequence. This is done to separate the roots in the equation so that they can be obtained by solving a set of linear equations. For example, consider the polynomial equation

$$p(x) = x^3 + a_1x^2 + a_2x + a_3$$
$$= (x-z_1)(x-z_2)(x-z_3) = 0.$$

If the magnitude of z_1 is much larger than the magnitude of z_2, which is in turn much larger than the magnitude of z_3, then p(x) is approximately

$$x^3 - z_1x^2 + z_1z_2x - z_1z_2z_3 = 0.$$

Thus, $a_1 = -z_1$, $a_2 = z_1z_2 = -a_1z_2$, and $a_3 = -z_1z_2z_3 = -a_2z_3$.

The root squaring processes is based on forming the product of $p(x)p(-x)(-1)^n$, where n is the degree of the original equation. This results in a polynomial of a degree twice that of p(x), but with only even powers of x. If each x^2 is replaced by x, the result is an equation of the same degree as p(x), but with roots that are the squares of the roots of p(x). This procedure is repeated until each coefficient in an equation is the square of the corresponding coefficient in the previous equation, within a desired tolerance.

Assume that k squarings (iterations) are needed to satisfy the tolerance requirement, and let $m=2^k$. The final equation can be solved to give the magnitudes of the m-th powers of the roots. Substitution is used to find the actual roots. By examining the form of the final few equations in the sequence, one can determine the types of roots (real or complex) that are in the equation. In general, if z_j and z_{j+1} are a complex conjugate root pair, then the coefficient of x^{n-j} will oscillate. Once this oscillation meets certain tolerance requirements, the magnitude of the roots and the cosine of m times the phase angle can be determined. The actual angle must be determined by trial.

The principle computation in Graeffe's method is in the repeated evaluation of $p(x)p(-x)(-1)^n$. Let the current equation be given by

$$B_0x^n + B_1x^{n-1} + \ldots + B_{n-1}x + B_n \qquad (1)$$

and let the next equation in the sequence (after replacing x^2 by χ) be given by

$$C_0x^n + C_1x^{n-1} + \ldots + C_{n-1}x + C_n . \qquad (2)$$

It has been shown [6] that the j-th coefficient, C_j, can be evaluated from the previous set of coefficients by the equation

$$C_j = B_j^2 - 2B_{j-1}B_{j+1} + 2B_{j-2}B_{j+2} - \ldots \qquad (3)$$

For each C_j, the evaluation stops when the subscripts on the required Bs fall outside the range of the coefficient set.

Unlike most iterative root-finding methods, Graeffe's method has the advantage of computing all of the roots in parallel and of having a basically parallel structure. This algorithm for evaluating one set of coefficients from the previous set is the basis of the parallel approach.

Graeffe's Method - A Parallel Algorithm

Based on the explanation in the previous section, the general algorithm to implement Graeffe's method will be of the form:

```
begin(findroot)
     k = 0      /*k is iteration number */
     while (termination criteria not met)
     begin(loop)
              evaluate next set of coefficients
              k = k + 1
     end(loop)
     solve for roots
end.(findroot)
```

The heart of the algorithm is the finding of the new set of coefficients. Therefore, this part of the algorithm will be considered first.

In the parallel implementation, each processor will compute one of the coefficients for the next equation in the iteration. Thus, for an n degree (n+1 coefficient) polynomial, p = n+1 processors can be used. (A smaller number of processors can be used with a corresponding increase in execution time.) During a given iteration, the same operations are performed to obtain each coefficient, but on different data, so SIMD (single instruction stream - multiple data stream) parallelism is indicated. The SIMD machine model used will consist of a control unit, interconnection network, and p PEs (processing elements), where each PE is a processor-memory pair [12]. The p PEs are numbered from 0 through p-1, with 0 denoting the rightmost PE and p-1 the leftmost PE. Each PE will initially contain one of the coefficients of the polynomial. It will be assumed that PE j, $0 \le j \le n$, contains the coefficient of x^{n-j}, i.e., at each iteration, PE j holds B_j, then C_j.

The procedure in Fig. 1 computes the new set of coefficients from the old set. In the algorithm, lcycle and rcycle denote the execution of inter-PE transfers. In lcycle, the value in PE j is transferred to the variable of the same name in PE j+1. The transfer occurs simultaneously for all j. The value that was in PE p-1 is lost, and a zero is shifted into the transfer variable in PE 0. Rcycle is similar.

Fig. 2 illustrates the data movement in procedure "coefficient." In each iteration, the coefficients can be found in parallel using a sequence of $\lfloor n/2 \rfloor + 1$ steps, in which each step consists of two transfers, three multiplications and one subtraction. The data transfers required are from each PE to its two nearest neighbors. The actual

```
procedure coefficient(old,new)
/* input:  old coefficients in variable "old"
   output: new coefficients in variable "new"
      "old" in PE j is B_j in eqn. (1)

      "new" in PE j is C_j in eqn. (2)

*/
   local variable a,ℓ,r;
   ℓ = old;  /* B obtained via left shift */
   r = old;  /* B obtained via right shift */
   a = old**2;  /* will accumulate result */
   for q = 0 until ⌊n/2⌋ do
   begin(loop)
        lcycle(ℓ);      /* left shift */
        rcycle(r);      /* right shift */
        a = a - 2 * (-1)**q * ℓ * r;
                  /* add next term in sequence */
   end(loop)
   new = a;
end.(coefficient)
```

Fig. 1. Procedure to compute the next set of coefficients.

	PE #	6	5	4	3	2	1	0
initial values		ℓ_6	ℓ_5	ℓ_4	ℓ_3	ℓ_2	ℓ_1	ℓ_0
		r_6	r_5	r_4	r_3	r_2	r_1	r_0
iteration 1		ℓ_5	ℓ_4	ℓ_3	ℓ_2	ℓ_1	ℓ_0	0
		0	r_6	r_5	r_4	r_3	r_2	r_1
iteration 2		ℓ_4	ℓ_3	ℓ_2	ℓ_1	ℓ_0	0	0
		0	0	r_6	r_5	r_4	r_3	r_2
iteration 3		ℓ_3	ℓ_2	ℓ_1	ℓ_0	0	0	0
		0	0	0	r_6	r_5	r_4	r_3

Fig. 2. Variable movement in procedure "coefficient," for n = 6, p = 7.

steps required to perform an lcycle or rcycle transfer will depend on the hardware organization of the particular parallel machine. A transfer can be done in one pass through most interconnection networks, with appropriate masking. The time to perform a transfer will therefore be small. (Barnes and Lundstrom report a 120 ns connection time for a 10-stage multistage network [2]. In a system with single stage ring or nearest neighbor connections, transfer time could be expected to be even less.) Depending on the relative time to perform arithmetic and transfer operations, it may be possible to overlap the network transfers with the computations. In this case, the time incurred by the data transfers will be negligible. Overhead is also introduced by the fact that the $\lfloor n/2 \rfloor+1$ steps are performed in each PE, even though all of the coefficients do not need this many steps. The lcycle and rcycle functions shift zeros into the "edge" PEs to nullify the effect of the extra multiplications and subtractions. These multiplications by zero are steps performed in the SIMD algorithm that are not executed in a serial algorithm. In one iteration, the (n+1)-PE SIMD algorithm will perform $3(\lfloor n/2 \rfloor+1)$ multiplications and $\lfloor n/2 \rfloor+1$ subtractions, compared to $3\lfloor n/2 \rfloor(n-\lfloor n/2 \rfloor)$ multiplications and $\lfloor n/2 \rfloor(n-\lfloor n/2 \rfloor)$ subtractions required in the serial algorithm. The speedup on arithmetic operations is therefore approximately p/2 for the p-PE algorithm.

The next step to be considered in Graeffe's method is the determination of whether or not the termination criteria have been met. Fig. 3 details an algorithm for this step. There are two cases: non-oscillitory and oscillitory. For the first, the difference between the magnitude of the current coefficient and the square of the previous coefficient is compared with the termination tolerance. This is done in one comparison step, performed simultaneously in all PEs. As a result of this step, those PEs in which a real root has been located to sufficient accuracy can be identified. If the non-oscillatory tolerance has been met in all PEs, there is no need to test the criteria for oscillatory tolerance. In the algorithm in Fig. 3, this condition is tested in the "if all" statement, which has value 1 if all PEs satisfy the stated condition. The way in which the "if all" is performed will depend on the parallel architecture. If it can be accomplished efficiently, this capability to evaluate a condition across all PEs can, in some cases, eliminate the need to test the oscillating criteria. Such statements (if all, if any) are implemented in PEPE [14]. A possible implementation for the PASM multimicroprocessor system is described in [13].

For oscillatory termination, one approach to determining whether or not to terminate is to obtain the phase angle based on both the new and old sets of coefficients. If the phase angle is the same, within a specified tolerance, for both coefficient sets, the criterion can be considered met. If oscillating termination is to be tested, each PE obtains, via lcycle and rcycle transfers, the current and previous coefficients from its two nearest neighbors. The coefficients (B_{j-1},B_j,B_{j+1}) and (C_{j-1},C_j,C_{j+1}) are used to determine if non-oscillatory tolerance is met in PE j. In the algorithm, PEs in which oscillatory tolerance is to be tested are enabled (and PEs which met non-oscillatory tolerance are disabled) by means of a "where" statement. The "where" construct is a data conditional mask [1, 3] in which each PE evaluates the condition using its own data, and sets its active/inactive status so that it is active for the statements following the "where" only if the condition is true. PEs in which the condition is false are disabled for those statements. At each iteration, the test for oscillatory tolerance is performed simultaneously in all PEs which fail the non-oscillatory tolerance test.

After the coefficient sequences satisfy the termination criteria, all of the real roots can be found in one parallel step and all of the complex roots can be found in one parallel step. This is detailed in Fig. 4. The complete root-finding algorithm is given in Fig. 5.

```
procedure tolerance(old,new,n_osc,tol_ok,osc)
/* input:
   old = previous set of coefficients
   new = current set of coefficients
   output:
   tol_ok = bit vector indicating where
            tolerance was met
   n_osc  = bit vector indicating where non-
            oscillatory tolerance was satisfied
   osc    = bit vector indicating where
            oscillatory tolerance was satisfied
   Oscillations are detected using β computed
   from old and new coefs:

         β rmold, β rmnew = β_r^m

         2β_r^m cos(mθ)=(E_r/E_{r-1})

         β_r^{2m}=(E_{r+1}/E_{r-1})

         where E_j can be either B_j or C_j,

         and if this is iteration k, m=2^k;
         0 < r < p-1
*/
    local_variable ℓold,rold,ℓnew,rnew;
    tol_ok = n_osc = osc = 0;
    where (error(old**2,new) < tol_criterion1)
          tol_ok = n_osc = 1;
    /* if the new coefficient is the square of the
       old one within a specified error, then the
       criterion is met.  This check is done in
       parallel in all PEs */
    if (not all (n_osc)) then
    begin(oscillatory)
        ℓold = old;
        rold = old;
        rcycle(rold);              /* B(j+1) */
        lcycle(ℓold);              /* B(j-1) */
        rnew = new;
        ℓnew = new;
        rcycle(rnew);              /* C(j+1) */
        lcycle(ℓnew);              /* C(j-1) */
        β rmold = sqrt(rold/ℓold);
        β rmnew = sqrt(rnew/ℓnew);
        where(error(arccos(old/(ℓold * β rmold*2)),
              arccos(new/(ℓnew * β rmnew*2))/2.0)
              < tol_criterion2)
                 tol_ok = osc = 1;
       /* Combine the two equations involving β to
       determine a possible θ for both of the last
       two coefficient sequences, and compare the
       two values of θ.  This is done in parallel
       in all PES that did not satisfy the squar-
       ing tolerance check */
    end(oscillatory)
end.(tolerance)
```

Fig. 3. Procedure to determine if termination criteria have been met.

```
procedure findroot(new,n_osc,tol_ok,osc,z,mag,ang)
/* input:
   new = elements of coefficient array
   n_osc,tol_ok,osc -- as in procedure tolerance
   output:
   z -- location of zero if root is real
   mag, ang -- magnitude and angle if
        root is complex
*/
   local_variable ℓ,r;
   tol_ok = .not.(osc .or. rcycle(osc))
      /* PEs not involved with  oscillatory  cases
         (right  shift  is  due to the fact that a
         complex root affects two adjacent coeffi-
         cients) */
  z = mag = ang = INVALID;      /* no roots yet */
  lcycle(ℓ);              /* new(j-1) */
  rcycle(r);              /* new(j+1) */
  where (n_osc .and. tol_ok)
  begin(non-oscillatory)
  /* all real roots are found in parallel */
       z = (new/ℓ) ** (1/(2**k));
       /* (new/ℓ)^{-m} */
       where (abs_value(p(z) > tol_criterion3)
            z = -z; /* test by evaluating p(z) */
  end(non-oscillatory)
  where (osc)
  begin(oscillatory)
  /* all complex roots are found in parallel */
       mag = (r/ℓ)**(1/(2*(2**k)));
       /* magnitude of complex pair */
       ang = arccos(new/(2*ℓ*sqrt(r/ℓ)))/(2**k);
       /* possible angle */
  end(oscillatory)
end.(findroot)
```

Fig. 4. Procedure to compute the roots from the final set of coefficients.

```
program parallel_root(new)
/* input:  new = coefficients of p(x)
   output: roots
*/
   k = 0;
   repeat
        old = new;
        coefficient(old,new);
        tolerance(old,new,n_osc,tol_ok,osc);
        k = k + 1;
   until (all (tol_ok));
   findroot(new,n_osc,tol_ok,osc,z,mag,angle);
end.(parallel_root)
```

Fig. 5. Program to perform parallel Graeffe's method.

4. Conclusions

Although exact comparisons between this parallel method and serial methods are difficult due to the data-dependent, iterative nature of the algorithms, some general comparisons can be made between the parallel and serial versions of Graeffe's method. First, the number of iteration steps performed will be the same as in the serial algorithm. However, within each step, the coefficients for the next equation in the sequence are computed in parallel rather than serially. The computational speedup will be approximately p/2 for computing the coefficient sequences. The less than ideal speedup is due to redundant operations required for the parallel algorithm. Computational speedup in evaluating the termination conditions will depend on the relative number of real and complex roots, and will range from approximately 0.2p to 0.9p. Speedup on computations for finding the roots from the final set of coefficients can be up to p/2, depending on the relative number of real and complex roots. The execution time will be clearly dominated by the computation of the coefficient sequences. The interprocessor communications required in the algorithm are from each PE to its two nearest neighbors. The ratio of arithmetic steps to transfer steps in each iteration is approximately 5:1. However, unless transfers are significantly slower than arithmetic operations, it will be possible to overlap the time spent in inter-PE communications with the time spent performing computations. Overall speedup will therefore be dictated by the speedup on computations, and will be on the order of p/2. This compares favorably with other iterative parallel root-finding algorithms, for which implementations using p processors have exhibited O(log p) speedup. For these methods, values of p considered have typically been smaller than the number of processors used by the SIMD Graeffe's method presented here.

In summary, a new parallel algorithm to perform root-finding has been developed. The approach taken in the algorithm differs significantly from that of other parallel root-finding methods.

References

[1] G. Barnes, et al., "The Illiac IV computer," IEEE Trans. Comp., Vol. C-18, Aug. 1968, pp. 746-757.

[2] G. H. Barnes and S. F. Lundstrom, "Design and validation of a connection network for many-processor multiprocessor systems," Computer, Dec. 1981, pp. 31-41.

[3] B. A. Crane, "PEPE computer architecture," Compcon 72, Sept. 1972, pp. 57-60.

[4] D. Gannon, "On mapping non-uniform P.D.E. structures and algorithms onto uniform array architectures," 1981 Int. Conf. Parallel Processing, Aug. 1981, pp. 100-105.

[5] P. Henrici, Elements of Numerical Analysis, John Wiley and Sons, Inc., NY, 1964, pp. 146-179.

[6] F. B. Hildenbrand, Introduction to Numerical Analysis, McGraw-Hill, NY, 1974, pp. 602-608.

[7] M. A. Jenkins and J. F. Traub, "A three-stage algorithm for real polynomials using quadratic iteration," SIAM J. Numerical Analysis, Vol. 7, Dec. 1970, pp. 545-566.

[8] M. A. Jenkins and J. F. Traub, "Zeros of a real polynomial," ACM Trans. Mathematical Software, 1975.

[9] W. L. Miranker, "Parallel methods for approximating the root of a function," IBM J. Res. Develop., Vol. 13, 1969, pp. 297-301.

[10] W. L. Miranker, "A survey of parallelism in numerical analysis," SIAM Review, Vol. 13, Oct. 1971, pp. 524-547.

[11] G. S. Shedler, "Parallel numerical methods for the solution of equations," Comm. ACM, Vol. 10, May 1967, pp. 286-290.

[12] H. J. Siegel, "A model of SIMD machines and a comparison of various interconnection networks," IEEE Trans. Comp., Vol. C-28, Dec. 1979, pp. 907-917.

[13] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller, Jr., H. E. Smalley, Jr., and S. D. Smith, "PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition," IEEE Trans. Comp., Vol. C-30, Dec. 1981, pp. 934-947.

[14] K. J. Thurber, Large Scale Computer Architecture, Hayden Book Co., Inc., Rochell Park, NJ, 1976, p. 243.

[15] S. Winograd, "Parallel iteration methods," in Complexity of Computer Computations, R.E. Miller and J.W. Thatcher, eds., Plenum Press, NY, 1972, pp. 53-60.

OPTIMIZING THE FACR(ℓ) POISSON-SOLVER ON PARALLEL COMPUTERS

R. W. Hockney

Computer Science Department
Reading University
Reading, Berks. UK. RG6 2AX

Abstract-- A two parameter description of any computer is given that characterises the performance of serial, pipelined and array-like architectures. The first parameter (r_∞) is the traditional maximum performance in megaflops, and the new second parameter ($n_{1/2}$) measures the apparent parallelism of the computer. The relative performance of two algorithms on the same computer, depends only on $n_{1/2}$ and the average vector length of the algorithm. The performance of a family of FACR direct methods for solving Poisson's equation is optimized on the basis of this characterisation.

Parallel Computers

A two-parameter description of the performance of any computer can be obtained by fitting the best straight line to the measured time, t, to perform a single vector operation on vectors of varying length , n, (e.g. A=B*C, where A, B and C are vectors). A similar description of computer performance has been developed by Calahan, Ames and their coworkers at the University of Michigan (see [2] and the references therein). Our work below differs in the definition of parameters and the use made of them. Two equivalent generic forms for the straight line define two primary and one useful secondary derived parameter:

$$t = r_\infty^{-1}(n+n_{1/2}) \qquad (1)$$

where

r_∞: **(maximum or asymptotic performance)** the maximum number of elemental arithmetic operations (i.e. operations between pairs of numbers) per second, usually measured in megaflops. This occurs for infinite vector length on the generic computer.

$n_{1/2}$: **(half-performance length)** the vector length required to achieve half the maximum performance.

Alternatively, when $n<n_{1/2}$, the generic line may be more usefully expressed as:

$$t = \pi^{-1}(1+n/n_{1/2}) \qquad (2)$$

where

π: **(specific performance)** or performance per unit parallelism, is defined as the ratio $r_\infty/n_{1/2}$.

The above definitions are shown graphically in Fig. 1 where we find:

r_∞ is the inverse slope of the generic line
$n_{1/2}$ is its negative intercept on the n-axis
π is its inverse intercept on the t-axis

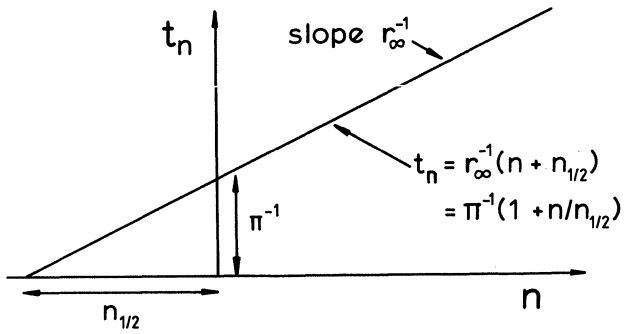


Fig. 1. The timing diagram for the generic parallel computer, showing the definitions of the parameters, r_∞, $n_{1/2}$ and π. (From Hockney and Jesshope 1981, courtesy of Adam Hilger).

It is useful to examine the values of r_∞ and $n_{1/2}$ that are expected from the common forms of computer architecture. This is done by considering the timing line for each type:

(a) Serial Computer - the execution time is proportional to the number of elemental operations

$$t = t_1 n \qquad (3)$$

where t_1 is the time for one elemental operation.

0190-3918/82/0000/0062$00.75 © 1982 IEEE

Comparison with Eqn. (1) shows that for a serial computer

$$r_\infty = t_1^{-1}, \qquad n_{1/2} = 0 \qquad (4)$$

(b) Pipelined Computer - the execution time is normally expressed by the manufacturers in a form similar to

$$t = (s+\ell+n-1)\tau \qquad (5)$$

where

τ is the clock period
s is the startup time in clock periods
ℓ is the number of segments in the arithmetic pipeline

Comparison with Eqn. (1) shows that for a pipelined computer

$$r_\infty = \tau^{-1}, \qquad n_{1/2} = s+\ell-1 \qquad (6)$$

(c) Processor Array - if there are N processors which simultaneously perform the same arithmetic operation on N elements of each vector (one element of each vector in each processor's memory), then the timing graph is stepwise as shown in Fig. 2

$$t = t_p\lceil n/N\rceil \qquad (7)$$

where

$\lceil x\rceil$ is the ceiling function of x, i.e. the smallest integer which is equal to or greater than x.

t_p is the time for one parallel arithmetic operation of all processors in the array.

The best straight line through the timing graph is the dotted line which corresponds to

$$r_\infty = N/t_p, \qquad n_{1/2} = N/2 \qquad (8)$$

This choice of parameters describes approximately the average behaviour of the array if the vectors presented to it are of varying lengths, more or less uniformly distributed.

On the other hand one may know that the vector length is always less than the number of processors ($n \leqslant N$) and that therefore one is always working on the first step of the timing graph. In this case the behaviour is exactly described by the second generic form with

$$\pi = t_p^{-1}, \qquad n_{1/2} = \infty \qquad (9)$$

We note that this condition is the one assumed in the complexity theory of parallel algorithms: that is to say that there are always enough processors. This can occur in general for the theoretical paracomputer which has an infinite number of processors. It is nice that in our formalism this theoretical limit occurs when $n_{1/2}=\infty$.

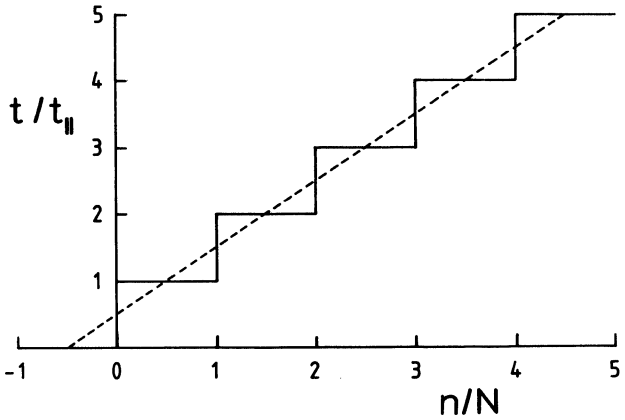


Fig. 2. The timing diagram for an array of N processing elements (solid line), showing the best approximating generic straight line (dotted) which determines the value of $n_{1/2}$ as N/2. (From Hockney and Jesshope 1981, courtesy of Adam Hilger).

The above theoretical results for a range of widely different computer architectures suggest that $n_{1/2}$ is a measure of the parallelism of the computer hardware, varying from zero for a serial computer with no parallelism to infinity for the infinite array of processors. The exception is the pipelined computer in which a large value of $n_{1/2}$ can occur either for a large amount of parallel operation in the pipeline (the number of segments ℓ is large), or for a large value of the setup time , s. In the former case $n_{1/2}$ is measuring the hardware parallelism, but in the latter case it is measuring an overhead. From the users , or algorithmic, point of view the behaviour of the computer is determined by the timing expression (1) and the value of $n_{1/2}$, however it arises. A pipelined computer with a large value of $n_{1/2}$ appears and behaves as though it has a high level of parallelism, even though this might be due to a long setup time. Hence we regard $n_{1/2}$ as a measure of the apparent parallelism of the computer, and from the algorithmic (i.e. timing) point of view it simply does not matter how much of this is real. The fact that true parallelism and setup time are interchangeable, incidently, shows that parallelism is an overhead, and therefore undesirable (by which we mean that parallelism is best avoided if at all possible,

or that one should always seek to achieve the required performance with the least possible parallelism).

The values of $n_{1/2}$ and r_∞ of a computer are best regarded as measured quantities obtained by executing the following FORTRAN code and plotting the timing graph of T against N:

```
                CALL SECOND(T1)
                CALL SECOND(T2)
                T0 = T2-T1

                DO 20 N = 1,NMAX        (10)
                CALL SECOND(T1)

                DO 10 I = 1,N
10              A(I) = B(I) * C(I)

                CALL SECOND(T2)
20              T = T2 -T1 -T0
```

In the above code, the DO 10 loop will be replaced by a single vector instruction by any vectorizing compiler. The measurement and subtraction of the timing overhead T0 is essential because, as we have seen, any overhead will appear as a contribution to $n_{1/2}$. In this case the overhead of measurement is nothing to do with the time of execution of the vector operation, and must therefore be subtracted.

The characterisation of the performance of computers by two parameters naturally leads to plotting computers as points in the two-dimensional $(n_{1/2}, r_\infty)$ phase plane, as is done for some well known designs in Fig. 3. In practice most computers may operate in different modes (scalar or vector, dyadic or triadic operations, different word lengths etc.) and therefore appear as a series of dots, joined to form a "constellation" in the diagram. The traditional characterisation of computer performance by the single parameter r_∞, corresponds to projecting this diagram onto, or viewing it through, the vertical axis. In the era of serial computers all of which have the same $n_{1/2}$ of zero, this was clearly valid. However in the age of the parallel computer, it is obviously important to recognise the different levels of apparent parallelism by spreading the computers out along the $n_{1/2}$ axis. We call this the two-dimensional spectrum of computers. We shall see in the next section that $n_{1/2}$ determines the choice of the best algorithm, and hence is a very important axis. As examples, Fig. 3 shows that the CRAY-1 ($n_{1/2}\approx 10$) and the CYBER 205 ($n_{1/2}\approx 100$) have similar values of r_∞ but behave very differently because their values of $n_{1/2}$ differ by a factor 10. For the same reason, the ICL DAP ($n_{1/2}\approx 1000$) differs from both the CRAY-1 and CYBER 205.

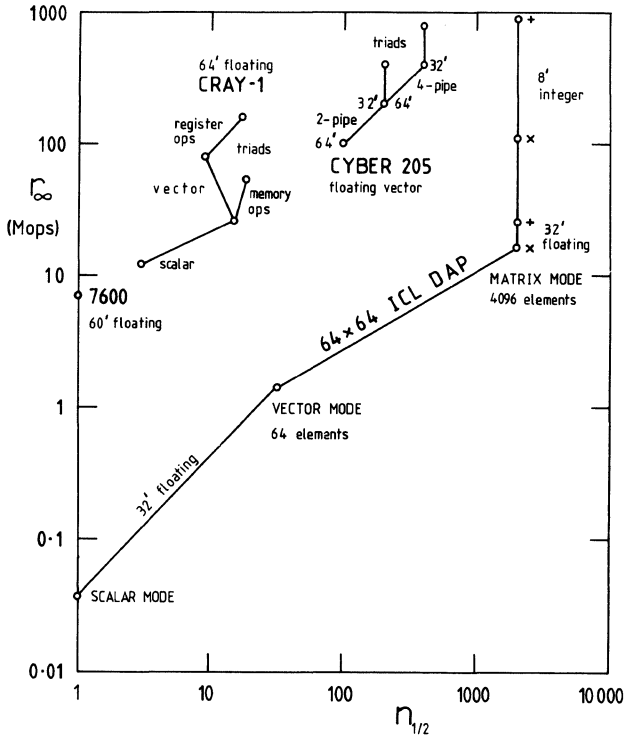


Fig. 3. The two-dimensional spectrum of computers, showing the CRAY-1, CYBER 205 and ICL DAP. (After Hockney and Jesshope 1981, courtesy of Adam Hilger).

Parallel Algorithms

To a first approximation an algorithm can be regarded as a sequence of vector operations of varying length (including one). Such a representation, of course, neglects many factors that may be important (even crucial) in particular cases. Such factors may be, for example, memory bank conflicts in pipelined computers, data routing delays in processor arrays, and the simultaneous operation of scalar and vector units. However we have to start somewhere and avoid too much complication if we are to obtain manageable results. Therefore, in common with other theoretical analyses of algorithm performance, we shall assume such factors are unimportant and express the total time, T, for the execution of an algorithm as

$$T = r_\infty^{-1} \sum_{\ell=1}^{lmax} q_\ell (p_\ell + n_{1/2}) \qquad (11)$$

where we regard the algorithm as lmax sequential stages, ℓ, each composed of q_ℓ vector operations of length p_ℓ. The generic timing formula (1) is then used to buildup the expression (11).

It is useful to define the following quantities:

$$q = \sum_{\ell=1}^{lmax} q_\ell$$

the total number of vector operations, the parallel operations' count or, in the language of complexity theory, the number of unit timesteps.

$$s = \sum_{\ell=1}^{lmax} q_\ell p_\ell$$

the number of elemental operations, or the traditional serial (scalar) operations' count.

$$\bar{p} = s/q$$

the average vector length, or average parallelism of the algorithm.

Using these variables the time of execution of an algorithm can be expressed either as

$$T = r_\infty^{-1} q(\bar{p}+n_{1/2}) \qquad (12)$$

where the algorithm is regarded as q sequential vector operations with average vector length $\bar{p}$, or as

$$T = r_\infty^{-1}(s+n_{1/2}q) \qquad (13)$$

where the first term is the contribution from the traditional count of all elemental arithmetic operations, and the second term is the contribution from the number of parallel (i.e. vector) operations. Equation (13) demonstrates clearly the role of $n_{1/2}$ in interpolating between the extremes of the serial computer ($n_{1/2}=0$) and the infinitely parallel computer ($n_{1/2}=\infty$). For serial computers only the first term or elemental operations' count matters. For the infinitely parallel computer only the second term or the number of parallel operations matters. For computers with finite parallelism, a linear combination of the two operations' counts is appropriate, and the value of $n_{1/2}$ gives the weighting between the two. Since $n_{1/2}=\infty$ corresponds to the assumptions made in the complexity analysis of parallel algorithms and q is the number of unit timesteps in such an analysis, equation (13) shows also how $n_{1/2}$ interpolates rationally between the extreme assumptions that are used in complexity analysis and those that have traditionally been used in the analysis of algorithms on serial computers.

It is instructive to relate the quantities defined above to those introduced by Kuck [3] for the analysis of parallel algorithms. The most important of these is SPEEDUP which relates the speed of an algorithm on a parallel multiprocessor array to the speed of the same algorithm on a serial uniprocessor with the same speed arithmetic units. Thus

$$\text{SPEEDUP} = \qquad (14)$$

$$= \frac{\text{time of execution on uniprocessor}}{\text{time of execution on multiprocessor}}$$

$$= \frac{\text{number of elemental operations}}{\text{number of parallel operations}}$$

$$= \frac{s}{q} = \bar{p}$$

That is to say the SPEEDUP is nothing other than the average vector length (or parallelism) of the algorithm.

The use of the SPEEDUP factor as a figure of merit for parallel algorithms can be misleading because it is only one of several factors that must be considered in any comparison between a real parallel multiprocessor array and a real serial uniprocessor. Let us define the performance (or speed), P, of an algorithm as the inverse of its time of execution, that is to say T^{-1}, the number of executions of the algorithm that are possible per second. Then the relative performance is given by

$$\frac{P_p}{P_s} = \frac{T_s}{T_p} = \frac{s_s x t_s}{q_p x t_p} \qquad (15)$$

where the subscripts s and p refer to the serial uniprocessor and parallel multiprocessor respectively, and t_s and t_p are respectively the time for a serial and a parallel operation. Equation (15) can be expressed as

$$\frac{P_p}{P_s} = \frac{s_p}{q_p} \times \frac{s_s}{s_p} \times \frac{t_s}{t_p} \qquad (16)$$

$$= \text{SPEEDUP} \times \text{algorithmic SLOWDOWN} \times$$

$$\times \text{ hardware SLOWDOWN}$$

The first factor in equation (16) is the SPEEDUP factor previously defined, however the second and third factors are SLOWDOWN factors. In order for the parallel multiprocessor to outperform the serial uniprocessor, it is necessary that the product of the SPEEDUP and the SLOWDOWN factors be greater than one. It is not sufficient that the SPEEDUP factor alone be greater than one. The first SLOWDOWN factor, the algorithmic SLOWDOWN, arises because the definition of SPEEDUP assumes

that the parallel algorithm is executed on the serial uniprocessor with an elemental operations' count of s_p. Almost certainly an algorithm chosen for a parallel computer will not be the best on a serial computer, and the number of elemental operations in the best serial algorithm s_s will almost certainly be less than s_p. Hence the algorithmic SLOWDOWN factor $\frac{s_s}{s_p} < 1$ (typically 1/5).

The second SLOWDOWN factor, the hardware SLOWDOWN, expresses the fact that if the multiprocessor and uniprocessor consume comparable resources, either in money , in number of chips, or in square millimetres of silicon, then the time to perform a serial operation on the uniprocessor, t_s will be much less than the time to perform a parallel operation on the multiprocessor, t_p. In other words if you build many thousands of processors, each of them is going to be very slow compared with the speed of a single processor built or purchased with the same resources. Hardware SLOWDOWN factors are likely to be very small ($\approx 10^{-3}$ to 10^{-4}). To take an extreme example, the CRAY-1 acts like a serial uniprocessor (small $n_{1/2} \approx 10$) and can produce an arithmetic result every 12.5 ns ($=t_s$). On the other hand, the ICL DAP is a parallel array of 4096 processors and performs a parallel operation in about 250 μs ($=t_p$). For these two computers the hardware SLOWDOWN is about 1/20,000. Taking the two example SLOWDOWN factors, we see that the SPEEDUP might have to exceed 100,000 before the parallel multiprocessor is likely to outperform the serial uniprocessor.

Traditional methods for comparing the performance of algorithms are based either on the assumption that the computers are serial, when we compare the elemental operations' count, s; or on the assumption that the computers are array-like and always with sufficient processors, when we compare the parallel operations' count, q. We prefer to use the more general timing expression (12) or (13) and obtain a performance comparison for computers with finite values of $n_{1/2}$. Suppose we compare the performance of algorithm (a) on computer (1) with algorithm (b) on computer (2), then

$$\frac{p^{(a,1)}}{p^{(b,2)}} = \frac{T^{(b,2)}}{T^{(a,1)}} \qquad (17)$$

$$= \frac{(s^{(b)}+n_{1/2}^{(2)}q^{(b)})}{(s^{(a)}+n_{1/2}^{(1)}q^{(a)})} \times \frac{r_\infty^{(1)}}{r_\infty^{(2)}} \times \frac{C^{(2)}}{C^{(1)}}$$

In the above, superscripts are used to distinguish the computer or algorithm; and we note that the algorithm is specified by the value of s and q (or $\bar{p}$ and q) and the computer is specified by values of $n_{1/2}$ and r_∞ (or $n_{1/2}$ and π). The first two factors in Eqn. (17) come from the timing expression (13) and the last factor may be added if the cost of computer time is a relevant factor. C denotes the cost per unit computer time.

Equation (17) is general and compares the cost performance of different algorithms on different computers. If, however, we limit consideration to the choice of the better algorithm (in the sense of having the higher performance) on a particular computer, then Eqn. (17) reduces to

$$\frac{p^{(a)}}{p^{(b)}} = \frac{s^{(b)}+n_{1/2}q^{(b)}}{s^{(a)}+n_{1/2}q^{(a)}} \qquad (18)$$

in which we note that the second and third factors in Eqn. (17) reduce to unity, and that the choice of the better algorithm depends only on the $n_{1/2}$ of the computer and the s and q operations' counts of the algorithms.

In the comparison of algorithms, the equal performance line along which $p^{(a)}=p^{(b)}$ plays a key role because it divides regions of phase planes in which algorithm (a) has the better performance from regions in which algorithm (b) has the better performance. Along the equal performance line we have

$$n_{1/2} = \frac{s^{(b)}-s^{(a)}}{q^{(a)}-q^{(b)}} \qquad (19)$$

the left-hand side of which depends only on the computer and the right-hand side only on the algorithm. In general the operations' counts s and q are non-linear functions of some quantity measuring the size of the problem being solved: for example the dimension, n, of the matrices in a matrix problem. The equal performance line (19) can then easily be drawn on the $(n_{1/2},n)$ phase plane, because $n_{1/2}$ is always an explicit function of n, albeit a non-linear one. The phase plane can thereby be divided into regions in which each algorithm has the better performance. Sometimes it may be desirable from the graphical point of view to scale the axes and plot, for example, the $(n_{1/2}/n,n)$ or $(n_{1/2}/n^2,n)$ phase plane. It is a useful convention, however, always to choose the x-axis proportional to $n_{1/2}$, the apparent parallelism of the computer. In this way serial computer algorithms always appear to the left of the diagram, and parallel computer algorithms to the right.

Poisson's Equation

In this section we apply the method of analysis developed in section III to the selection of the best member of a family of direct methods for the solution of the model Poisson problem. The problem is the solution of the 5-point difference approximation to Poisson's equation on a square nxn finite difference mesh with simple boundary conditions (either given value, gradient or periodicity). Such a problem may seem artificially simple and of little practical importance, however history has shown that there are many important problems in physics (plasma, astro-, and dense matter), electrical engineering (semiconductor device simulation) and meteorology that require especially rapid methods for solving this problem (see, for example, Potter [4]; Hockney and Eastwood [5]).

The method to be analysed is direct, and involves the optimum combination of Fourier analysis in the x-direction and block cyclic reduction by lines in the y-direction. The method is known as the FACR(ℓ) algorithm, where ℓ is the number of stages of line cyclic reduction that are performed before Fourier analysis takes place. It represents a family of algorithms because the parameter ℓ can be used to minimise the time of execution. The first algorithm in this family, FACR(1), was published in 1965 by Hockney [6] working in collaboration with Golub. Subsequently the optimum value of $\ell(\approx\log_2\log_2 n)$ for serial computers was discovered empirically by Hockney [7], and the asymptotic form given later by Swarztrauber [8].

On parallel computers, it is interesting that the optimum value of ℓ depends not only on the size of the problem, n, as it does on a serial computer, but also on the parallelism of the computer as measured by its half-performance length, $n_{1/2}$. Hockney and Jesshope [1] have given the analysis for one way of implementing the FACR algorithm on a parallel computer which is most suitable for low levels of parallelism (the SERIFACR algorithm). Here we extend the previous work to a way of implementation that maximises the parallelism (i.e. vector length) of the algorithm and is most suitable for highly parallel computers (the PARAFACR algorithm). The reader is referred to the above book for a derivation of the operations' counts for Fourier analysis and cyclic reduction. We will quote these here and concentrate on the problem of finding the optimum value of ℓ.

SERIFACR Algorithm

The FACR algorithm involves five stages, and the variables that are related in each stage are shown for the FACR(1) algorithm in Fig. 4. The stages are:

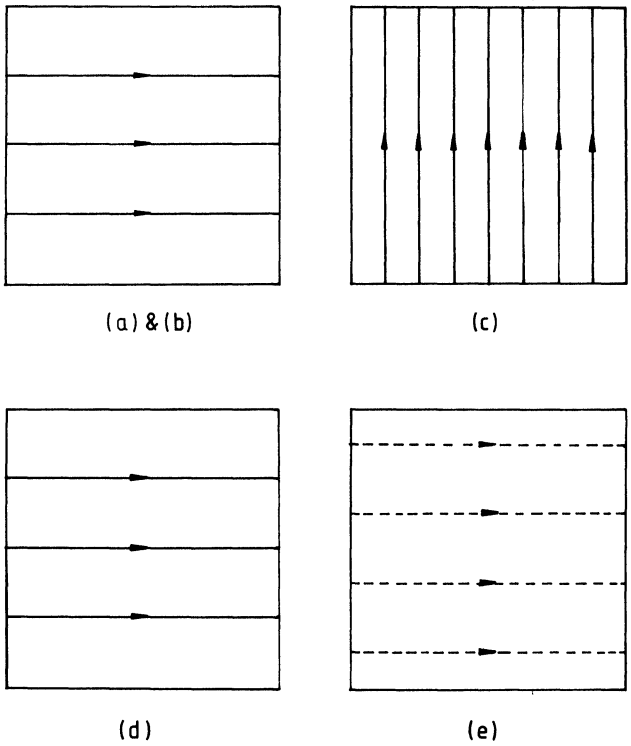


Fig. 4. Data relationships in the FACR(1) algorithm. The arrowed lines join variables related by equations or an FFT during different stages of the algorithm. (From Hockney and Jesshope 1981, courtesy of Adam Hilger).

(a) Modify RHS - block cyclic reduction by lines means the modification of the right-hand side of the Poisson equation on $n2^{-r}$ lines, where $r=1,2,\ldots,\ell$. Vectors are run in the vertical direction,and are composed of corresponding variables in each of the $n2^{-r}$ lines. The vector length is therefore $n2^{-r}$. The number of parallel operations is $(3\times2^{r-1}+2)n$, thus the time for this stage of the algorithm is proportional to

$$t_a = \sum_{r=1}^{r=\ell} (n_{1/2}+n2^{-r})(3\times2^{r-1}+2)n \qquad (21)$$

the factor r_∞^{-1} is omitted in the above because, as was seen in section III, it cancels out in any comparison of different algorithms on the same computer.

(b) Fourier analysis - is performed on $n2^{-\ell}$ lines in parallel. Vectors are of length $n2^{-\ell}$ are run vertically across the lines. The transforms are real and of length n and can be per-

formed by the fast Fourier transform (FFT) in $2\frac{1}{2}n\log_2 n$ vector operations of length $n2^{-\ell}$, hence

$$t_b = (n_{\frac{1}{2}}+n2^{-\ell})2\tfrac{1}{2}n\log_2 n \qquad (22)$$

(c) Solve harmonic equations - n tridiagonal equations, each of length $n2^{-\ell}$, are solved for the n harmonic amplitudes. The vectors now run horizontally and are of length n. The tridiagonal systems only involve variables from the last lines modified in stage (a) and Fourier transformed in stage (b). The time of execution is proportional to

$$t_c = 5(n_{\frac{1}{2}}+n)n2^{-\ell} \qquad (23)$$

the coefficient five is appropriate for solution by Gauss elimination, taking into account that the immediate sub- and super-diagonals of the tridiagonal matrices are unity, and that the main diagonal is a constant.

(d) Fourier synthesis - on the same lines as stage (b) gives the solution on these lines. The FFT is used in the same way as in stage (b) giving

$$t_d = (n_{\frac{1}{2}}+n2^{-\ell})2\tfrac{1}{2}n\log_2 n \qquad (24)$$

(e) Filling in - having found the solution on every 2^{ℓ} line in stage (d), fill-in takes place recursively. Each level, r, requires the formation of a right-hand side (2 operations) and the successive solution of 2^{r-1} tridiagonal systems. Vectors run vertically as in stage (a), and the time is proportional to

$$t_e = \sum_{r=1}^{\ell}(n_{\frac{1}{2}}+n2^{-r})(5\times 2^{r-1}+2)n \qquad (25)$$

Evaluating the sums in Eqns. (21) to (25) we find the total time of execution per mesh point to be proportional to

$$n^{-2}t_{SERIFACR} = s+\left[\frac{n_{\frac{1}{2}}}{n}\right]q' \qquad (26)$$

where

$$s = 4\ell+4+(1+5\log_2 n)2^{-\ell}$$

$$q' = 4\ell-8+8\times 2^{\ell}+5\times 2^{-\ell}+5\log_2 n$$

The equal performance line between the algorithm with ℓ levels of reduction and that with $\ell+1$ is easily found to be given by

$$\left[\frac{n_{\frac{1}{2}}}{n}\right] = \frac{(1+5\log_2 n)2^{-(\ell+1)}-4}{4+8\times 2^{\ell}-5\times 2^{-(\ell+1)}} \qquad (27)$$

The form of Eqn. (27) suggests that a suitable parameter plane for the analysis of SERIFACR is the $(n_{\frac{1}{2}}/n,n)$ phase plane, and this is shown in Fig.5. The equal performance lines given by Eqn. (27) divide the plane into regions in which ℓ=0,1,2,3 are the optimum choices. Lines of constant value of $n_{\frac{1}{2}}$ in this plane lie at 45 degrees to the axes, and the lines for $n_{\frac{1}{2}}$=16,128,2048 are shown dotted in Fig. 5. These lines are considered typical for the behaviour, respectively, of the CRAY-1, CYBER 205, and the average performance of the ICL DAP. For practical mesh sizes (say n<500) we would expect to use ℓ=1 or 2 on the CRAY-1, ℓ=0 or 1 on the CYBER 205, and ℓ=0 on the ICL DAP. The lower of the two values for ℓ applies to problems with n<100. Temperton [9] has timed a SERIFACR(ℓ) program on the CRAY-1 and measured the optimum value of ℓ=1 for n=32, 64 and 128. This agrees with our figure except for ℓ=128 where Fig. 5 predicts ℓ=2 as optimal. This discrepancy is probably because Temperton uses the Buneman form of cyclic reduction (see Hockney [7]) which increases the computational cost of cyclic reduction and tends to move the optimum value of ℓ to smaller values.

For a given problem size (value of n) Fig. 5 shows more serial computers (smaller $n_{\frac{1}{2}}$) to the left and more parallel computers (larger $n_{\frac{1}{2}}$) to the right. We see therefore that the more parallel the computer, the smaller is the optimum value of ℓ.

In the SERIFACR algorithm the vectors are laid out along one or other side of the mesh and never exceed a vector length of n. It is an algorithm suited to computers that perform well on such vectors, i.e. those that have $n_{\frac{1}{2}}$<n, and/or which have a natural parallelism (or vector length) which matches n. The latter statement refers to the fact that some computers (e.g. CRAY-1) have vector registers capable of holding vectors of a certain length (64 elements in the CRAY-1). There is then an advantage in using an algorithm that has vectors of this length and therefore fits the hardware design of the computer. For example, the SERIFACR algorithm would be particularly well suited for solving a 64×64 Poisson problem on the CRAY-1 using vectors of maximum length 64; particularly as this machine is working at better than 80 percent of its maximum performance for vectors of this length. On other computers, such as the CYBER 205, there are no vector registers and $n_{\frac{1}{2}}\approx$100. For these machines it is

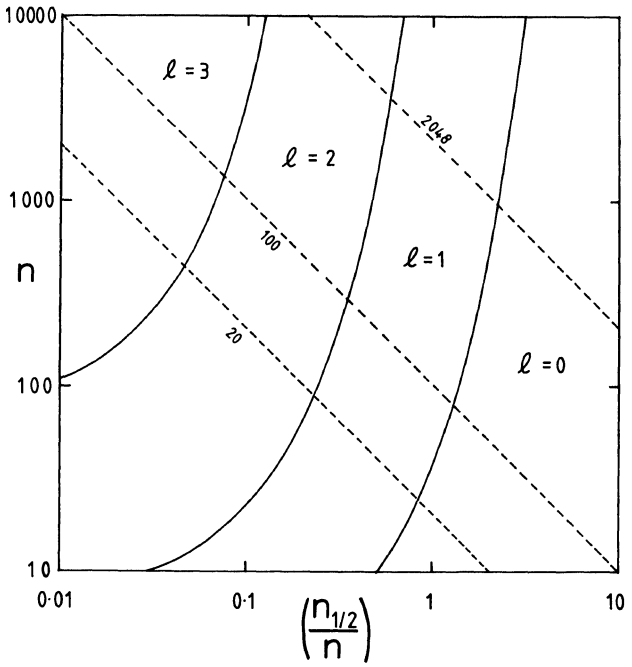


Fig. 5. The $(n_{1/2}/n,n)$ parameter plane for the SERIFACR(ℓ) algorithm. The solid lines delineate regions where the stated values of ℓ lead to the minimum execution time. The dotted lines are lines of constant $n_{1/2}$ corresponding to the CRAY-1 (=20), CYBER 205 (=100) and the average performance of the ICL DAP (=2048).

desirable to increase the vector length as much as possible, preferably to thousands of elements. This means implementing the FACR algorithm in such a way that the parallelism is proportional to n^2 rather than n. That is to say the vectors are matched to the size of the whole two-dimensional mesh, rather than to one of its sides. The PARAFACR algorithm that we now describe is designed to do this.

PARAFACR Algorithm

Each of the stages of the FACR algorithm can be implemented with vector lengths proportional to n^2:

(a) Modify RHS - at each level, r, of cyclic reduction the modification of the right-hand side can be done in parallel on all the $n^2 2^{-r}$ mesh points that are involved. Hence the timing formula becomes

$$t_a = \sum_{r=1}^{\ell} (n_{1/2}+n^2 2^{-r})(3\times 2^{r-1}+2) \qquad (28)$$

(b) Fourier analysis - The $n2^{-\ell}$ transforms of length n are performed in parallel as in SERIFACR, but now we use a parallel algorithm, PARAFT, for performing the FFT with a vector length of n. The vector length for all lines becomes $n^2 2^{-\ell}$ and the timing equation is

$$t_b = (n_{1/2}+n^2 2^{-\ell})4\log_2 n \qquad (29)$$

The factor 4 replaces the $2\frac{1}{2}$ in Eqn. (22) because extra operations are introduced in order to keep the vector length as high as possible in the PARAFT algorithm (see Hockney and Jesshope [1], page 315). We also note that the factor n has moved inside the parentheses in comparing Eqn. (22) with (29), because the vector length has increased from $n2^{-\ell}$ to $n^2 2^{-\ell}$.

(c) Solve harmonic equations - the harmonic equations are solved in parallel as in SERIFACR, but we use a parallel form of scalar cyclic reduction, PARACR, instead of Gauss elimination for the solution of the tridiagonal systems (see Hockney and Jesshope [1], page 289). For the special case of the coefficients previously noted, there are 3 parallel operations at each of $\log_2 n$ levels of scalar cyclic reduction. The vector length is $n^2 2^{-\ell}$ giving

$$t_c = (n_{1/2}+n^2 2^{-\ell})3\log_2 n \qquad (30)$$

(d) Fourier synthesis - as stage (b)

$$t_d = (n_{1/2}+n^2 2^{-\ell})4\log_2 n \qquad (31)$$

(e) Filling in - at each level, r, $n2^{-r}$ tridiagonal systems of length n are to be solved. Using PARACR as in stage (c) the vector length is $n^2 2^{-r}$. Afterwards a further two operations are required per point which may also be done in parallel giving

$$t_e = \sum_{r=1}^{\ell} (n_{1/2}+n^2 2^{-r})(3\times 2^{r-1}\log_2 n+2) \qquad (32)$$

The time per mesh point for the PARAFACR algorithm is therefore proportional to

$$n^{-2}t_{PARAFACR} = s+\left\lceil \frac{n_{1/2}}{n^2} \right\rceil q'' \qquad (33)$$

where

$$s = \tfrac{1}{2}(3\log_2 n+1)\ell+4+(11\log_2 n-4)2^{-\ell}$$

$$q'' = 4\ell+(3\log_2 n+1)(2^{\ell}-1)+11\log_2 n$$

The equal performance line between the level ℓ and $\ell+1$ algorithms is given by

$$\left[\frac{n_{1/2}}{n^2}\right] = \frac{(11\log_2 n-4)2^{-(\ell+1)}-\tfrac{1}{2}(3\log_2 n+1)}{4+(3\log_2 n+1)2^{\ell}} \qquad (34)$$

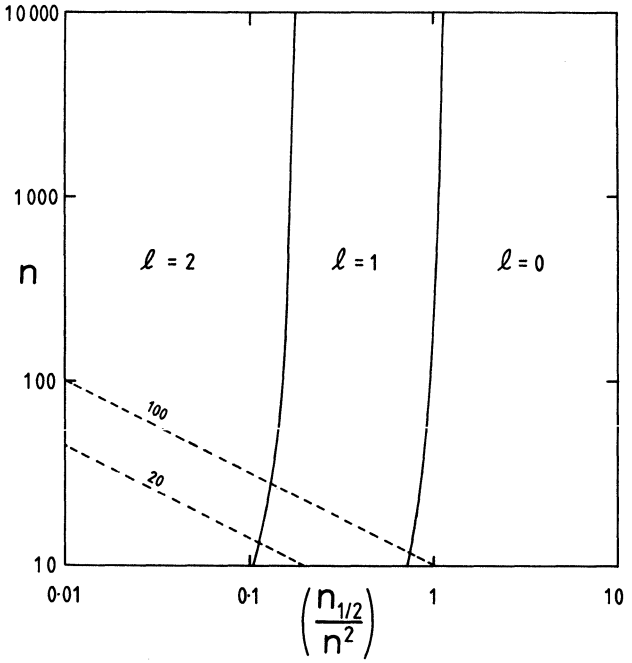


Fig. 6. The $(n_{1/2}/n^2,n)$ parameter plane for the PARAFACR(ℓ) algorithm. Notation as in Fig. 5.

The form of Eqn. (34) leads us to choose to plot the results for the PARAFACR algorithm on the $(n_{1/2}/n^2,n)$ parameter plane, and this is done in Fig. 6. We find that the equal performance lines are approximately vertical in this plane, and conclude that $\ell=2$ is optimal for $n_{1/2}<0.1n^2$, $\ell=1$ for $0.1n^2<n_{1/2}<n^2$ and, $\ell=0$ for $n_{1/2}>n^2$. There are no circumstances when more than two levels of reduction are worth while, thus justifying our use of the unstabilised FACR algorithm (see Hockney and Jesshope [1],page 348). In particular, for a processor array with as many or more processors than mesh points ($N>n^2$), we take $n_{1/2}=\infty$ and find $\ell=0$. This case corresponds to the solution of a 64×64 problem on the ICL DAP which is an array of 64×64 processors. The dotted line for $n_{1/2}=100$ is shown in Fig. 6, corresponding to the CYBER 205. For all but the smallest meshes (i.e. for $n>30$) we find $l=2$ optimal. The line for $n_{1/2}=20$ is also given, from which we conclude that $\ell=2$ is optimal in all circumstances if this algorithm is used on the CRAY-1.

SERIFACR/PARAFACR Comparison

So far we have considered the choice of the best value of ℓ for each algorithm. Having optimised each algorithm we now consider which is the best algorithm to use. This is done by plotting $t_{SERIFACR}$ and $t_{PARAFACR}$ against $(n_{1/2}/n)$ for a series of values of n, in order to determine approximately which algorithms abut each other in different parts of the parameter plane. One can then calculate the equal performance line between PARAFACR(ℓ) and SERIFACR(ℓ') from

$$\left[\frac{n_{1/2}}{n}\right] = \frac{a-b}{c-d} \qquad (35)$$

where

$$a = \tfrac{1}{2}(3\log_2 n+1)\ell+4+(11\log_2 n-4)2^{-\ell}$$

$$b = 4\ell'+4+(1+5\log_2 n)2^{-\ell'}$$

$$c = 4\ell'-8+8\times 2^{\ell'}+5\times 2^{-\ell'}+5\log_2 n$$

$$d = \left[\frac{4\ell+(3\log_2 n+1)(2^{\ell}-1)+11\log_2 n}{n}\right]$$

The interaction of the two algorithms is shown in Fig. 7 on the $(n_{1/2}/n,n)$ parameter plane. This division between the two algorithms is about vertical in this plane showing that SERIFACR is the best algorithm for smaller $n_{1/2}<0.4n$ (the more serial computers), and the PARAFACR is the best for larger $n_{1/2}>0.4n$ (the more parallel computers). Lines of constant $n_{1/2}$ are shown for the CRAY-1 and CYBER 205. We conclude that SERIFACR should be used on the CRAY-1 except for small meshes with $n<64$ when PARAFACR(2) is likely to be better. On the CYBER 205, PARAFACR is preferred except for very large meshes when SERIFACR(2) ($300<n<1500$) or SERIFACR(1) ($n>1500$) is better.

Conclusions

The optimum choice of algorithm for the solution of Poisson's equation on a

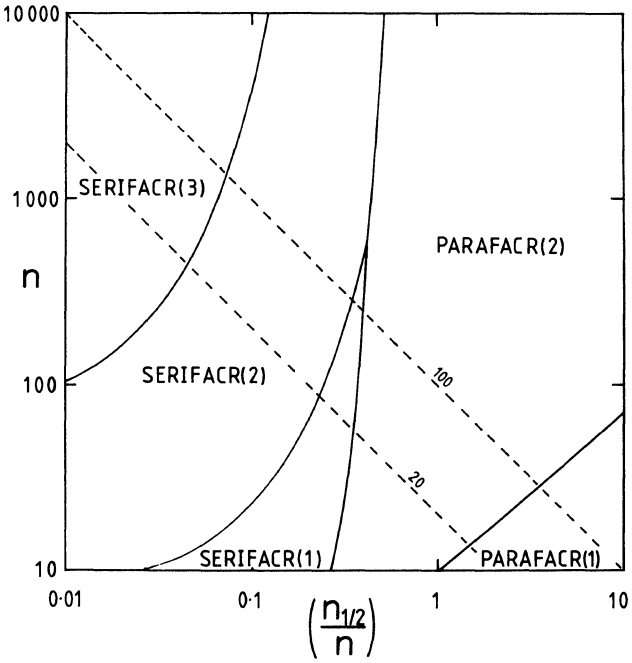


Fig. 7. Comparison between the SERIFACR(ℓ) and PARAFACR(ℓ), showing the regions of the ($n_{1/2}/n$,n) parameter plane where each has the minimum execution time.

parallel computer is found to depend on the ratio of the parallelism of the computer (as measured by its half-performance length) to the size of the finite difference mesh. Two implementations of the FACR(ℓ) algorithm have been considered, and in both cases we conclude that less cyclic reduction (lower ℓ) should be performed the more parallel is the computer. We find that the implementation with the smallest vector length (or algorithmic parallelism) SERIFACR, is most suitable for computers with low hardware parallelism (i.e. the more serial with low $n_{1/2}$), and that the implementation with the longest vector length PARAFACR, is most suitable for computers with high hardware parallelism.

The above conclusions are based on the simplifying assumptions given in the introduction. The best practice is to write a program for both algorithms with variable ℓ, and determine empirically the optimum algorithm and level of reduction. Our graphs can be a guide.

Acknowledgements

The author wishes to thank Chris Jesshope, Jim Craigie and Edward Detyna for help in clarifying the ideas in this paper, and Knut Mørken for pointing out several misprints in the original manuscript.

References

[1] R. W. Hockney and C. R. Jesshope, Parallel Computers - Architecture, Programming and Algorithms, Bristol: Adam Hilger, (1981). (Distributed in North and South America by Heyden & Son Inc., Philadelphia).

[2] D. A. Calahan and W. G. Ames "Vector Processors: Models and Applications", IEEE Trans. Circuits and Systems, vol. CAS-26, (1979), pp. 715-726.

[3] D. J. Kuck, Computers and Computations, New York: Wiley, (1978).

[4] D. Potter, Computational Physics, London: Wiley, (1973).

[5] R. W. Hockney and J. W Eastwood, Computer Simulation Using Particles, New York: McGraw-Hill, (1981).

[6] R. W. Hockney "A fast direct solution of Poisson's equation using Fourier analysis", J. Assoc. Comput. Mach., vol. 12, (1965), pp. 95-113.

[7] R. W. Hockney, "The potential calculation and some applications", Methods Comput. Phys., vol. 9, (1970), pp. 135-211.

[8] P. N. Swarztrauber, "The methods of cyclic reduction, Fourier analysis and the FACR algorithm for the discrete solution of Poisson's equation on a rectangle", SIAM Rev., vol. 19, (1977), pp. 490-501.

[9] C. Temperton, "Fast Fourier transforms on the CRAY-1", Infotech State of the ART Report: Supercomputers, Vol 2 Eds C. R. Jesshope and R. W. Hockney, Maidenhead: Infotech Intl., (1977), pp. 359-379.

PARALLEL POISSON AND BIHARMONIC SOLVERS IMPLEMENTED ON THE EGPA MULTIPROCESSOR

Marián Vajteršic
Institute of Technical Cybernetics
Slovak Academy of Sciences
Bratislava, Czechoslovakia

Abstract

In this paper the use of the EGPA (Erlangen General Purpose Array) computer system of the MIMD (Multiple Instruction - Multiple Data) mode of parallelism for solving the model elliptic boundary value problems of the second and fourth orders is presented. A direct method for solving the Poisson equation and a semidirect biharmonic solver are structured for parallel execution on this hierarchical multiprocessor system. Both the computational and intercommunication requirements of the system are taken into account in order to minimize the transfer and synchronization steps required. Both algorithms considered have been actually run on the EGPA parallel computer with considerable speed - ups in comparison to sequential execution.

Introduction

From the advent of parallel computers, attention has been paid to designing efficient algorithms for the numerical solution of the boundary value problems for elliptic partial diferential equations. Among them parallel algorithms for solving Poisson and biharmonic equations are frequently discussed in studies concerning parallel numerical algorithms [e.g. 1,2]. Especially, parallel Poisson and biharmonic solvers for SIMD (Single Instruction - Multiple Data) machines are well developed [3]. However, in most of these algorithms the number of processors required is equated to the number N^2 of discrete solution values in the domain. Despite the fact that there exists a machine with large number of processors [4] and the next one [5] is being developed, on parallel systems currently available it is often not possible to meet this demand for solving large-scale ($N \geq 128$) problems arising in practice. The most recent comparison of actual performance of processor arrays ICL DAP and Borroughs BSP, as well as of three pipeline machines, on Poisson solving has appeared in [6].

On MIMD systems currently in operation the number of processors is not greater than 50 [7] and therefore the algorithms should be modified for practical computation. Some iterative methods of Jacobi type have been tested on the Cmmp and Cm* systems [8,9]. The mesh points were divided into portions, each of them assigned to one process. The experiments have shown the best performance for the purely asynchronous method where the iteration values are evaluated without any need for synchronization. However, most of the Poisson solvers are synchronous and therefore significantly more difficult to implement efficiently. The per-

This work was pursued during author's stay at IMMD, Erlangen - Nürnberg University, in 1981, under the research fellowship of the Alexander von Humboldt Foundation.

0190-3918/82/0000/0072$00.75 © 1982 IEEE

formance is strongly influenced by synchronization of the process and by transferring the intermediate results between the processors.

We have implemented two synchronous algorithms on the EGPA system [10], which is one of the operating MIMD computers in the world. The current configuration of this hierarchically organized multiprocessor represents an elementary pyramide. The aim in developing the system was to assure a high rate of flexibility of operating modes. Some characteristics of the system are described in the following section.

In the third section description is given of the two algorithms selected for implementation. The Poisson equation was solved by the direct method using the decomposition property of the matrix which results from the discretization of the problem [11]. The biharmonic problem solved by the semidirect method [12] based on splitting the fourth-order elliptic operator into two operators of the second order is the second one under consideration. The algorithm proceeds iteratively where solutions of the two Poisson equations are to be computed in one iteration.

These fast sequential algorithms have been tailored to the EGPA considering the connections between the processors. The allocation of the sub-tasks to the processors affects the ammount of information transferred and the synchronization steps required. This strategy is described in the fourth section of this paper.

The execution results are summarized in the final section. The efficiency of the perfomance defined there is problem dependent and for sufficiently large discretization parameters N the efficiencies are high enough to support the conclusion that EGPA is a viable alternative to sequential computer for the numerical solution of the elliptic equations considered.

Characteristics of the EGPA system

The multiprocessor EGPA has been developed and run at Erlangen University. It reflects the idea formulated by Händler, Hofmann and Schneider in [13]. Its essential features are an extensible hierarchical structure composed of pyramidal cells and more operating modes using, as far as posible, commercially available hardware. The set of operating modes involves array-processing, associative computation, data-flow approach, micropipelining, multiprocessing as well as sequential computation. On the Figure 1 a model of the 3-level EGPA-pyramide [14] is demonstrated, where one circle corresponds to a processor and its associated memory. The arrows illustrate unidirectional connection between a processor and the memory of its neighbour while the other lines represent bidirectional connections.

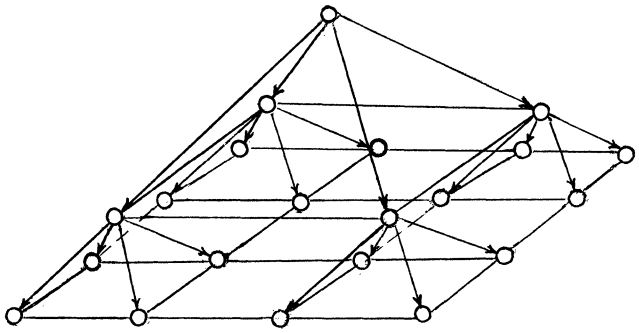


Figure 1.

The current configuration consists of five processors which form a pyramide with four slave processors in the base (further A processors). The one processor in the head of the pyramide (B processor in the following text) is either controlling a computational process or

computing a sub-task of the problem like the processors in the base.

The processors are 32-bit control computers of the AEG 80-60 type. Each processor is microprogrammable and can be used for associative processing with additional hardware. The experiments with vertical processing [14] show promising acceleration factors.

The connections between processors are realized through the multiport memory. The processors of the array are neighbour connected in both directions. The processor in the top can access the memories of all the slave processors while the access in the opposite direction does not exist.

Due to the interconnections the data transfer between the diagonal processors is a bottleneck. In numerical applications it is often the case that intermediate results should be available to each processor after a synchronization step. One way how to overcome it lies in transporting the data via the left or right neighbours, as described in the fourth section of this paper. However, each transport of evaluated results is time consuming and therefore the cooperation of several processors executing one task is only useful if each processor works on large independent data quantities. For more detailed information concerning the EGPA system we refer to [10].

Before putting down the parallel program version of an algorithm on EGPA, the program for the sequential execution is written in the ALGOL-68-like programming language SL3. The program for parallel execution where each processor executes an independent process is then rewritten from the sequential program using special instruction set of the EGPA-Monitor [15]. Generally, a special programming module should be formulated for each processor. Such module contains a set of procedures to be executed as well as declarations of variables. The variables are of local and global types. The variables of the former type are local in the main program or in one of the processors executed while the latter ones are used by more processors and are located in special memory segments. If the processes executed on all A processors are the same, as it has been the case in our applications, only one program module for all processors has to be written. However for each process on the A processors, one special module is to be formulated to create control segments where the corresponding global variables can be declared. In order to identify the processors if the identical processes are run on all processors, there is a variable for actual processor number in the program module to be declared.

The main program for the B processor encompasses procedures for initialisation, execution, synchronization and termination of the whole computational process.

The Poisson and Biharmonic Solvers

The Dirichlet problem for Poisson equation in two dimensions for an unknown function u and for given functions f and g

$$\begin{aligned} u_{xx} + u_{yy} &= f \quad \text{in } R \\ u &= g \quad \text{on } \bar{R} \end{aligned} \qquad (1)$$

on a unit square R with the boundary $\bar{R}$ is considered. After discretizing the domain in both directions by a step of size $(N+1)^{-1}$ for an integer N, the second order derivatives in (1) can be approximated by finite difference formulae. The resulting sparse linear system of equations is

$$Mu = w \qquad (2)$$

where the vector u contains N^2 values of u to be evaluted in the interior grid points. The matrix M is of known block-tridiagonal structure M = (-I,T,-I) with tridiagonal blocks T = (-1,4,-1) and with identity matrices I of order N.

The algorithm of matrix decomposition [11] is adopted to solve (2) making advantage of the property

$$T = VDV^T$$

where

$$V \equiv (v_{ij}) = \sqrt{\frac{2}{N+1}} \sin \frac{ij\pi}{N+1}, \qquad i,j = 1,\ldots,N$$

and

$$D = \text{diag}\,(d_1,d_2,\ldots,d_N) \text{ with } d_i = 4 - 2\cos\frac{i\pi}{N+1}, \qquad i = 1,\ldots,N$$

are, respectively, the eigenvector and eigenvalue matrices of T. Defining the matrices

$$T_i = (-1,d_i,-1) \qquad i = 1,\ldots,N,$$

the steps of the algorithm are as follows:

P0. Evaluate the right-hand side vector w of (2).

P1. Compute VW = Y,
where W is a matrix representation of w.

P2. Solve for i = 1,...,N
$$T_i\bar{u}_i = \bar{y}_i \qquad (3)$$
where $\bar{y}_i$ is i-th row of Y and
$$T_i = (-1,d_i,-1), \qquad i = 1,\ldots,N \qquad (4)$$
are of order N.

P3. Compute $V\bar{U} = U$
where the i-th row of the matrix $\bar{U}$ is $\bar{u}_i$ transposed for i = 1,...,N.

A fast algorithm for solving the biharmonic equation

$$u_{xxxx} + 2u_{xxyy} + u_{yyyy} = f \quad \text{in } R,$$
the boundary conditions being (5)
$$u = g_1, \quad u_n = g_2, \quad \text{on } \bar{R}$$
is proposed in [12]. One iteration of the algorithm proceeds by

$$u^{(m+1)} = [2\omega I - 2(1-\omega)^2 M^{-1}(M^{-1}H)]u^{(m)} - \omega^2 u^{(m-1)} + d \qquad (6)$$

where the N^2 order matrix H is a sparse diagonal one of the form
$H = \text{diag}(I+H_0,H_0,\ldots,H_0,I+H_0)$, with $H_0 = (1,0,\ldots,0,1)$. The vector d is computed from constant vectors b and e which result from the discretization of the given functions of (5). The parameter ω is necessary for ensuring the convergence of the process.

The iterations (6) are computed until

$$\max_{i,j=1,\ldots,N} \left| U_{ij}^{(m+1)} - U_{ij}^{(m)} \right| < \varepsilon \qquad (7)$$

becomes valid for a prescribed accuracy ε. The iterative solution u of (5) is then obtained by

$$u = Fu^{(m+1)} \qquad (8)$$

where $u^{(m+1)}$ results from (7) and the blockdiagonal matrix F = diag(V,V,...,V) is of order N^2.

The algorithm proceeds in the following steps:

m ← 1

B0. Evaluate the constant vectors b and e and compute
$d = FM^{-1}b + FM^{-1}(M^{-1}e)$.
Set $u^{(0)} = 0$, $u^{(1)} = M^{-1}b$.

LAB: m ← m + 1

B1. Calculate $w^{(m)} = HFu^{(m)}$ taking advantage of the sparsity of H.

B2. Compute $w = Fw^{(m)}$ from the sparse vector $w^{(m)}$ and order it into a matrix W.

B3. Solve $T_i\bar{y}_i = \bar{w}_i$ where $\bar{w}_i$ is i-th row of W and T_i is as

defined in (4).

B4. Solve $T_i\bar{u}_i = \bar{y}_i$ for $\bar{u}_i$, $i = 1,2,\ldots,N$ which are components of the vector $M^{-1}(M^{-1}H)u^{(m)}$ from (6).

B5. Evaluate $u^{(m+1)}$ by (6) and execute the test (7).

B6. Repeat from LAB if (7) is not valid.

B7. Evaluate u by (8).

Implementation of the algorithms

To execute both the algorithms on EGPA we have tried to split up the computational task into four portions among the four processors only. The role of the B processor is to supervise the process and to realize the output of evaluated results. For this reason it will be further supposed that N is a multiple of 4 . For explanation purposes let us distinguish the processors on the A level by AJ, J = 1,2,3,4 as shown in the Figure 2.

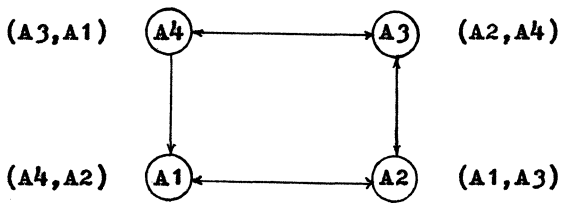


Figure 2.

The left and right neighbours of each processor AJ defined by AJL and AJR respectively are given in parentheses. The first term corresponds to the AJL while the second one to the AJR for J = 1,...,4. To identify values of a global variable x, we shall respectively write x(J), x(JL) and x(JR) according to the definition of processors AJ, AJL and AJR. Further, for a more convenient formulation of the algorithms, some processing functions are introduced.

For the synchronization, use is made of Wait (C), where the condition expressed by C must become valid before proceeding to the next step of the algorithm. In practice, the processor is asking during its active waiting state for response to whether C is being fulfilled or not. For the transport phase realized by processor AJ, where a quarter of array X located on the processor AJR is transported into the memory of AJL, Trans (X_{JR}) is used. If the value of a variable x is to be transported in the same way, the statement Trans (x) is used analogously.

In the program formulated for processor AJ, Execute (S) will denote execution of the whole task of step S. In contrast, by $\text{Execute}_J(S)$ only the J-th quarter (we recall J = 1,...,4) of the task in step S will be executed by the processor AJ.

Turning back to the algorithm for Poisson equation, one can observe that steps P1 and P2 are independently solvable on all four processors. Indeed, for N being a multiple of 4, the matrix product of step P1 can be performed in four subtasks partitioning the matrix V horizontally into four rectangular matrices V_J, J = 1,...,4, having the same type. However, before calculating the matrix products $V_JW = Y_J$ on four processors concurrently, the constant matrix W should be evaluated by each processor completely. Without any communication between processors, in step P2 the rows of Y_J for each processor AJ are only needed for evaluation of $\bar{U}_J$ which is rowwise structured from N/4 solutions $\bar{u}_i$ of (3), where i = (J-1)N + k, k=1,...,N.

In order to compute the matrix multiplication in step P3 in parallel, the complete matrix $\bar{U}$ must be available to each processor. Hence, after step P2 synchronization and transport of quarters $\bar{U}_J$ of $\bar{U}$ between diagonally positio-

ned processors should follow. The synchronization of the process is realized by setting the value of the variable berfertig on processor AJ on true after this processor has finished its job on steps P1 and P2. Each processor AJ is waiting actively until its right neighbour is ready in order to be able to transport the array $\bar{U}_{JR}$ from AJR into the memory of AJL. If this transport is realized the value for the variable transfertig on AJL is reset on true. However, it may occur that the left neighbour AJL has not yet finished the execution of steps P1 and P2 or even the processor AJ has not obtained the portion of $\bar{U}$ which should be transported to it by AJR from its diagonally located processor.

In both cases it is necessary to wait in order to realize the transport phase of the algorithm completely. Then under the same principle as in step P1, step P3 can be calculated in parallel. Processor B realizes output of the resulting matrix U in the finishing stage of the EGPA Poisson Solver (EPS) algorithm which can be formulated for parallel execution on four processors AJ, J = 1,...,4 from the viewpoint of one processor AJ as follows:

```
    berfertig ←— false
    transfertig ←— false
  Execute (P0);
  Execute_J(P1,P2);
    berfertig (J) ←— true
  Wait (berfertig (JR) ←— true);
  Trans (Ū_JR);
    transfertig (JL) ←— true
  Wait (berfertig (JL) ←— true);
  Wait (transfertig (J) ←— true);
  Execute_J(P3).
```

The algorithm for the biharmonic equation is of an iterative structure and is therefore more demanding on synchronization. In the preprocessing phase the right-hand side vectors b and c are to be evaluated on each processor. According to the previous explanation of the EPS algorithm, the evaluation of the vector d can follow in parallel whereby each processor evaluates the corresponding quarter of the vector.

Attention is focused to the iterative section of the algorithm which is crucial from the viewpoint of implementation efficiency. We introduce a global variable m where the number of currently evaluated iteration is being stored. The first synchronization phase is in the beginning of each iteration in order to start the new iteration after each processor has finished the evaluation of preceding iteration values. It is realized by waiting AJ on AJR, i.e. until m(J) = m(JR) and by transport of values of m after which the processor AJ is waiting until m(J) = mdiag(J) and m(J) = m(JL). Here, mdiag(J) is the value of the variable m transported from the diagonally located processor to the processor AJ. The computation starts with the evaluation of the complete "window" $HFu^{(m)}$ on each processor in order that parallel multiplication by V_J be executable. The twofold elimination phase is to be performed analogously as in the previous algorithm. Having stored its portion of preceding iteration values, each processor can also calculate independently corresponding new iteration values $u^{(m+1)}$ by (6).

Since each processor AJ can evaluate the maximum value max of differences of two successive iteration values in its corresponding quarter of grid points only, there is a problem how to decide on the termination of the process (6) by (7). For this reason a global boolean variable iter has been defined and initialized on each processor by false in

the beginning of the iteration. Its value is reset on true on those processors where max is greater than a given ε. After the evaluation of max on processor AJ the value berfertig (J) is increased by 1 and followed by synchronization and subsequently by the transport of new iteration values as well as of iter values between diagonal processors. The value of the synchronization variable transfertig is increased by 1 in the left neighbouring processor AJL.

The values of the iter variable from each processor being available to all of them, the decision from these values is made whether the iteration will continue or the final computation with output of results should follow. The final matrix multiplication is performed on A processors while the output of results is made through the B processor.

The programming scheme for parallel execution of one iteration of the EGPA Biharmonic Solver (EBS) algorithm for each processor AJ is the following:

```
LAB: m ← m + 1
     Wait (m(J) = m(JR));
     Trans (m(JR));
     Wait (m(J) = mdiag (J) and
     m(J) = m(JL));
        iter ← false
     Execute (B1);
     Execute_J(B2,B3,B4,B5);
        if max > ε  then
        iter (J) ← true
        berfertig (J) ← berfertig (J)+1
     Wait (berfertig (J) =
           = berfertig (JR));
     Trans (U_JR^(m));
     Trans (iter(JR));
        transfertig (JL) ← transfer-
        tig (JL)+1
     Wait (transfertig (JL) = transfer-
     tig (J) and berfertig (J) =
     = berfertig (JL));
        if iter (J) or iterdiag (J) or iter
        (JL) or iter (JR) then repeat from
        LAB.
```

Results and concluding remarks

The two algorithms of the previous section were executed on EGPA. Also the sequential versions described in third section were run on a single processor AEG 80-60. The speed-up for parallel implementation of the algorithm against the sequential one is defined by

$$s = \frac{t_s}{t_p}$$

where t_s is CPU time of the serial execution and t_p corresponds to the parallel one. The assumed efficiency of implementation is evalueted in terms of s by

$$e = \frac{s}{n} 100\ \%$$

where n is the number of processors participating in computational work. The time values given in Tables 1 and 2 are CPU times measured for the whole computational process not including the times for the output of results. Since both the algorithms EPS and EBS have been tailored to employ four A processors on arithmetical and transport operations, n = 4 is considered in the efficiency results. In spite of this the results for n = 5 due to the 5-processor configuration of EGPA are also given in parentheses.

The results for solving the Poisson equation for the function x^2+y^2 are summarized for various N in Table 1. The speed-up can be seen to increase in dependence on the problem size. It illustrates the influence of the synchronization and transport phase on the efficiency which increases apparently when the independently solvable arithmetical tasks become dominant for large N. Sin-

ce the steps of the algorithm are highly parallel there arises the question why the speed-up ratio does not tend more closely to the ideal value of 4. The main reason is that all the A processors evaluate the complete vector w simultaneously, i.e. the speed-up for this stage of the algorithm is 1 only. (Of course, it might be possible to divide the task of evaluating w into four processors but with additional costs for synchronization and transport of portions of w.)

N	t_s(in sec.)	t_p(in sec.)	s	e(in %)
32	6.959	4.837	1.44	36.0 (28.8)
40	12.988	6.494	2.00	50.0 (40.0)
64	49.834	16.449	3.03	75.8 (60.6)
80	94.852	28.610	3.32	82.9 (66.3)
128	374.624	103.237	3.63	90.7 (72.6)

Table 1.

The biharmonic equation (5) was solved for the unknown function x^3-3y^2+2xy for more values of ε on a variety of grids. The iteration parameters ω used are estimated as proposed in [16]. It can be seen from Table 2, where the number of iterations m corresponds to the accuracy $\varepsilon = 0.001$, that the results for the algorithm EBS are less dependent on parameter N than in the preceding algorithm. On the other hand, the efficiency results achieved are slightly worse than in the algorithm EPS even for large N. It is due to the iterative nature of the algorithm where two synchronization and transport phases within one iteration affect the speed-up considerably.

N	m	t_s(in sec.)	t_p(in sec.)	s	e (in %)
32	27	50.384	25.838	1.95	48.8 (39.0)
40	33	96.742	43.189	2.24	56.0 (44.8)
64	60	436.387	161.010	2.71	67.8 (54.2)
80	69	782.288	277.406	2.82	70.5 (56.4)
100	75	1 389.704	446.849	3.11	77.8 (62.2)

Table 2.

It should be investigated whether a five-processor realization would bring an improvement of s and e for both algorithms. It would also be interesting to know

whether the synchronization of the A processors performed through the B processor would yield better results or not. We note that there is also a chance to use the FFT routine instead of a classical matrix product procedure in the algorithm EPS. However, in such approach the performance strategy has to be modified because the transform by V instead of multiplication by V_J must be realized on each processor. Hence, the synchronization and transport should be performed twice compared to its being performed once in our strategy.

The results of experiments indicate that both algorithms can be efficiently implemented on the EGPA system. Some other experiments and analyses also support the view that EGPA, as a generally oriented multiprocessor, appears to be adequate for solving a broad variety of non-numerical as well as numerical problems.

Acknowledgements

The author is deeply indebted to Prof. W. Händler for his personal attention and support he gave to this research work. Also the collaboration with colleagues M. Gössman, W. Kleinöder, M. Rathke and H. Zischler is acknowledged.

References

[1] R.W. Hockney, "A Fast Direct Solution of Poisson's Equation Using Fourier Analysis", JACM 12, (1965), pp. 95-113.

[2] D. Heller, "A survey of Parallel Algorithms in Numerical Linear Algebra", Siam Review 20, (1978), pp. 740-777.

[3] A.H. Sameh, S.C. Chen, and D.J. Kuck, "Parallel Poisson and Biharmonic Solvers", Computing 17, (1976), pp. 219-230.

[4] P.M. Flanders, D.J. Hunt, S.F. Reddaway, and D. Parkinson, "Efficient High Speed Computing with the Distributed Array Processor", in High Speed Computer and Algorithm Organization, Academic Press, (1977), pp. 113-128.

[5] K.E. Batcher, "Design of a Massively Parallel Processor", IEEE Trans.Comp. C-29, (1980), pp. 836-840.

[6] R.W. Hockney, and C.R. Jesshope, Parallel Computers, Adam Hilger Ldt., (1981), 416 pp.

[7] R.J. Swan, S.H. Fuller, and D.P. Siewiorek, "Cm* - A Modular Multi-Microprocessor", Proc. AFIPS, (1977), pp. 637-644.

[8] G.M. Baudet, "Asynchronous Iterative Methods for Multiprocessors", JACM 25, (1978), pp. 226-244.

[9] S.H. Fuller, A.K. Jones, and I. Durham, "Carnegie-Mellon University Multi-Microprocessor Review (Cm*), Carnegie-Mellon Univ., Dept. of Comp. Science, AD - A 050135, (1980).

[10] U. Hercksen, R. Klar, and W. Kleinöder, "Hardware - measurements of storage access conflicts in the processing array EGPA", Proc. 7th Intern. Symp. on Computer Architecture, La Baule (1980), pp. 317-324.

[11] B.L. Buzbee, G.H. Golub, and C.W. Nielsen, "On direct methods for solving Poisson Equations", Siam J. Num. Analys. 7, (1970), pp. 627-656.

[12] M. Vajteršic, "A Fast Algorithm for Solving the First Biharmonic Boundary Value Problem", Computing 23, (1979), pp. 171-178.

[13] W. Händler, F. Hofmann, and H.J. Schneider, "A General Purpose Array with a Broad Spectrum of Applications", Informatik Fachberichte 4, Springer Verlag, (1976), pp. 311-334.

[14] A. Bode, "Vertical Processing: The emulation of associative and parallel behavior on conventional hardware", Euromicro, (1980), pp. 215-220.

[15] M. Rathke, Parallel - Schnittstelle, Einführung EGPA, PAR. EINF. Internal documentation file, EGPA Project, IMMD III, University of Erlangen - Nürnberg, (1980).

[16] L.W. Ehrlich, "Solving the biharmonic equation as coupled finite difference equations", SIAM J. Num. Analys. 8, (1971) pp. 278-287.

ITERATIVE ALGORITHMS FOR TRIDIAGONAL MATRICES ON A WSI-MULTIPROCESSOR+

D. D. Gajski,* A. H. Sameh,* and J. A. Wisniewski**

*Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801-2987

**Sandia National Laboratories
Division 2113
Albuquerque, New Mexico 87185

Abstract -- With the rapid advances in semiconductor technology, the construction of Wafer Scale Integration (WSI)-multiprocessors consisting of a large number of processors is now feasible. We illustrate the implementation of some basic linear algebra algorithms on such multiprocessors.

1. Introduction

The need has always existed for fast and efficient numerical computations that accurately and truly simulate physical processes. The VLSI technology increased that demand by making numerical computation affordable. This, in turn, further expanded their popularity and application. The computers used for this purpose usually come in three forms:

(a) High-speed parallel and/or pipelined machines. These are general-purpose, expensive machines (CRAY-1, CYBER 203/205, NASA's Numerical Aerodynamics Simulator, and S-1) optimized for high performance (usually using special packaging technology, cooling system, and on the leading edge of technology).

(b) Attached array processors. These are less-general, order-of-magnitude less costly machines with high-speed arithmetic, designed to perform well on certain applications (FPS AP series). They are attached to a host processor, which supplies only numerically-intensive special tasks to the attached processor. The attached processor is fully programmable but knowledge of its architecture is necessary if the machine is to be properly exploited.

(c) Special-purpose co-processors. These are special-purpose, low cost "black boxes" that can execute efficiently a few well-defined problems. They are minimally programmable and they are used as arithmetic accelerators (Intel's 8087), as special library subroutines (FFT, triangular solvers, filters).

In this paper we are interested in systems of the third type. Here, one or more special-purpose accelerators attached by shared bus to the low-cost, low-speed, general-purpose host can transform the host into a powerful number cruncher for a specific application. These special applications may be real time applications, or just some frequently used algorithms that we would like to speed up. The addition of one or more special-purpose accelerators, at $100-$200 each, may improve performance, resulting in performance-cost ratios that are not available with any other general-purpose system.

The WSI Model

VLSI technology has brought us the capability of increasing processor speed, size and complexity of several orders of magnitude. However, it needs a hierarchical and regular design, since the old von Neumann model is difficult to upgrade for the emerging WSI (Wafer Scale Integration) technology. This has influenced the search for new models of computation and machines. In addition to the constraints imposed by packaging and manufacturing technology, a WSI model must satisfy the following requirements:

(a) a few types of simple processor memory modules replicated throughout the wafer;

(b) regular communication network between modules with a constant number of crossovers;

(c) I/O ports on the boundary of the wafer;

(d) asynchronous communication among modules;

(e) I/O rate independent of the size of the problem; and

(f) high level of fault-tolerance.

The most popular VLSI model for numeric computations, the systolic array model introduced by H. T. Kung [KuLe79] satisfies our conditions (a),

+This work was supported in part by the National Science Foundation under Grant No. US NSF MCS81-17010.

0190-3918/82/0000/0082$00.75 © 1982 IEEE

(b), and (c). It implicitly assumes a global synchronizing clock, even though it can be adapted for asynchronous communication without loss of generality. The model, however, does not satisfy conditions (e) and (f); it assumes as many I/O ports as needed, that is, an I/O rate that grows with the size of the problem. Furthermore, it assumes an adequate memory (outside of the systolic array) that stores all the data, and an environment that fetches the data (possibly in parallel) from the memory and supplies them at the proper moment to the systolic array at no cost.

Moreover, the systolic array model assumes that all processors in the array are good, and if one processor becomes faulty the whole systolic array becomes faulty. This implies that the systolic array does not satisfy the definition of our WSI model since each processor must be a separate die, and has tested fault-free before it is assembled into a systolic array. A realistic model must allow for any number of faulty processors in the array. An occurrence of a fault should cause only degraded performance but not cause the system to break down.

A multiprocessor array model satisfying criteria (a) through (f) is shown in Fig. 1. The model is an array of identical Switch-Processor-Memory (SPM) modules. Each processor P communicates with its local memory M and with other processors (and memories) through the switch S. Each S is a 5×5 crossbar and communicates asynchronously with four neighboring switches. The entire switch array operates in circuit switching mode. Since there are only four possible paths from any input port (straight ahead, left, right, and toward processor), only two bits of information are needed to set up S. The communication bus between two switches, that is, the crossbar width, can be one or more bits wide and, generally, it should match the processor (arithmetic) bandwidth. In addition, each memory submodule can independently communicate with memory submodules of its neighbors. This link allows any number of SPM modules to have their memory submodules organized as one uniform queue.

We will assume that if any S, P, or M submodules is faulty, that the entire SPM module is faulty. Furthermore, any number of SPM modules can be faulty at any time. A module may be manufactured faulty or it may fail during normal operation. The problem is how to configure a partly faulty array of modules into a fault-free array. If this is done during the wafer testing it increases the yield, but a failure during the operation brings the entire system down. If a fault-tolerant operation is required, the configuration algorithm must be distributed throughout the array so that upon the detection of a fault the multiprocessor will reconfigure itself, excluding the faulty module from the set of available SPM modules. For that reason, each P stores a processor status-map in which a 0 or 1 for each SPM indicate its status, good or faulty, respectively. After each fault detection, the processor bit-map is updated.

Generally, there are two problems in WSI design. Firstly, an algorithm must be developed that can be easily mapped into the logical model of the WSI multiprocessor, that is, the fault-free array of SPM modules.

Secondly, the logical model must be mapped into the physical model which is a partly faulty array of SPM modules on the wafer. It helps, as is the case in our paper, if the logical model is one-dimensional and the physical model is two-dimensional. The algorithms for mapping into the physical model are beyond the scope of this paper. Some interesting work in this area is described in [AuCa78], [Kore81], and [FuVa82].

In this paper we describe the configuration shown in Fig. 2 and demonstrate its suitability for three important linear algebra problems.

2. Algorithms

Here we consider three simple, yet important, problems. The first deals with the determination of the distribution of the eigenvalues of a large positive definite tridiagonal matrix. This problem arises frequently in the area of mathematical physics. The second problem, which is related to the first, is that of obtaining the distribution of the real roots of a polynomial of degree n, $P_n(x) = \sum_{i=0}^{n} \gamma_i x^{n-i}$, in which all the coefficients γ_i are real and nonzero. Finally, the third problem is concerned with the solution of large positive definite tridiagonal linear systems; a problem that arises in numerous applications.

For the first two problems we employ Rutishauser's quotient difference algorithm (the QD-algorithm), e.g., see [Ruti63], [Henr58 and 63], and [ScRS73]. For the third problem, we also use an iterative algorithm. Specifically, we use the cyclic Chebyshev semi-iterative method, see [Varg62], [Wach66], and [Youn71].

2.1 The QD-Algorithm

(a) Problem 1

Let

$$\underset{\sim}{T} = [\beta_{i-1}, \alpha_i, \beta_i] \tag{1}$$

be the positive definite tridiagonal matrix under consideration, where $\beta_i \neq 0$, $1 \leq i \leq n-1$. Since the eigenvalues $\lambda_1 > \lambda_2 > \ldots > \lambda_n$ of $\underset{\sim}{T}$ are invariant under similarity transformations, we consider instead the tridiagonal matrix $\hat{\underset{\sim}{T}} = \underset{\sim\sim\sim}{DTD^{-1}}$, where $\underset{\sim}{D}$ is a diagonal matrix chosen such that

$$\hat{\underset{\sim}{T}} = \underset{\sim 1}{J} = [\beta_{i-1}^2, \alpha_i, 1] \tag{2}$$

Applying the classical LR-algorithm to J_1, see [Ruti58 and 63] and [Wilk65], as given by the iterations

$$J_k = L_k R_k$$
$$J_{k+1} = R_k L_k \qquad k = 1, 2, 3, \ldots \tag{3}$$

in which

$$L_k = \begin{bmatrix} 1 & & & \\ e_1^{(k)} & 1 & & \\ & e_2^{(k)} & 1 & \\ & & \ddots & \ddots \\ & & & e_{n-1}^{(k)} \quad 1 \end{bmatrix}, \text{ and}$$

$$R_k = \begin{bmatrix} q_1^{(k)} & 1 & & \\ & q_2^{(k)} & 1 & \\ & & \ddots & \ddots \\ & & & q_{n-1}^{(k)} \quad 1 \\ & & & q_n^{(k)} \end{bmatrix},$$

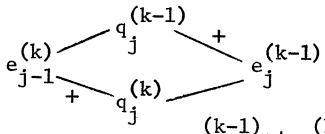
are unit lower triangular and upper triangular matrices, respectively, it is known that as $k \to \infty, L_k \to I$, and $q_j^{(k)} \to \lambda_j$. Now, from the fact that $L_{k+1} R_{k+1} = R_k L_k$, we can derive the QD-scheme for computing approximations to λ_j, $1 \le j \le n$. The scheme is given by

$$q_1^{(1)} = \alpha_1,$$
$$e_j^{(1)} = \beta_j^2 / q_j^{(1)},$$
$$q_{j+1}^{(1)} = \alpha_{j+1} - e_j^{(1)}, \qquad 1 \le j \le n-1$$

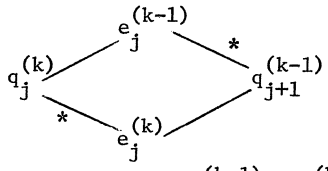
and for $k = 2, 3, \ldots$

$$q_{j+1}^{(k)} = q_{j+1}^{(k-1)} + (e_{j+1}^{(k-1)} - e_j^{(k)}),$$
$$e_j^{(k)} = q_{j+1}^{(k-1)} \, e_j^{(k-1)} / q_j^{(k)}, \qquad 0 \le j \le n-1 \tag{4}$$

where $e_0^{(k)} = e_n^{(k)} = 0$. This QD-scheme is represented by two Rhombus rules:



$$q_j^{(k-1)} + e_j^{(k-1)} = \underline{q_j^{(k)}} + e_{j-1}^{(k)} \; ; \tag{5a}$$



$$q_{j+1}^{(k-1)} * e_j^{(k-1)} = q_j^{(k)} * \underline{\underline{e_j^{(k)}}} \tag{5b}$$

(quantities to be computed are underlined). The two sequences $\{q_j^{(k)}\}$ and $\{e_j^{(k)}\}$ are guaranteed to be positive, with $\lim_{k\to\infty} q_j^{(k)} = \lambda_j$ and $\lim_{k\to\infty} e_j^{(k)} = 0$. The rate of convergence, which is only linear, can be accelerated by various techniques, see [ScRS73]. Since we are only interested in the distribution of the eigenvalues, rather than accurate approximation of them, the unaccelerated QD-algorithm is adequate for our purposes. For any iteration k, the eigenvalue λ_j lies in an interval of center

$$\mu_j^{(k)} = q_j^{(k)} + e_j^{(k)},$$

and radius

$$\rho_j^{(k)} = \sqrt{q_{j+1}^{(k)} \, e_j^{(k)}} + \sqrt{q_j^{(k)} \, e_{j-1}^{(k)}} \; .$$

We terminate the algorithm when, for any k, $(q_i^{(k)} \; e_{i-1}^{(k)})$, $2 \le i \le n$, is less than a given tolerance.

Fig. 3 shows the dataflow diagram of our algorithm. The obvious mapping of the dataflow diagram onto the chain of Ps is to assign each row to one P. In this case, one single value has to be sent from one P to the neighboring P during the time of two arithmetic operations. Therefore, for a balanced design, the communication bandwidth of a processor P must be $b/\min(t_d+t_m, t_a+t_s)$ bps, where b is the number of bits in each value sent and t_d, t_m, t_a, and t_s are the execution time for division, multiplication, addition, and subtraction, respectively. Furthermore, it takes $2k(t_d+t_m+t_a+t_s)$ to generate $q_1^{(k)}$, and $(n+2k)$ $(t_d+t_m+t_a+t_s)$ to obtain $q_1^{(k)}, q_2^{(k)}, \ldots, q_n^{(k)}$.

(b) Problem 2

Consider the n-th degree polynomial with unit leading coefficient

$$P_n(x) = x^n + \gamma_1 x^{n-1} + \ldots + \gamma_{n-1} x + \gamma_n$$

where all the γ_i's are real and different from zero. The roots of $P_n(x)$ are, therefore, either real or appear in complex conjugate pairs. The QD-algorithm used for Problem 1 can be adapted to obtain all the real roots, as well as the real coefficients of the quadratic factors whose roots

constitute the complex conjugate roots of $P_n(x)$, e.g., see [Henr58, 64, and 67]. The QD-table for this problem is generated row by row using the two Rhombus rules (5), where the top two rows of the table are given by:

$$q_1^{(0)} = -\gamma_1, \quad q_j^{(-j+1)} = \gamma_j \qquad 2 \le j \le n, \tag{6a}$$

and

$$e_j^{(-1)} = -1, \quad e_j^{(-j)} = 1 \qquad 2 \le j \le n-1, \tag{6b}$$

with $e_0^{(0)} = e_n^{(-n)} = 0$.

In Fig. 4, we show the flow of the computation for n = 3. If the horizontal strip (indicated by dashed lines in Fig. 4) is mapped on one PE, then the communication rate between two PEs is $b/\min(t_d+t_m, t_a+t_s)$ bps. If only half the strip is mapped on one PE, the rate is doubled. The execution time is the same as in the previous example. We now state the following theorem regarding the convergence of the algorithm.

Theorem [Henr67]

Let the roots of $P_n(x)$ be $z_1, z_2, \ldots$, then

(i) for every j such that $|z_j| > |z_{j+1}|$,

$$\lim_{k\to\infty} e_j^{(k)} = 0.$$

(ii) for every j such that $|z_{j-1}| > |z_j| > |z_{j+1}|$,

$$\lim_{k\to\infty} q_j^{(k)} = z_j.$$

(iii) for every j such that $|z_{j-1}| > |z_j| \ge |z_{j+1}| > |z_{j+2}|$,

$$\lim_{k\to\infty} q_j^{(k)}\, q_{j+1}^{(k)} = z_j\, z_{j+1}, \text{ and}$$

$$\lim_{k\to\infty} [q_j^{(k+1)} + q_{j+1}^{(k)}] = z_j + z_{j+1}$$

Hence, we terminate the algorithm when, for any k,

$$|(q_j^{(k)} + q_{j+1}^{(k-1)}) - (q_j^{(k-1)} + q_{j+1}^{(k-2)})| \le \varepsilon$$

where ε is a given tolerance.

2.2 An Iterative Tridiagonal Solver

Here, we assume that one requires solving linear systems of the form $Tx = f$, where T is that positive definite matrix given in (1) with all the diagonal elements $\alpha_i = 1$. Without loss of generality, we assume that n is even.

Let

$$\underset{\sim}{x}^T = (x_1, x_2, \ldots, x_n),$$

$$\underset{\sim}{f}^T = (f_1, f_2, \ldots, f_n),$$

and $\underset{\sim}{P}$ be a permutation matrix such that the linear system $(\underset{\sim}{P}^T \underset{\sim}{T} \underset{\sim}{P})(\underset{\sim}{P}^T \underset{\sim}{x}) = (\underset{\sim}{P}^T \underset{\sim}{f})$ is of the form

$$\begin{bmatrix} \underset{\sim}{I}_{n/2} & \underset{\sim}{E} \\ \underset{\sim}{E}^T & \underset{\sim}{I}_{n/2} \end{bmatrix} \begin{bmatrix} \underset{\sim}{x}^{(R)} \\ \underset{\sim}{x}^{(B)} \end{bmatrix} = \begin{bmatrix} \underset{\sim}{f}_R \\ \underset{\sim}{f}_B \end{bmatrix} \tag{7}$$

where $\underset{\sim}{E}$ is the lower bidiagonal matrix

$$\underset{\sim}{E} = \begin{bmatrix} \beta_1 & & & \\ \beta_2 & \beta_3 & & \\ & \beta_4 & \beta_5 & \\ & & \ddots & \ddots \\ & & & \beta_{n-2} \quad \beta_{n-1} \end{bmatrix},$$

$\underset{\sim}{x}^{(R)}$ and $\underset{\sim}{f}_R$ contain the odd-indexed elements of $\underset{\sim}{x}$ and $\underset{\sim}{f}$, respectively, and $\underset{\sim}{x}^{(B)}$ and $\underset{\sim}{f}_B$ contain the even-indexed elements. The cyclic Chebyshev semi-iterative scheme is given as follows, see [Varg62].

$\underset{\sim}{x}_0^{(R)}$ is chosen arbitrarily,

$$\underset{\sim}{x}_1^{(B)} = (\underset{\sim}{f}_B - \underset{\sim}{E}^T \underset{\sim}{x}_0^{(R)}), \text{ and}$$

$$\underset{\sim}{x}_{2k}^{(R)} = (1 - \omega_{2k})\underset{\sim}{x}_{2k-2}^{(R)} + \omega_{2k} (\underset{\sim}{f}_R - \underset{\sim}{E}\, \underset{\sim}{x}_{2k-1}^{(B)}),$$

$$k = 1, 2, 3 \ldots \tag{8}$$

$$\underset{\sim}{x}_{2k+1}^{(B)} = (1 - \omega_{2k+1})\underset{\sim}{x}_{2k-1}^{(B)} + \omega_{2k+1} (\underset{\sim}{f}_B - \underset{\sim}{E}^T \underset{\sim}{x}_{2k}^{(R)}).$$

Here, ω_j's are the optimal acceleration parameters and are given by

$$\omega_2 = 2/(2-\rho^2),$$

$$\omega_{j+1} = (1 - \frac{\rho^2}{4}\omega_j)^{-1}, \qquad j \ge 2,$$

in which $\rho < 1$ is the spectral radius of the matrix

$$\underset{\sim}{H}_J = \begin{bmatrix} \underset{\sim}{0} & -\underset{\sim}{E} \\ -\underset{\sim}{E}^T & \underset{\sim}{0} \end{bmatrix}.$$

Assuming that ρ is given, the iteration (8) may be written as

1. $\eta_1 := 2$
2. $\underset{\sim}{x}_1 :=$ a random vector of order $n^2/2$
3. $\underset{\sim}{y}_1 := \underset{\sim}{f}_B - \underset{\sim}{E}^T \underset{\sim}{x}_1$
4. $i := 2$
5. $\xi_i := (1 - \frac{\rho^2}{4}\eta_{i-1})^{-1}$ (9)
6. $\underset{\sim}{x}_i := (1 - \xi_i)\underset{\sim}{x}_{i-1} + \xi_i(\underset{\sim}{f}_R - \underset{\sim}{E}\,\underset{\sim}{y}_{i-1})$
7. $\eta_i := (1 - \frac{\rho^2}{4}\xi_i)^{-1}$
8. $\underset{\sim}{y}_i := (1 - \eta_i)\underset{\sim}{y}_{i-1} + \eta_i(\underset{\sim}{f}_B - \underset{\sim}{E}^T \underset{\sim}{x}_i)$
9. $i := i + 1$
10. If stopping criterion is not satisfied, go back to 5.

The stopping criterion is based on the convergence of $\underset{\sim}{x}^{(R)}_{2k}$ ($\underset{\sim}{x}_i$ in the above program segment) and is given by

$$||\underset{\sim}{f}_R - \underset{\sim}{E}\,\underset{\sim}{y}_{i-1}||_\infty \leq \varepsilon$$

where ε is a specified tolerance. If ρ is not given a priori, we have several options:

(i) If the system $Tx = f$ is to be solved for many right-hand sides (not necessarily all at once), then it may be advantageous to use our QD-algorithm in Problem (1) to evaluate the largest eigenvalue $\mu = 1 + \rho$ of the tri-diagonal matrix

$$\underset{\sim}{T} = [\beta^2_{i-1},\ 1,\ 1]$$

(ii) If $\underset{\sim}{T}$ is strongly diagonally dominant, then we may either take all ω_i's to be 1, i.e., use the classical Jacobi method, or estimate ρ by $||\underset{\sim}{E}||_\infty$.

We show the flow of the computation, equation (9), in Fig. 5, where each node performs the computation indicated either by step 6 or step 8. The quantities ξ, $(1-\xi)$, and η, $(1-\eta)$ are computed ahead of the main computation and stored in each PE as constants. Since there are four multiplications and three additions per node, the total time is $t^* = 4t_m + 3t_a$. Each node requires two components of $\underset{\sim}{x}$ and $\underset{\sim}{y}$ in addition to three constants β_{i-1}, β_i and f_j. However, the switch in P_k is used by $PE_{(k-1)}$ to communicate the other three constants (not used in the k-th PE) to the $PE_{(k+1)}$. Hence, for a balanced design we must have $8b(4t_m+2t_a+t_s) \simeq 1$, (see Fig. 5).

3. Processor Design

The processing submodule which satisfies the requirements given in Section 1 (Fig. 6) consists of an arithmetic unit performing floating-point division, multiplication, addition, and subtraction, two register-files with three registers each and four queues. The input queues DQ, XQ, and YQ are register files with a serial input port and a parallel output port, while the output queue OQ has a parallel input port and a serial output port. The communication through the switch submodule is asynchronous and bit-serial. The operands are always sent or taken from each queue in the same order. However, the order in which the operands are distributed among three input queues is not guaranteed. For example, the operand for the XQ may arrive before or after the operand to be stored in the YQ. If an input queue is empty, the control unit waits until the data arrives. In our statically scheduled operation it is not necessary to associate a validity bit with each operand (as in some dataflow machines), since the order is determined ahead of time. Two extra bits must be added to each data value to distinguish among the three input queues. Furthermore, if the data is sent to a nearest neighbor, two additional bits are needed to set up the switch in the neighboring SPM module. Thus, four bits of overhead are necessary for the communication between nearest neighbors.

As an example of processor operation, the dataflow-diagram of a node from Fig. 5 is shown in Fig. 7(a). There are four multiplications, two additions, and one subtraction to be performed. There are three system constants (stored in DQ) and two variables (stored in XQ and YQ) passing through the switch. The variable $x_{i,j}$ is already in the XQ from the previous iteration. Each node in the dataflow graph generates one result, which is stored in the register indicated on the arc going out of the node. The sequence of microinstructions for the given dataflow graph is shown in Fig. 7(b). An arithmetic operation and a move to the OQ are performed in parallel. Similar microinstruction sequences can be written for other dataflow diagrams.

Using the present-day technology rate for floating-point arithmetic ([WaMc82], t_d = 2.5 μs, $t_m = t_a = t_s$ = .5 μs) and communication rate (50 MB/sec) for a wafer communication, we can compute the ratio t_{comm}/t_{arith}. If the 32-bit floating-point format is assumed, then $t_{comm}/t_{arith} = (8\times36/50\times10^{-6})/(4t_m + 2t_a + t_s) = 5.44/3.5 \simeq 1.6$. We see that our algorithm for problem 3 is communication intensive. The performance can be improved by doubling the width of the switch from one bit to two bits. This will, of course, double the cost of the switch but will result in more balanced design, since the t_{comm}/t_{arith} will become 0.8 for 32-bit floating-point format.

The algorithms for problems 1 and 2 are

arithmetic intensive.

4. Conclusion

We have presented in this paper a multiprocessor model for systolic algorithms. We think it is better suited for Wafer-Scale-Integration than the systolic-array model. In particular, in our model the communication is asynchronous and the model is fault-tolerant, which will improve the yield during manufacturing and allow graceful degradation during the operational life of the system.

Since the packaging technology is the bottleneck in system design, the iterative method in problem 3 compares favorably with direct methods. While the direct method requires roughly $5(t_m + t_a)n$, our algorithm needs time $(1 + [j/k])(n + 2k)(4t_m + 2t_a + t_s)$ for j iterations, where k is the number of SPM modules in the chain. For strongly diagonally dominant matrices the number of iterations, even with the ω's taken as unity, is small, and this iterative algorithm is competitive with the direct method for large n. For example, if $||\underset{\sim}{E}|| \leq 0.8$, and $\omega_j = 1$, then the maximum norm of the error after 90 iterations will be roughly 10^{-9} that of the initial error. Hence, for $n \simeq 1000$, one hundred good SPM modules on a wafer will yield a solution with a reasonable accuracy in roughly 0.8 the time needed by the direct method; we used the floating-point arithmetic rates stated above. The performance is approximately the same since most of the algorithms are I/O limited and not arithmetic limited. Although our model takes more silicon than models used for direct implementations, it offers fault-tolerance, simplicity and regularity which outweigh, in our opinion, the cost.

References

[AuCa78] R. C. Aubuson and I. Catt, "Wafer-Scale Integration--A Fault Tolerant Procedure," Journ. of Solid-State Cir., Vol. SC-13, (1978), pp. 339-344.

[FuVa82] D. Fussell and P. Varman, "Fault-Tolerant Wafer-Scale Architecture for VLSI," Proc. of the 9th International Conf. on Computer Architecture (1982), pp. 190-198.

[Henr58] P. Henrici, "The Quotient-Difference Algorithm," Nat'l. Bureau of Standards Appl. Math. Ser. 49, (1958), pp. 23-46.

[Henr63] P. Henrici, "Some Applications of the Quotient-Difference Algorithm," Proc. Symposia in Appl. Math., Vol. 15, (1963), pp. 159-183.

[Henr64] P. Henrici, Elements of Numerical Analysis, John Wiley & Co., (1964).

[Henr67] P. Henrici, "Quotient-Difference Algorithms," in Math. Methods for Digital Computers, Vol. 2, A. Ralston and H. Wilf (eds.), pp. 37-62, John Wiley & Co., (1967).

[Kore81] J. Koren, "A Reconfigurable and Fault-Tolerant VLSI Multiprocessor Array," Proc. of the 8th International Conf. on Computer Architecture, (1981), pp. 425-442.

[KuLe79] H. T. Kung and C. E. Leiserson, "Systolic Arrays (for VLSI)," in I. S. Duff and G. W. Stewart (eds.), Sparse Matrix Proceedings 1978, SIAM, Phila., (1979), pp. 256-282; also, in C. A. Mead and L. A. Conway, Introduction to VLSI Systems, Sec. 8.3, Addison-Wesley, Reading, (1980).

[Ruti58] H. Rutishauser, "Solution of Eigenvalue Problems with the LR-Transformations," Nat'l. Bureau Standards Appl. Math. Ser. 49, (1958), pp. 47-81.

[Ruti63] H. Rutishauser, "Stabile Sonderfalle des Quotienten-Differengen Algorithmus," Numer. Math., Vol. 5, (1963), pp. 95-112.

[ScRS73] H. Schwarz, H. Rutishauser, and E. Stiefel, Numerical Analysis of Symmetric Matrices, Prentice-Hall, (1973).

[Varg62] R. Varga, Matrix Iterative Analysis, Prentice-Hall, (1962).

[Wach66] E. Wachpress, Iterative Solution of Elliptic Systems and Applications to the Neutron Diffusion Equations of Reactor Physics, Prentice-Hall, (1966).

[WaMc82] F. Ware and W. McAllister, "C-mos Chip Set Streamlines Floating-Point Processing," Electronics, (Feb., 1982), pp. 149-152.

[Wilk65] J. Wilkinson, The Algebraic Eigenvalue Problem, Oxford, (1965).

[Youn71] D. Young, Iterative Solution of Large Linear Systems, Academic Press, (1971).

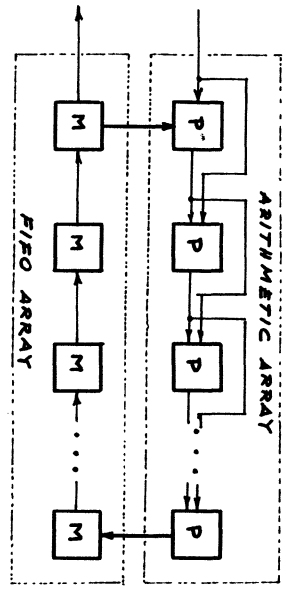


Fig. 2. Logical model for iterative algorithms

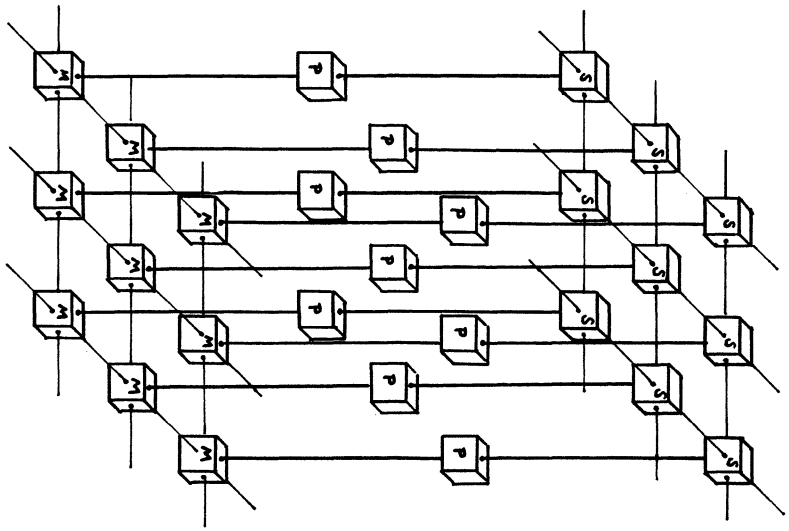


Fig. 1. WSI-Multiprocessor model

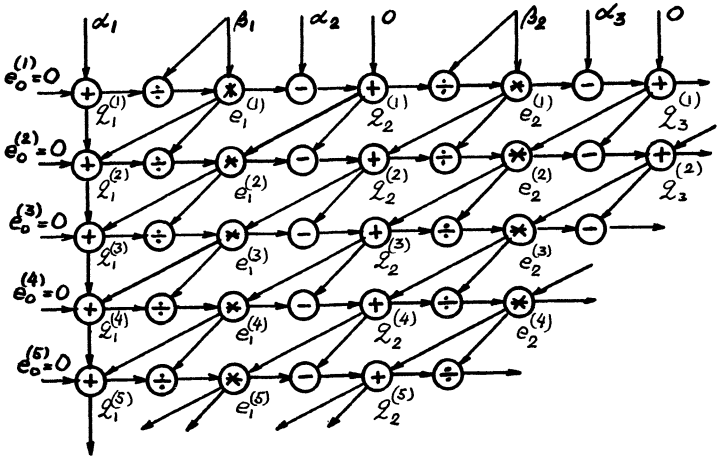


Fig. 3. Dataflow diagram for the eigenvalue problem

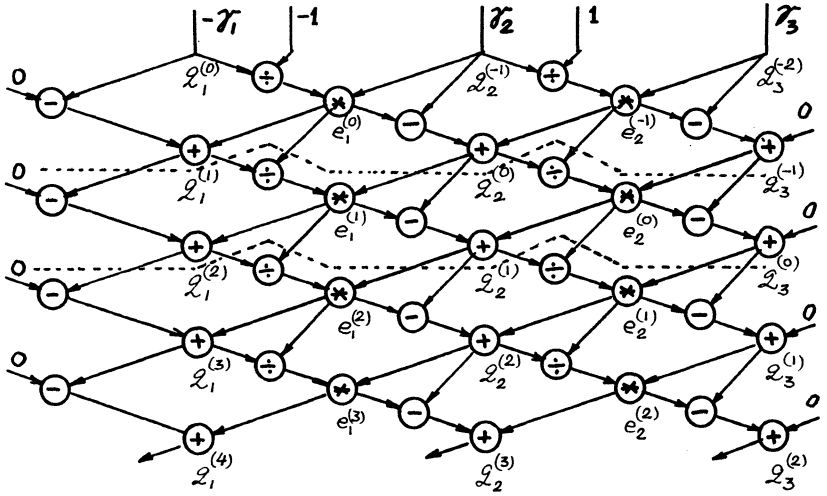


Fig. 4. Dataflow diagram for roots of polynomials

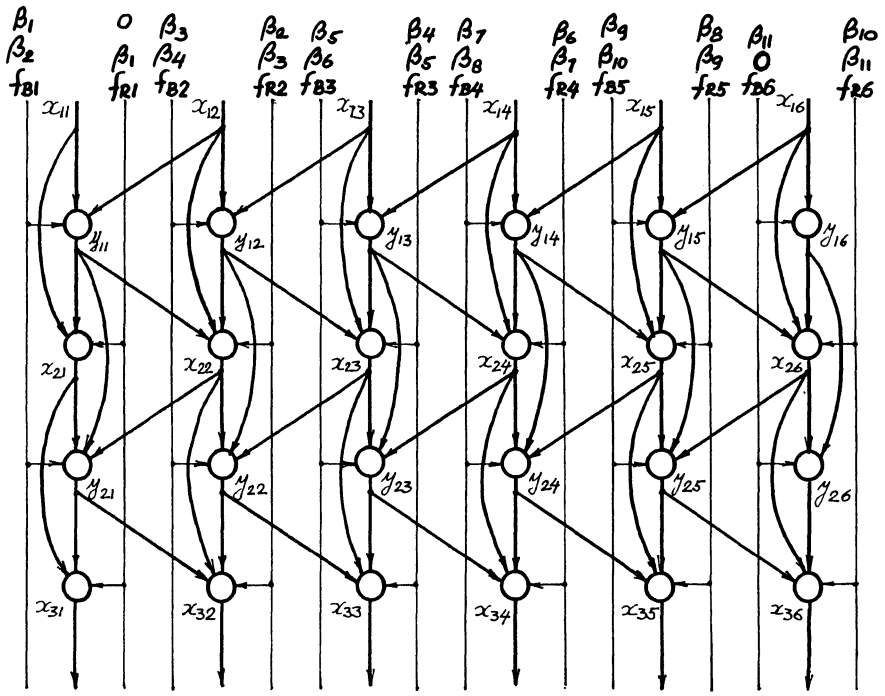


Fig. 5. Dataflow diagram for tridiagonal solver

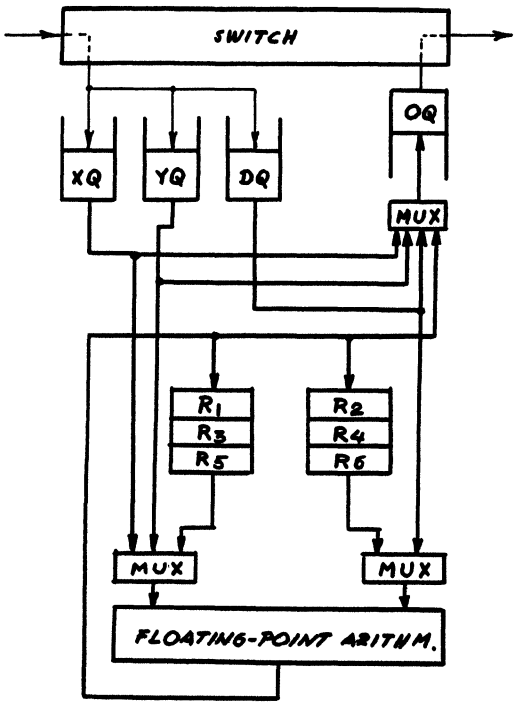
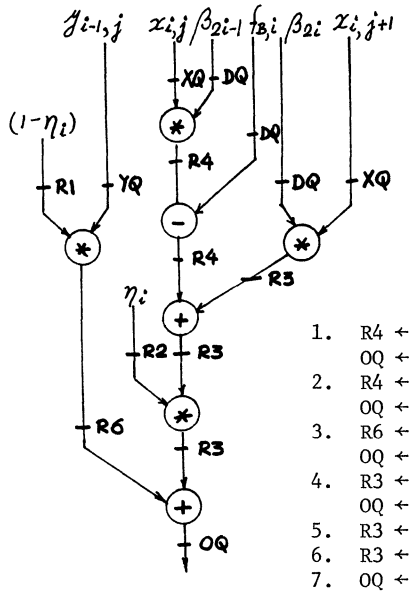


Fig. 6. Processor block diagram



(a)

```
1.  R4 ← XQ * DQ,
    OQ ← DQ, POP DQ;
2.  R4 ← R4 - DQ,
    OQ ← DQ, POP DQ;
3.  R6 ← R1 * YQ,
    OQ ← XQ, POP XQ, POP YQ;
4.  R3 ← DQ * XQ,
    OQ ← DQ, POP DQ;
5.  R3 ← R4 + R3;
6.  R3 ← R2 * R3;
7.  OQ ← R6 + R3;
```

(b)

Fig. 7. (a) Dataflow diagram for node y_{ij} from Fig. 5;

(b) Microinstruction sequence for processor in Fig. 6

OPTIMAL IMPLEMENTATION OF SIGNAL FLOW GRAPHS ON SYNCHRONOUS MULTIPROCESSORS

T. P. Barnwell, III
C. J. M. Hodges

School of Electrical Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332

INTRODUCTION

This paper discusses some results of a theoretical and experimental study of a general procedure for implementing recursive and nonrecursive signal flow graphs and other similar arithmetic algorithms on synchronous digital machines composed of multiple identical programmable processors. A fundamental feature of our approach is that it attempts to make maximum use of the Skewed Single Instruction Multiple Data (SSIMD) mode [1-3] of the synchronous multiprocessor system. When operating in this mode, the multiprocessor executes exactly the same program on all of the processors simultaneously, but with a fixed time skew imposed between the instructions execution times on the separate processors. In addition, the single program utilized in the SSIMD mode is always exactly a single processor realization of all the computations associated with a single time index of the signal flow graph. Hence, where the SSIMD mode of an appropriate single-processor implementation.

A primary goal of this research is to develop procedures for automatically generating optimal multiprocessor signal flow graph implementations from a simple, non-parallel representation of the algorithms to be implemented. An appropriate algorithmic representation might be a set of difference equations or a matrix presentation [2-4] of the signal flow graph.

In a study of this type, it is very important to carefully define the criterion for optimality. In this study, three definitions of "optimal" are used. An implementation is said to be processor-optimal if the use of M processors leads exactly to an M-fold increase
in the systems throughput as compared to a single processor implementation. An implementation is said to be time-optimal if the absolute theoretical limit [5] for that signal flow graph has been achieved for the particular constituent processor. Finally, an implementation is said to be absolutely-optimal or just optimal if it is time-optimal and there exists no other solution which requires fewer processors.

In these terms, it is now possible to summarize our results thus far. First, for a very large class of recursive signal flow graphs, the SSIMD approach results in absolutely-optimal implementations. This class includes all direct form digital filters and their transposed forms, all lattice form digital filters, all parallel or cascade digital filters based on lattice or direct forms, and many more. Second, where an absolutely-optimal SSIMD solution exists, it can be constructed automatically. Third, where an absolute-optimal SSIMD solution does not exist, a time-optimal solution can be constructed using a Parallel Skewed Single Instruction Multiple Data (PSSIMD) structure. In a PSSIMD implementation, two or more programs coexist in the same implementation. The question of whether the PSSIMD time-optimal solution is absolutely-optimal is difficult to answer in general, but it is clear that the PSSIMD solutions obtained in this fashion are very efficient.

THE SSIMD MODE

The techniques of interest all utilize the Skewed Single Instruction Multiple Data (SSIMD) mode of the synchronous multiprocesors to realize the implementations. In this mode, exactly the same instruction stream is executed on all the processors, but a fixed time skew is maintained between instruction execution times on separate processors.

The fundamental concept is illustrated by the simple example of Figure 1. In this example, the second order direct form filter of Figure 1a is implemented as a single processor program as shown in Figure 1b. In this single processor realization, none of the delay elements are realized directly, but rather the output from each delay element becomes an input to the program and the input to each delay element becomes an output from the program. In the SSIMD realization, these delayed values are not computed by this processor, but are supplied from identical computations on other processors.

Figure 2 illustrates the fundamental character of an SSIMD solution. In the one processor solution of Figure 2a, the same processor which generates the output point r(n) is also the processor which has generated all the previous output points, r(m) for $m < n$. Hence these points are always available when needed. In Figure 2b, a two processor solution is illustrated. The key point is that, even though the value of r(n-1) must be available before r(n) is computed, it is not necessary for it to be available before the computation of r(n) is begun. What is required, rather,

0190-3918/82/0000/0090$00.75 © 1982 IEEE

is that the value of r(n-1) must be available before it is used by processor 1. Hence processor 1 may be started as soon as it is guaranteed that r(n-1) will be available from processor 2 before it is needed by processor 1.

Figure 3 shows the diagram for a one processor, a two processor, and a five processor realization for the signal flow graph of Figure 1. In the single processor solution of Figure 3a, all of the past values of r(n) are supplied by the same processor, and there is never any issue of data availability. In the two processor realization of Figure 3b, alternate points are supplied by each processor, and the two processors must be skewed such that the data requirements of each is always met by the other. Likewise, Figure 3c shows a 5 processors solution in which every 5th set of points is supplied by each of the 5 processors. It should be noted that all these SSIMD solutions are "free running" such that whenever a processor completes the computations associated with one time index, it immediately begins the computations associated with another time index. Hence, each program realizes an infinite loop (one time index per loop) and, under the assumption that the program timings are not data dependent, each loop takes exactly the same amount of time. Thus, if M processors are started at M starting time, $t_m, 0 \leq m \leq M-1$, then the relative time skew so imposed remains fixed untill the program is halted externally. Hence, the program of implementing a particular recursive and iterative arithmetic program reduces to specifying the M starting times, t_0 ..... t_{M-1}, such that all the data necessary in the various computations is available before it is needed.

Fixed Program Implementations

The problem of implementing a particular recursive signal flow graph in SSIMD mode can be divided into two related problems. The first problem is that of finding and characterizing all legal SSIMD solutions for a particular single processor program realization of the signal flow graph. The second problem is that of constructing the particular single processor program so that the eventual SSIMD solution will be optimal. This section addresses the first problem.

In fitting the program together in the SSIMD mode, the only information necessary concerns the length of the program, the times at which recursive inputs are needed, and the times at which recursive output are available. Hence, a program with a single recursive output such as that of Figure 1 can be characterized as

$$K(I(L), I(L-1), \ldots\ldots, I(1), R, T) \qquad (1)$$

where K is the task identifier, T is the task length, R is the output time for the recursive output, $I(\ell)$ is the input time for the ℓth delayed recursive output, and L is the value of the longest delay. The important theoretical results for this environment can be summarized as follows [3-4]:

1) All SSIMD M-processor solutions are bounded by the solution in which the processors are started at equal intervals and the outputs are periodic. For such a solution, the time between outputs is T/M and

$$t = \frac{mT}{M} \qquad 0 \leq m \leq M-1 \qquad (2)$$

Stated another way, if an M-processor (processor-optimal) solution exists, it can be implemented with the above starting times.

2) The maximum number of processors which can be used in such a solution, M, is given by

$$M_x = INT[M(\ell_x)] = INT\left[\min_{\ell}[M(\ell)]\right] \qquad (3a)$$

$$M_x = INT\left[\min_{\ell}\left[\frac{\ell\, T}{I(\ell)-R}\right]\right] \qquad (3b)$$

where $M(\ell)$ is the non-integer number of processors which could be utilized if the only constraint came from the recursive input $I(\ell)$, ℓ_x is the value of ℓ for which $M(\ell)$ is minimum, and $INT[\cdot]$ means "the integer part".

3) Any SSIMD implementation for the given program can be obtained with uniform time skews as shown in (1) above so long as $M \leq M_x$.

4) The greatest throughput which is achievable by these techniques is obtained with a time skew of

$$t_x = \frac{I(\ell_x) - R}{\ell_x} \qquad (4)$$

This solution is generally achieved with $M_x + 1$ processors by adding extra non-functional delays to the program, and although time-optimal, is generally not processor-optimal. A solution which is both time-optimal and processor-optimal occurs for M_x processors only for the unlikely case of $M_x = M(\ell_x)$.

5) Time-optimal solutions are available for $M_x + 1$ processors, and the addition of more than one processor will never increase the throughput beyond a sample rate of $1/t_x$.

Based on these results, three important features should be noted. First, given a single processor program for a signal flow graph or other algorithm describable as in equation (1), the maximum number of processors which can be used is immediately available (eq. 3b) and the starting times in the SSIMD solutions are trivially simple to compute (eq. 2). Hence, for a given program the SSIMD implementation procedure is very simple. Second, and more important, the maximum number of

processors which can be utilized (eq. 3) is a function of only a single input time, $I(\ell_x)$. Hence, a simple constraint exists for optimizing a particular program for an SSIMD implementation. This program is obtained by maximizing the minimum value of $M(\ell_x)$. Finally, and perhaps most important, the optimum time skew, t_x, is a function of neither the program duration or the number of recursive inputs or outputs for the program. This allows for several important generalizations to be made, and, for properly written programs, leads to very impressive solutions. For example, the system of Figure 1 can typically be implemented with 8 or 9 processors, even though it has only two recursive inputs.

OPTIMAL SIGNAL FLOW GRAPH IMPLEMENTATIONS

Based on the results summarized in the previous section, it is clear that any program which implements a signal flow graph can be the basis for an SSIMD solution. In regard to the optimality of such programs, three separate issues must be addressed. First, how is the maximum throughput, the rate which defines a time-optimal solution, determined for any signal flow graph? Second, how is the question of the existence of a time-optimal SSIMD solution to be addressed, and how is a time-optimal SSIMD solution constructed if it does exist? Finally, if no time-optimal SSIMD solution exists, how can a time-optimal PSSIMD solution be constructed?

The first question has been addressed in a paper by Renfors and Neuvo [5], in which they show how to determine the maximally attainable throughput for any signal flow graph given the arithmetic constraint of the processor. The procedure can be summarized as follows. The first step is to expand the signal flow graph node structure so that all arithmetic operations occur as individual branches (see Figure 4). Note that this procedure deterministically sets the precedence relations among all arithmetic operations. The second step is to measure the arithmetic delays and count the delay elements (z^{-N_ℓ}) associated with each loop. The minimum sampling period (and hence the maximum throughput) for such a system is given by

$$T_x = \max_{\substack{\text{all} \\ \text{loops}}} \left\lceil \frac{T_\ell}{N_\ell} \right\rceil \qquad (5)$$

where T_ℓ is the total arithmetic delay in the ℓth loop and N_ℓ is the total (integer) number of delay elements in the ℓth loop. For an implementation to be time-optimal it must attain this limit.

Using the above result, it is a straightforward process to construct all the time-optimal SSIMD solutions which exists, and the best SSIMD solution if no time-optimal solution exists. All that is required is to first construct all possible signal flow graphs of the type of Figure 4 from the original signal flow graph. This is accomplished by systematically expanding each node into all possible sets of nodes each of which involves only one multiply and one add. Then, for each of these expanded signal flow graphs, all possible arithmetic orderings can be enumerated in a directed search using the intrinsic precedence relations [6]. Each of these orderings constitutes a program for realizing a single processor implementation, and the maximum throughput for each is computable from eq. (4). The program(s) with the minimum value of t_x are the best SSIMD solution, and if $t_x = T_x$, then that solution is time-optimal. A key point in this regard is that if a time-optimal SSIMD solution exists, then it is also absolutely-optimal. This is because the SSIMD solution uses all the cycles of all the processors on the algorithm, which is the best that can ever be achieved.

It is possible to structure the searching procedure described above so as to simultaneously construct PSSIMD solutions if no time-optimal SSIMD solution exists. The procedure begins by removing all the delay elements from the signal flow graph as shown in Figure 1b. Then all of the loops in the system are tabulated as follows. First, all first order loops are found by replacing each delay element and tabulating any resulting loops. Second, all second order loops are found by replacing all delay elements in sets of two, and tabulating all loops not previously found. This process is repeated for increasing numbers of delay elements until all loops are tabulated. One of the loops tabulated in this procedure must be the limiting loop of eq. (5), and this loop must occur as a set of contiguous arithmetic operations in any SSIMD solution. Hence, the optimal program construction task reduces to ordering the remaining operations so as not to violate equation (5).

For many classes of recursive systems, such as direct form, cascade direct form, and parallel direct form filters, the individual loops do not "overlap" or, in other words, they do not share arithmetic operations. For such system, each loop can be implemented separately in several allowable orders and several absolutely-optimal SSIMD solutions exist. For other systems, such as the lattice filter of Figure 4, the loops overlap, but it is possible to order the remaining operations so as not to violate equation (5). For still other systems, such as the coupled form, no time-optimal SSIMD solution exists. Alternately, a time-optimal PSSIMD solution can be constructed by systematically "offloading" the loop overlap operations to other processors. These secondary "slave" processors are synchronously locked to the "master" processor, which is defined as the processor which implements the limiting loop. This PSSIMD construction can always be used to construct a time-optimal solution, but the question of its absolute-optimallity is harder to address. It is clear, however, that the slave processors still automatically benefit from the same SSIMD gains as the master processor. This means, for example, that if the total arithmetic processing time for the master processor is T_M and the total arith-

metic processing time for a slave processor is T_m/N, then N master processors can be serviced by one slave.

DISCUSSION

All of the above results are really a reflection of the intrinsic data flow constraints inherent in recursive algorithms, and they are obtained by mixing three sets of constraints: the fundamental recursive constraints of the algorithm; the simple, highly structured constraints of the signal flow graphs; and the constraints imposed by SSIMD realizations. There are several important points here. The first is that if a SSIMD mode exists in a multiprocessor, then there are no better ways for implementing many digital filters. This fact is made even more attractive by the fact that the SSIMD implementations are generally much simpler than other multiprocessor options which typically involve the parsing of the signal flow graph to exploit the local parallelism. The second important point is that all the limits on the number of processors and the throughput (t_x) are a reflection of the recursive nature of the algorithms. If the programs are not recursive, then the programs can be implemented such that there are no such constraints, and the number of usable processors goes to infinity. What this means, clearly, is that the solution is no longer constrained by the algorithm but rather by the nature of the I/O hardware. The key point here is that the SSIMD solution for non-recursive programs is processor-optimal for any number of processors, and these solutions are even simpler to implement than the solutions for the recursive case.

The largest potential problem in SSIMD solutions concerns the inter-processor communication issues. Since the entire SSIMD development has been done under the assumption that the processors could communicate "at will", this would at first seem like a critical issue. It turns out, however, that it is not. This is true for two reasons. First, the fundamental periodicity of the SSIMD solution makes the communications requirements very uniform, which avoids many potential time conflicts. Second, and most important, the nature of the communications environment can be systemtically controlled. To see this, one simply needs to note that the number of processors with which a particular processor must communicate is controlled by the Maximum length of its delay elements (see Figures 1, 2, and 3). The use of long delay chains does improve the final solution since it leads to SSIMD realizations which require fewer processors to realize a time-optimal solution. But the entire procedure still works if the maximum delay length is constrained to be one. Indeed, this is true in the classical formulation for the signal flow graph [4]. For such realizations, each processor only talks to its two nearest neighbors, and the communications is always in one direction. Such realizations have the same maximum throughput rate, but, in general, require a few more processors to attain it. Most important, however, they have a communication environment which is always trivally implementable.

REFERENCES

1. C. J. M. Hodges, T. P. Barnwell, III and D. McWhorter, "Implementation of an all Digital Speech Synthesizer Using a Multi-processor Architecture," 1980 International Conference on Acoustics, Speech, and Signal Processing, Denver, Colorado, April 1980.

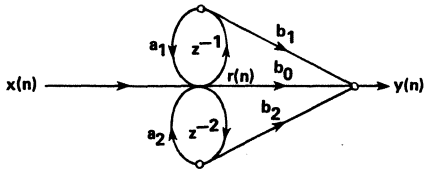
2. T. P. Barnwell, "Optimal Implementations of Recursive Signal Flow Graphs on Synchronous Multiprocessor Architectures in SSIMD Mode," Paper in preparation.

3. T. P. Barnwell and C. J. M. Hodges, "Optimum Implementation of Single Time Index Signal Flow Graphs on Synchronous Multiprocessor Machines," 1982 International Conference on Acoustics, Speech, and Signal Processing, Paris, France, May 1982.

4. R. Crochiere and A. V. Oppenheim, "Analysis of Linear Digital Networks," Proc. IEEE, vol. 63, pp. 581-595, April 1975.

5. M. Renfors and Vrjo Neuvo, "The Maximum Sampling Rate of Digital Filters Under Hardware Speed Constraints," IEEE Transactions on Circuits and Systems, T-CAS, pp. 196-202, March 1981.

6. T. P. Barnwell, C. J. M. Hodges and S. Gaglio, "Efficient Implementation of One and Two Dimensional Digital Signal Processing Algorithms on a Multiprocessor Architectures," 1979 International Conference on Acoustics, Speech, and Signal Processing, Washington, D. C., April 1979.



$$y(n) = b_0 r(n) + b_1 r(n-1- + b_2 r(n-2)$$

$$r(n) = a_1 r(n-1) + a_2 r(n-2) + x(n)$$

Figure 1a: Signal flow graph for a 2nd order recursive direct form II digital filter.

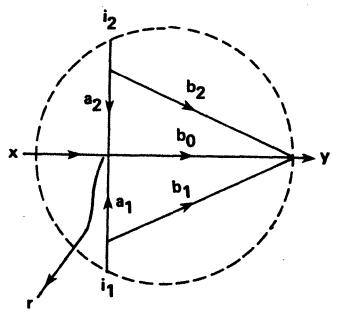


Figure 1b: Single processor realization for the signal flow graph shown above. All delays are not implemented by the program, but these are realized by the parallel structure.

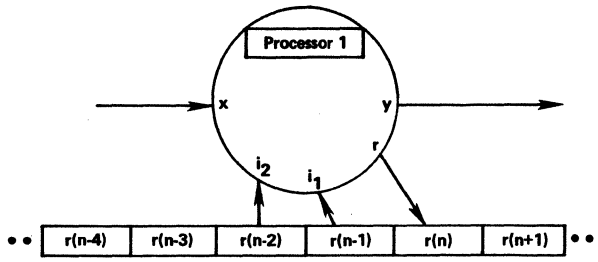


Figure 2a: Single processor implementation of the signal flow graph of Fig. 1. All points in the output stream and all values of r(n) are computed by the same processor.

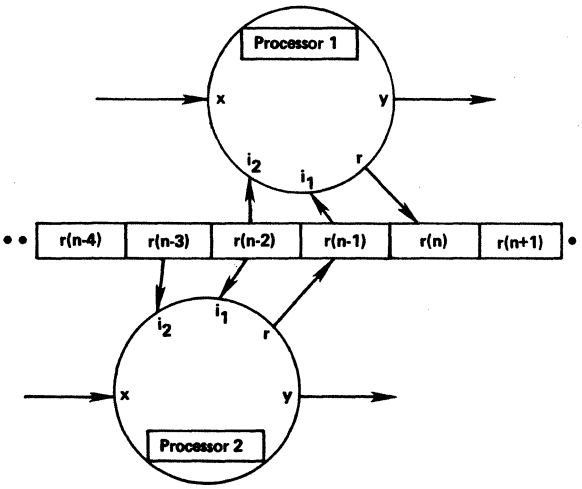


Figure 2b: Two processor implementation of the signal flow graph of Fig. 1. The computation of r(n) by processor 1 can be started as soon as processor 2 has been running long enough to guarantee r(n-1) will be available when needed by processor 1.

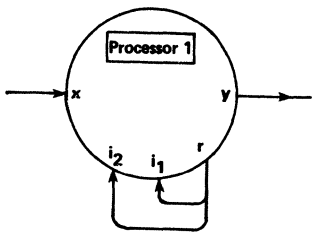


Figure 3a: In a single processor SSIMD realization, all recursive outputs are supplied by the same processor.

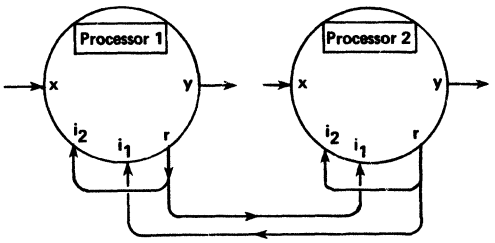


Figure 3b: In a two processor SSIMD realization, alternate recursive outputs are supplied by each processor.

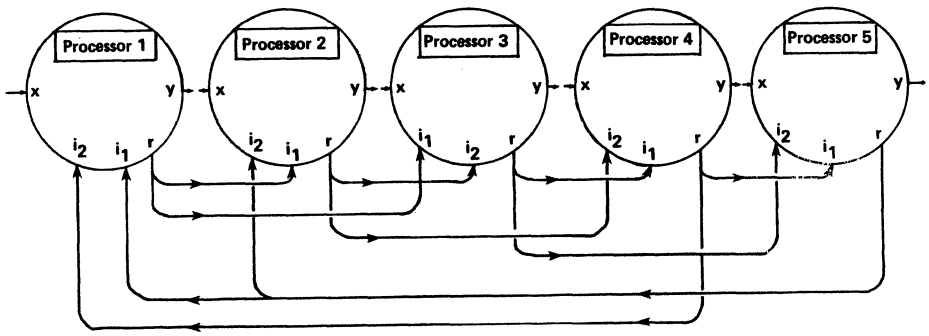
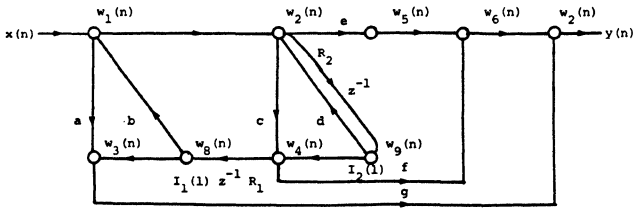


Figure 3c: In a five processor SSIMD realization, every 5^{th} time index is computed by each processor. The processors are skewed in time so as to guarantee all recursive inputs are available when needed.



LOOP INDEX	LOOP SEQUENCE	LOOP OPERATIONS	ARITHMETIC DELAY	T_ℓ
1	1-2-4-8	1*/1+/2+/4*/4+	$2d_m + 3d_a$	$2d_m + 3d_a$
2	2-9	2*/2+	$d_m + d_a$	$d_m + d_a$
3	1-2-9-4-8	1*/1+/4+	$d_m + 2d_a$	$d_m/2 + d_a$

MINIMUM SAMPLING PERIOD = $T_1 = 2d_m + 3d_a = T_0$

OPTIMUM SSIMD SOLUTIONS 2*/1*/1+/2+/4*/4+ ··· other operations ···

$$\frac{R_1 - I_1(1)}{1} = 2d_m + 3d_a \qquad \frac{R_2 - I_2(1)}{1} = 2d_m + 2d_a$$

Figure 4: Example of the derivation of an optimal SSIMD program for a 2nd order lattice filter. Each node involves one multiply, "n*", and one add, "n+", where n is the node number. The loop tabulation gives the minimum sampling period, T_0. The program has two delay outputs, R_1 and R_2, and two delay inputs, $I_1(1)$ and $I_2(1)$. The program construction procedure gives the ordering indicated, which gives a value of $t_x = T_0$. Thus, the SSIMD solution is absolutely-optimal. [Note: The storage and I/O operations have been left out of this analysis for simplicity. They can easily be included in the analysis.]

A TEST STRATEGY FOR PACKET SWITCHING NETWORKS[a]

Willie Y-P. Lim
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract -- A test strategy for packet switching networks is described. The effect of a single stuck-at fault is either misdirected packets, missing packets, corrupted data in packets, or multiple packets. A fault can either prevent packet transmission or affect the integrity of the data sent in the packet and it is detected as one of 4 cases -- both output ports of the switching element inaccessible to an input port, an output port inaccessible to an input port, an input port permanently connected to an output port and erroneous packet length.

Introduction

Packet communication architecture has been discussed in the context of implementing data-flow machines [1]. Such systems use packet switching networks for inter-processor connection. In [1] for example, the network used is composed of packet switching elements called 2x2 routers. Packet switching for another class of networks are discussed in [4]. Each packet is routed through the network using the information carried in the packet. Due to this distribution of the switching function, many packets can be simultaneously transmitted through the various stages of the network. When asynchronous or self-timed communication protocols are used, the testing of such networks requires new approaches. A strategy for testing such networks is described in this paper.

Fault diagnosis of networks has been studied by [3] for on-line fault diagnosis and by Wu and Feng [7]. The work in [7] dealt mainly with the fault diagnosis of networks in which the switching elements have single bit inputs. Wider inputs are used in packet communication. Furthermore the packet format and communication protocol used affect the test and fault diagnosis strategy.

Packet Format and Packet Communication Protocol

A packet is a sequence of bits and is usually transmitted as a sequence of sub-units with each sub-unit being some fixed number of bits. For convenience, a sub-unit is referred to as a byte. The number of bits in a byte is usually determined by chip pin-out and communication bandwidth considerations. The information contained in a packet is composed of the destination address, the data to be sent and the length of the packet. Since only packet switched networks are considered in this paper, the destination address is necessary for routing the packets through the network. The packet length information can be included in the data transmitted, or an extra bit can be used to indicate which byte is the last one in the packet.

Packets are assumed to be transmitted using asynchronous communication protocols. The transmission of each packet or byte is accompanied by an event signalling the arrival of the packet or byte at the destination and each successful receipt of a packet must be acknowledged by the explicit sending of a control signal. For example, a special signal may be used to indicate the arrival of the packet, or the arrival event may be encoded in the data signal lines as in the "dual-rail" communication protocol [2].

(a) This research was supported by the National Science Foundation under grant no. MCS-7915255 and the Department of Energy under contract no. DE-AC02-79ER10473.

A Packet Switching Network

The switching element in the network is a 2x2 router which receives packets at its two input ports and sends them out at its two output ports. The least significant bit of the address byte of the packet is used for selecting the output port for sending the packet. Output ports can be independently selected by the input ports and, if there is contention for an output port, only one of the input ports is connected while the other waits until the output port becomes free, i.e. the input is temporarily blocked. If there is no contention for an output port, then the packet transmission from an input port to an output port can proceed in parallel with a non-conflicting one. The various input-output port configurations possible are shown in Figure 1. The least significant bit of the destination address byte having a value of 0 will cause output port 0 to be selected while output port 1 will be selected if that bit is 1.

The packet switching network has the same interconnection structure as the baseline network [5], [6]. Figure 2 shows the structure of a 16x16 network. Each router in the network is connected to another router or a processor through links. For a network with N input ports and N output ports, there are $\log_2 N$ stages of routers and $1 + \log_2 N$ levels of links. The ports of the routers in each stage are numbered from top to bottom starting with 0 at the top. These numbers are not shown in Figure 2. Instead the destination addresses of the output ports of the last stage of the network are shown. If $P_s\ P_{s-1} \ldots P_1\ P_0$ with $s = (\log_2 N) - 1$ is the bit representation of the port number, then the router number in that stage is given by the value of the bit string $P_s\ P_{s-1} \ldots P_1$, i.e. by dropping the least significant bit of the port number. With this network structure, destination address bytes of the same value will route packets to the same output port of the network. The number of output ports that can be addressed, i.e. the network size, is fixed by the number of bits in the destination address byte. Port and router numbers are important for identifying ports and routers within a given stage during testing or fault diagnosis. In this paper, the network is assumed to be for connecting N processors, where N (> 0) is some power of 2.

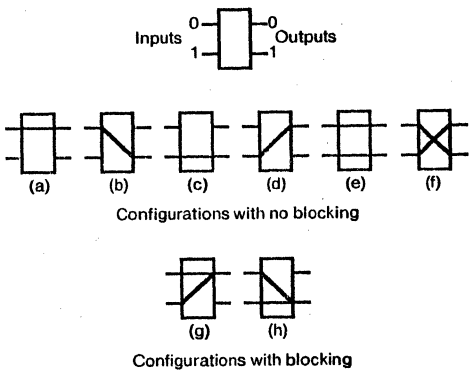


Figure 1. Port Configurations of the 2x2 Router

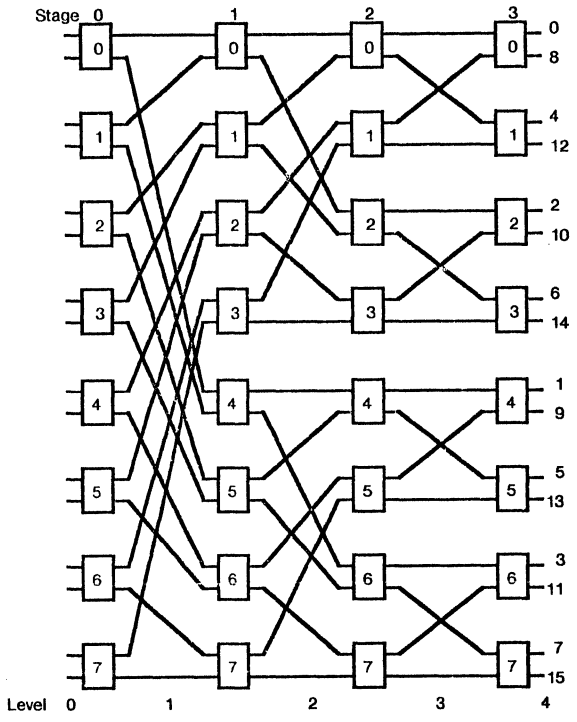


Figure 2. A 16x16 Network of 2x2 Routers

Fault Model

One or more of the following effects will be produced by every single stuck-at fault occurring in a link or router of the network - 1) misdirected packets,
2) missing packets,
3) corrupted data in packets, and,
4) multiple packets being received.

The types of faults occurring in the network can be divided into two classes. The first of these is the class of faults that affects the asynchronous communication protocol. Examples of faults in this class include the packet acknowledge or packet control signals being stuck at one of the logical values or a switching element being stuck in some erroneous state due to a fault occurring inside it. The effect of this class of faults is missing packets, i.e. no packet is received when one or more is expected. This occurs when the packets fail to arrive within some specified time which is larger than the normal packet transmission time. The packets are held up somewhere in the network due to faults. The other class of faults affects the integrity or interpretation of data in a packet. A stuck-at fault occurring in a link, for example, can cause packets to be misdirected due to an erroneous destination address bit being used. Or, the occurrence of an internal fault in a switching element can cause the address bit to be interpreted wrongly. If the full address space available is not used, a fault in a link need not necessarily cause packets to be misdirected. We may get instead, erroneous data in the received packet.

Figure 3 shows the various faulty router configurations. The dashed lines indicate the connections that are operable while the dark lines indicate the connection being permanently fixed. Case (a) in the figure is for faults that prevent packet transmission through an input port while case (b) is for faults that prevent packet transmission through an input-output port pair. The third case is for faults that cause an input port to be permanently connected to an output port. Note that a connection is said to be good if packets can be sent through it using the asynchronous communication protocol. Hence the case of

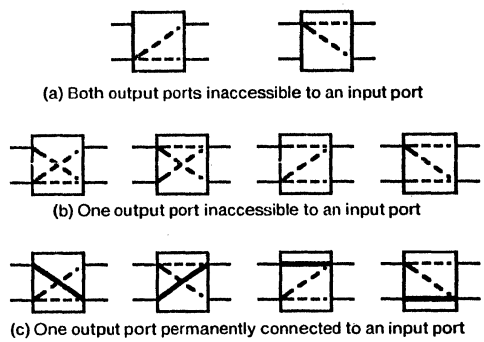


Figure 3. Faulty Configurations of the 2x2 Router

corrupted data in a received packet at the proper destination is not shown. Neither is the case where the faulty router sends packets to two output ports considered. This is because in order that the router be able to send packets to two output ports, the fault must make it behave like a fork in sending the packet arrival signal out and like a merge in receiving the packet acknowledge signals. We assume that the design of the router is such that this will cause packet transmission to hang.

Test Strategy

The test strategy proposed here involves checking that packets can be sent through the input ports of every router in the network. To do this, the two test phase approach discussed in [7] is used. Since each phase involves a single test as each router is tested by the transmission of a single packet through it, we call the two phases tests 1 and 2. In test 1, each router input port is checked to see if it can be connected directly to the corresponding output port -- input port 0 to output port 0 and input 1 to output port 1. All routers are set up as in (e) of Figure 1 by using the proper destination addresses. Test 2 checks to see if each input port can be connected to the output port across from it -- input port 0 to output port 1 and input port 1 to output 0. This means that all routers are set up as in (f) of Figure 1. Note that in both cases packet transmission through each input port of the router is independent of each other.

In each test, exactly one specially formatted packet is sent from a source to a destination and there are exactly N such source-destination pairs that will be communicating concurrently. Hence if the network is working properly, each processor will send and receive exactly one packet. The format of the packet used depends to some extent on the router implementation. In any case, the source address is also included in the packet. The source address in the received packet is checked to make sure that the packet received is sent by the expected source. Some test bit patterns are also sent to check for stuck-at faults in the data bits. This test pattern is composed of two bytes, the first of which is an alternating sequence of 0's and 1's, while the second is the same sequence rotated by 1. The width of the test pattern is the same as the width of the data path of the byte serial transmission used. In those implementation where the Last Byte bit is used, the length of the packet is also included to check for stuck-at faults in that bit. If the packet length is fixed during the test, the length information need not be sent as data in the packet.

With this test strategy, if case (a) of Figure 3 occurs, the effect will be two missing packets - one for each test. In this case a stuck-at fault occurring in the attached input link of the router cannot be distinguished from one that occurs inside the router. Case (b) will have the effect of a missing packet in one of the tests. In case (c), the effect will be a missing packet and more than one packet received by a destination in one test. If a fault occurs that causes packets of the wrong lengths to be sent, the destination will see a shortened packet and it as well as some other destinations may receive additional packets.

Fault Diagnosis

If a fault is detected in the test, the fault diagnosis strategy described in [7] is used to identify the faulty router. However, it is important to note that the strategy given in [7] deals only with single bit input lines, while in this paper we are dealing with multiple signal lines carrying bytes observing some asynchronous communication protocol. Hence, instead of getting faulty output patterns, we get one of the effects described earlier. For example, the logically unidentified output value (open circuit) "-" and logically erroneous value (two independent logic signals being tied together) "ϕ" correspond to the effects of missing packets and multiple packets, respectively.

Both Output Ports Inaccessible to an Input Port

Since in this case a fault occurring in a link cannot be distinguished from one that occurs in the connected router, the fault is assumed to be in a link. Once the link is located, further tests are then done to locate the actual fault. Since a link is on exactly one path for each test, the set of links that are on the faulty path can be identified as follows. Each link is identified by the number of the input port that it is connected to. In test 1, if $P_s\ P_{s-1} \ldots P_1\ P_0$ is the link that is connected to the source processor then the link at the output side of the i-th stage is $P_0\ P_1 \ldots P_i\ P_s \ldots P_{i+1}$, where $0 \le i \le s$. Similarly for test 2, the link at the output side of the i-th stage is $\bar{P}_0\ \bar{P}_1 \ldots \bar{P}_i\ P_s \ldots P_{i+2}\ P_{i+1}$. In test 1, the link at the output side of the i-th stage is identified by rotating, to the right by 1, the righmost s-i+1 bits of the link number of the previous stage while in test 2, the process is the same except that the least significant bit of the s-i+1 bits is always complemented after the rotation. To identify the "faulty" link, the source addresses for the two tests are obtained from the destination addresses of the processors that did not receive a packet. Note that the source-destination addresses are related as follows: for test 1, the addresses are bit reversals of each other and for test 2 they are the complement of the bit reversals of each other. The set of links of the path is determined for each test. The "faulty" link is the intersection of the two link sets. Two tests are required for locating the "faulty" link and to determine if the fault is in the link or the router, one more test is necessary. This test involves checking to see if packet arrivals and packet acknowledgments can be detected at the input port of the router. The absence of the former means that the link is bad and the absence of the latter means that the router is bad.

An Output Port Inaccessible to an Input Port

For this case, there is only one destination that receives no packets for both tests. To locate the faulty router, a binary search is done. The objective of the search is to identify the stage in which the router is located. Knowing the path and the stage, the router can be pinpointed. A search tree with each of the stages of the network as leaves is constructed. Starting at the root, the stages 0 to $\frac{s}{2}$ (if s is even) or $\frac{s+1}{2}$ (if s is odd) will be in the left subtree and the rest of the stages in the right subtree. The left subtree is set up to be of the same configuration as the test in which the fault occurs while the right subtree is set up in the same configuration as the other test. The network is then tested. If no faulty response is obtained then the fault is in the right subtree; otherwise it is in the left subtree. This process is repeated for the faulty subtree until the stage is located. The number of tests required is of the order log(log N).

An Output Port Permanently Connected to an Input Port

In this case one of the tests will give two faulty responses - missing packet and multiple packets at two distinct destinations. From the test at which the fault occurs, the fault type can be determined - for test 1, the left two cases of (c) in Figure 3 and for test 2 the other two cases. At most 2 tests are required to locate the router.

Erroneous Packet Length

For this case, the destination will receive a shorter than expected packet. Since the fault may occur in a link or a router, depending on whether the Last Byte bit is used or not, the situation is similar to that of both output ports being inaccessible to an input port. A fault has the effect of sending fragments of the packets through the network. More than one destination may receive multiple packets; all but one of these will receive a normal packet followed by at least one erroneous packet. The remaining one destination will receive one shortened packet followed possibly by some erroneous packets. The faulty path is identified by the latter since it is the proper destination and is guaranteed to receive at least the destination address byte of the packet. Each test will give a faulty path and the intersection of the set of links or routers in the two paths is the faulty link or router.

Summary

A test strategy for packet switching networks has been presented. The strategy is developed for byte serial packet communication using an asynchronous communication protocol. It has been shown that the effect of a single stuck-at fault can be classified into misdirected packets, missing packets, corrupted data in packets, or multiple packets. There are basically two types of faults - those that prevent packet transmission and those that affect only the integrity or the interpretation of the data sent in the packet. The presence of a fault in the network will show up as one of 4 cases - both output ports of the switching element not accessible to an input port, an output port not accessible to an input port, an input port permanently connected to an output port and erroneous packet length. An approach for fault location is also presented and it is shown that the number of tests required is either constant or of the order of log (log N).

Acknowledgements

The author is indebted to Professor Jack Dennis as well as to Bill Ackerman, Andy Boughton, Dean Brock and Ken Todd for their criticisms, comments and help in the preparation of the paper.

References

[1] J. Dennis, G. Boughton, and C. Leung, "Building Blocks for Data Flow Prototypes," Proceedings of 1980 Symposium on Computer Architecture, LaBaule, France, (May, 1980), pp. 1-8.

[2] C. Mead and L. Conway, Introduction to VLSI Systems, Addison-Wesley, Reading, MA, (1980), 396 pp.

[3] J. Narraway and K-M. So, "Fault Diagnosis in Inter-processor Switching Networks," Proceedings of the IEEE International Conference on Circuits and Computers, ICCC 80, (October, 1980), pp. 750-753.

[4] A. Tripathi and G. Lipovski, "Packet Switching in Banyan networks," Proceedings of the 6th Annual Symposium on Computer Architecture, (April, 1979), pp. 160-167.

[5] C. Wu and T. Feng, "On a Class on Multistage Interconnection Networks," IEEE Transactions on Computers, (August, 1980), pp. 694-702.

[6] C. Wu and T. Feng, "The Reverse-exchange Interconnection Network," IEEE Transactions on Computers, (September, 1980), pp. 801-811.

[7] C. Wu and T. Feng, "Fault-diagnosis for a Class of Multistage Interconnection Networks," IEEE Transactions on Computers, (October, 1981), 743-758.

ON FAULT-DIAGNOSIS OF SOME MULTISTAGE NETWORKS

Tse-yun Feng and I-pieng Kao
Department of Computer and Information Science
The Ohio State University
2036 Neil Avenue Mall
Columbus, OH 43210

Abstract -- It was shown previously that four tests are required in order to detect single faults and to locate single link stuck faults for a class of multistage interconnection networks. In this paper we show that only three tests are actually necessary and sufficient both to detect single faults and to locate single link stuck faults. The test schemes described achieve the least number of tests required for detecting and locating such faults.

Introduction

In a paper previously presented at this conference [1] it was shown that four tests are required in order to detect single faults and to locate single link stuck faults for a class of multistage interconnection networks. This paper is to show that only three tests are actually necessary and sufficient to detect and locate such faults.

Fault Model

The fault model described here applies to a class of multistage interconnection networks [2], although the discussion is mainly on the baseline network. The interconnection network discussed in this paper consists of $N\log_2 N/2$ switching elements where N is the number of inputs and $N\log_2(N+1)$ links. Each switching element has two inputs and two outputs, and it can have only two valid states as shown in Fig. 1. The faulty and the valid states constitute the 16 possible states of the switching elements listed in Table I. The faults to be diagnosed for a switching element in valid states S10 and S5 are listed in Tables II and III, respectively, where "-" means the logically undefined output and "ϕ" means logically erroneous output resulting from the simultaneous input of 0 and 1. It is assumed that - and ϕ can be differentiated from each other and from 0 and 1 during the test. The links of the network can have stuck kind of faults (Tables II and III).

Detection of Single Faults

According to the fault model, a fault in an interconnection network can be either a link fault or a switching element fault. A link fault can be either a stuck-at-0 or stuck-at-1. A switching element fault can be considered to be the malfunction of the switching element from its valid states.

Theorem 1: Three tests are necessary and sufficient for detecting single faults in a baseline network constructed of switching elements with two valid states S10 and S5.

Proof: Consider one switching element with inputs x_1, x_2 and outputs $\hat{x}_1$, $\hat{x}_2$ first. To detect a single fault we need at least two tests, one for switching element at state S10 and the other at S5. From Tables II and III it can be seen that the test $(x_1, x_2) = (0, 1)$ or $(1, 0)$ can detect all types of S10 and/or S5 malfunctions, but the test $(x_1, x_2) = (1, 1)$ or $(0, 0)$ cannot. However, any combination of the two tests [(0, 1) for both S10 and S5, (1, 0) for both S10 and S5, (0, 1) for S10 and (1, 0) for S5, or (1, 0) for S10 and (0, 1) for S5] is not sufficient to detect all the link faults. In other words, one additional test is needed during either S10 or S5 test. Therefore, at least three tests are required. Let (0, 1) and (1, 0) tests be used for switching element functioning at S10 so that any single link fault or the S10 malfunction can be detected. Then, let (0, 1) test be used for the switching element functioning at S5 to detect the S5 malfunction. Thus, three tests are necessary and sufficient to detect single faults for the network.

Detection and Location of Link Faults

There are two test phases as shown in Fig. 2. During Phase 1, the input terminals, labelled in binary numbers, with even or odd number of 1's receive input vector 01 or 10 (or alternately 10 or 01), respectively. Based on the result of Phase 1 test, all input terminals then receive either all 1's or all 0's (Fig. 2.b or 2.c) during Phase 2 test. Fig. 3 shows an alternate test scheme.

Theorem 2: Independent of network sizes three tests are necessary and sufficient for detecting and locating single link faults in a baseline network constructed of switching elements with two valid states S10 and S5.

Proof: The necessary condition is quite obvious because it requires at least two tests (Phase 1) in order to detect the link faults and at least one additional test to locate the fault. The sufficient condition can be proved due to the fact that during Phase 1 test the type of link stuck fault is determined and unique faulty path can be computed between the faulty output and its input [1], thus, only one subsequent test is required to determine the other faulty path during Phase 2 so that the intersection of these two paths gives the faulty link.

Fig. 4 gives an example of the detection and

0190-3918/82/0000/0099$00.75 © 1982 IEEE

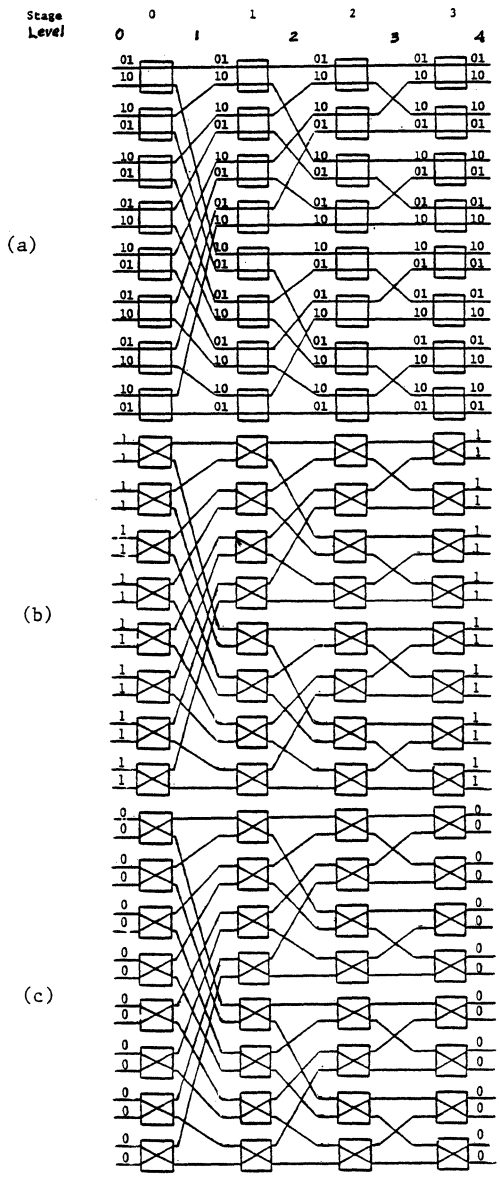


Fig. 2. Fault-free response. (a) Phase 1 test. (b) Phase 2 test for stuck-at-0 fault. (c) Phase 2 test for stuck-at-1 fault.

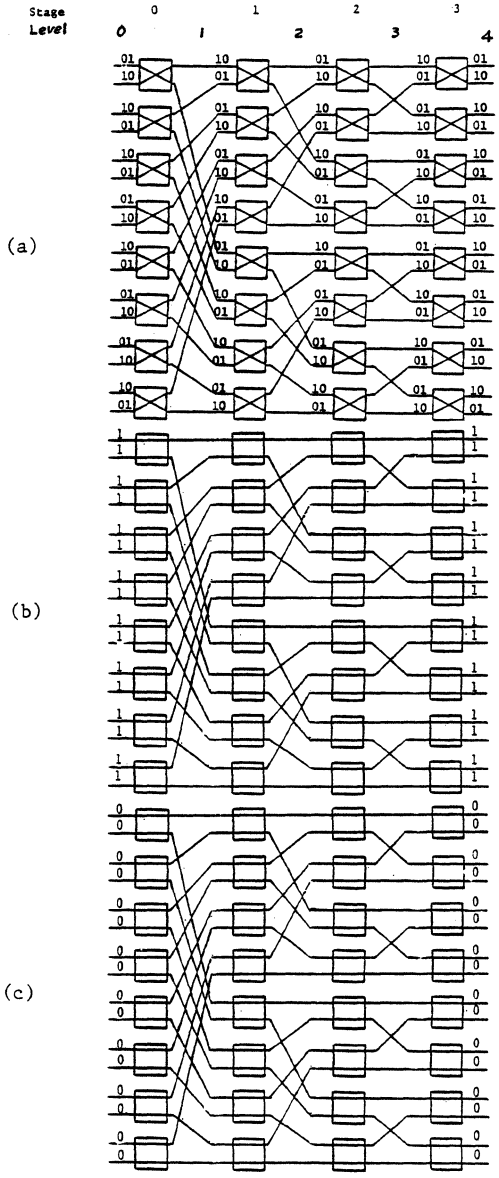


Fig. 3. Fault-free response of an alternate test scheme. (a) Phase 1 test. (b) Phase 2 test for stuck-at-0 fault. (c) Phase 2 test for stuck-at-1 fault.

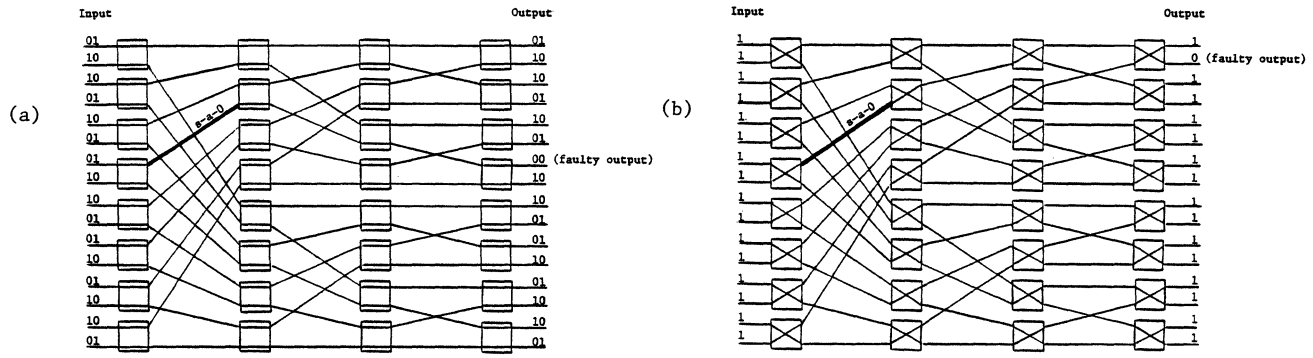


Fig. 4. Locating the link stuck fault. (a) Phase 1 test. (b) Phase 2 test.

location of link faults. Since Phase 1 test identifies the link fault to be a stuck-at-0 type, every input terminals then receives a 1 during the Phase 2 test. From these two tests the possible faulty links are identified to be (6, 6, 3, 5, 6) for Phase 1 and (7, 6, 2, 0, 1) for Phase 2. Intersecting these two sets we find that the link stuck-at-0 fault is located at link 6 of level 1.

Discussion

The test schemes described in this paper achieve the least number of tests required for detecting single faults and locating single link stuck faults for a class of multistage interconnection networks. It is obvious that additional tests are required in order to determine the type and location of switching element faults.

References

[1] C. Wu and T. Feng, "Fault-Diagnosis for a Class of Multistage Interconnection Networks", Proc. ICPP (August 1979), pp. 269-278.

[2] C. Wu and T. Feng, "On a Class of Multistage Interconnection Networks", IEEE Trans. on Computers (August 1980), pp. 694-702.

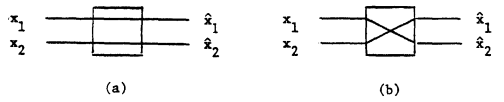


Fig. 1. A 2x2 switching element. (a) Direct connection (b) Crossed connection

Table I. Set of 16 States and the Related Symbolic Representation of a 2×2 Switching Element

State Name	Switching Element Symbol	Crosspoint Switching Matrix Symbol
S_0		(0000)
S_1		(0001)
S_2		(0010)
S_3		(0011)
S_4		(0100)
S_5		(0101)
S_6		(0110)
S_7		(0111)
S_8		(1000)
S_9		(1001)
S_{10}		(1010)
S_{11}		(1011)
S_{12}		(1100)
S_{13}		(1101)
S_{14}		(1110)
S_{15}		(1111)

Table II. Faults, Test Inputs, and Outputs in Valid State S10

	Fault	Test x_1 x_2	Output: Normal $\hat{x}_1$ $\hat{x}_2$	Output: Faulty $\hat{x}_1$ $\hat{x}_2$
Part I: Link Stuck Fault	$x_1^0, \hat{x}_2^0$	1 0	0 1	0 0
		1 1	1 1	1 0
	$x_1^1, \hat{x}_2^1$	0 0	0 0	0 1
		0 1	1 0	1 1
	$x_2^0, \hat{x}_1^0$	0 1	1 0	0 0
		1 1	1 1	0 1
	$x_2^1, \hat{x}_1^1$	0 0	0 0	1 0
		1 0	0 1	1 1
Part II: Switching Element Fault	S_5-S_0	0 1	1 0	- -
		1 0	0 1	- -
		0 0	0 0	- -
		1 1	1 1	- -
	S_5-S_1	0 1	1 0	1 -
		1 0	0 1	0 -
		0 0	0 0	0 -
		1 1	1 1	1 -
	S_5-S_2	0 1	1 0	- 1
		1 0	0 1	- 0
		0 0	0 0	- 0
		1 1	1 1	- 1
	S_5-S_3	0 1	1 0	1 1
		1 0	0 1	0 0
	S_5-S_4	0 1	1 0	- 0
		1 0	0 1	- 1
		0 0	0 0	- 0
		1 1	1 1	- 1
	S_5-S_6	0 1	1 0	- ϕ
		1 0	0 1	- ϕ
		0 0	0 0	- 0
		1 1	1 1	- 1
	S_5-S_7	0 1	1 0	1 ϕ
		1 0	0 1	0 ϕ
	S_5-S_8	0 1	1 0	0 -
		1 0	0 1	1 -
		0 0	0 0	0 -
		1 1	1 1	1 -
	S_5-S_9	0 1	1 0	ϕ -
		1 0	0 1	ϕ -
		0 0	0 0	0 -
		1 1	1 1	1 -
	S_5-S_{10}	0 1	1 0	0 1
		1 0	0 1	1 0
	S_5-S_{11}	0 1	1 0	ϕ 1
		1 0	0 1	ϕ 0
	S_5-S_{12}	0 1	1 0	0 0
		1 0	0 1	1 1
	S_5-S_{13}	0 1	1 0	ϕ 0
		1 0	0 1	ϕ 1
	S_5-S_{14}	0 1	1 0	0 ϕ
		1 0	0 1	1 ϕ
	S_5-S_{15}	0 1	1 0	ϕ ϕ
		1 0	0 1	ϕ ϕ

Table III. Faults, Test Inputs, and Outputs in Valid States S5

	Fault	Test x_1 x_2	Output: Normal $\hat{x}_1$ $\hat{x}_2$	Output: Faulty $\hat{x}_1$ $\hat{x}_2$
Part I: Link Stuck Fault	$x_1^0, \hat{x}_1^0$	1 0	1 0	0 0
		1 1	1 1	0 1
	$x_1^1, \hat{x}_1^1$	0 0	0 0	1 0
		0 1	0 1	1 1
	$x_2^0, \hat{x}_2^0$	0 1	0 1	0 0
		1 1	1 1	1 0
	$x_2^1, \hat{x}_2^1$	0 0	0 0	0 1
		1 0	1 0	1 1
Part II: Switching Element Fault	S_{10}-S_0	0 1	0 1	- -
		1 0	1 0	- -
		0 0	0 0	- -
		1 1	1 1	- -
	S_{10}-S_1	0 1	0 1	1 -
		1 0	1 0	0 -
		0 0	0 0	0 -
		1 1	1 1	1 -
	S_{10}-S_2	0 1	0 1	- 1
		1 0	1 0	- 0
		0 0	0 0	- 0
		1 1	1 1	- 1
	S_{10}-S_3	0 1	0 1	1 1
		1 0	1 0	0 0
	S_{10}-S_4	0 1	0 1	- 0
		1 0	1 0	- 1
		0 0	0 0	- 0
		1 1	1 1	- 1
	S_{10}-S_5	0 1	0 1	1 0
		1 0	1 0	0 1
	S_{10}-S_6	0 1	0 1	- ϕ
		1 0	1 0	- ϕ
		0 0	0 0	- 0
		1 1	1 1	- 1
	S_{10}-S_7	0 1	0 1	1 ϕ
		1 0	1 0	0 ϕ
	S_{10}-S_8	0 1	0 1	0 -
		1 0	1 0	1 -
		0 0	0 0	0 -
		1 1	1 1	1 -
	S_{10}-S_9	0 1	0 1	ϕ -
		1 0	1 0	ϕ -
		0 0	0 0	0 -
		1 1	1 1	1 -
	S_{10}-S_{11}	0 1	0 1	ϕ 1
		1 0	1 0	ϕ 0
	S_{10}-S_{12}	0 1	0 1	0 0
		1 0	1 0	1 1
	S_{10}-S_{13}	0 1	0 1	ϕ 0
		1 0	1 0	ϕ 1
	S_{10}-S_{14}	0 1	0 1	0 ϕ
		1 0	1 0	1 ϕ
	S_{10}-S_{15}	0 1	0 1	ϕ ϕ
		1 0	1 0	ϕ ϕ

Fault Tolerance Analysis of Several Interconnection Networks

John Paul Shen

Department of Electrical Engineering
Carnegie-Mellon University
Schenley Park, Pittsburgh, PA 15213

ABSTRACT -- *A β-network is an interconnection network composed of 2 x 2 switching elements called β-elements. β-networks can be used as multicomputer communication networks. In a previous paper, a theoretical framework facilitating the fault-tolerance analysis of β-networks was developed. In this paper, the analytical results from the earlier work are applied to the analysis of several well-known β-networks. These β-networks include the shuffle-exchange network, the double-tree network, the indirect binary n-cube network, and the Benes rearrangeable switching network. A formal technique for describing topological structure of a β-network, and some useful techniques for analyzing complex β-networks are also presented.*

1. INTRODUCTION

A class of interconnection networks called β-networks has been proposed as intercomputer communication networks (ICN) for multicomputer systems [1]. An n x n β-network is an interconnection network which provides connections from n input terminals to n output terminals and is composed of 2 x 2 switching elements called β-elements. Each β-element can be set to one of two states, namely the "through" (T) state or the "cross" (X) state, corresponding to the two possible permutations of its input terminals. The n computing units of a multicomputer correspond to both the n input terminals and the n output terminals. Hence, the n input links and the n output links of the β-network are considered to be identical and have been defined as the n terminal links of the β-network [1].

In a previous paper [2] a theoretical framework facilitating the fault-tolerance analysis of β-networks was developed. This paper constitutes a sequel to that work. The analytical results from the earlier work are applied to the analysis of several well-known β-networks. These β-networks include the inverse shuffle-exchange network [3], the double-tree network [4], the indirect binary n-cube network [5], and the rearrangeable switching network [6].

Pertinent results from [2] are now summarized here. A fault model was specified which allows β-elements to be stuck in either of their two normal states, i.e., stuck-at-through (s-a-T) or stuck-at-cross (s-a-X). A new connectivity property called dynamic full access (DFA) was introduced which serves as the criterion for fault tolerance in β-networks. A β-network has the DFA property if each of its inputs can be connected to any one of its outputs via a finite number of passes through the β-network. A fault in a β-network is a collection of β-element stuck-at faults. A fault is said to be critical if it destroys the DFA property of the β-network. A minimal critical fault is a critical fault none of whose proper subsets constitutes a critical fault. A β-network with DFA is k-fault tolerant or k-FT if the failure, either s-a-T or s-a-X, of any k or fewer β-elements does not destroy DFA. The largest k for which a β-network is k-FT is called the fault-tolerance (FT) parameter of the β-network.

A graph model for analyzing β-networks called a β-graph was introduced in [2]. The labeled β-graph of a β-network is a labeled directed graph with vertices representing the β-elements, and edges representing the links of the β-network. An edge is labeled and called a terminal edge if it corresponds to a terminal link of the β-network, otherwise it is not labeled and is called an intermediate edge. An unlabeled β-graph, or simply a β-graph, is a labeled β-graph with all its edge labels deleted. Figure 1 illustrates the labeled β-graph of a β-network called the 8 x 8 inverse shuffle exchange (ISE) network [3] which connects eight computing units {0,1, . . . ,7}. Each computing unit is implicitly represented by a terminal edge in the β-graph. Usually the terminal edges are labeled with the indices of the associated computing units as depicted in Fig. 1. Each β-element in a β-network is modeled by a vertex with two incoming and two outgoing edges in the corresponding β-graph. A β-element stuck-at fault can be modeled by the splitting of the corresponding vertex into two subvertices, each with one incoming and one outgoing edge. Furthermore, it is easily seen that a β-network has the DFA property if and only if the corresponding β-graph is strongly connected.

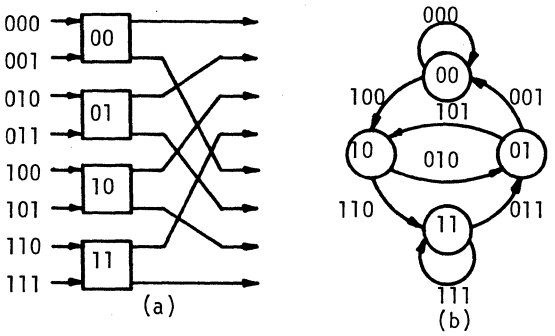


Fig. 1. (a) The 8x8 inverse shuffle exchange (ISE) network; (b) Its labeled β-graph.

Given an n x n β-network N, a connection of N is a one-to-one mapping from the n input to the n output terminals, and can be represented by a permutation of n elements. The state of N is

0190-3918/82/0000/0102$00.75 © 1982 IEEE

determined by the states of its z β-elements. If $s_i = s(b_i) \varepsilon\{T,X\}$ denotes the state of the β-element b_i, then a state of N is represented by a z-tuple $s(B) = s(b_1,b_2,...,b_z) = (s_1, s_2,...,s_z)$. If some of the β-elements are not specified, or their states are not of interest, then we can characterize s(B) by the partial state, $s(B) = (s_1, s_2,...,s_z)$, where $s_i \varepsilon\{T,X,d\}$, and d represents an unspecified or don't care state. A connection p of N is realizable if there exists a state s of N such that by setting N to state s, the one-to-one mapping specified by p is established. It is possible that a connection of N can be realized by more than one state of N.

A second network parameter based on the intercomputer communication delays was introduced in [7] as a measure of the performance of a β-network. This parameter d is obtained by considering the communication delays between all pairs of computing units and choosing the maximum or worst case value of these delays. The above definition was formalized in [7] by making use of the β-graph model of β-networks. The edge-distance, or simply distance, from edge i to edge j in a β-graph is the number of intermediate vertices in the shortest directed path having edges i and j as its first and last edges, respectively. The edge-diameter, or simply diameter, of a β-graph is the longest distance between any two edges of the β-graph. The communication delay (CD) parameter d of a β-network is the diameter of its β-graph.

The CD parameter d thus indicates the worst possible delay, measured in terms of the number of β-elements, between any pair of computing units in the multicomputer system. Meanwhile, the FT parameter k indicates the maximum number of β-elements in a β-network, whose failures, either s-a-T or s-a-X, do not destroy the DFA property of the network. In this paper, the FT and CD parameters are derived for several well-known β-networks. Section 2 presents a formal technique for describing the topological structure of a β-network, and some useful techniques for the analysis of complex β-networks. Sections 3 and 4 contain the analysis of the inverse shuffle-exchange (ISE) network and the double-tree (DOT) network, respectively. It is shown that both the ISE and the DOT networks are non-fault tolerant, i.e., their FT parameters are k = 0. However, modified versions of the ISE and DOT networks are presented which are fault tolerant. It is also shown that the shuffle-exchange (SE) network possess the same FT and CD parameters as that of the ISE network. Section 5 analyzes the indirect binary n-cube (nIBC) network. It is shown that the nIBC network exhibits very desirable FT and CD parameters. It is further shown that the flip network used in the Staran SIMD parallel processor [8] and the omega network [9] also possess the same FT and CD parameters as the nIBC network. In Sec. 6 the CD parameter and bounds for the FT parameter of Benes' rearrangeable switching (BRS) network [6] are presented. A conjecture of the actual FT parameter is also included.

2. GENERAL ANALYSIS TECHNIQUES

A β-network is a collection of β-elements interconnected by fixed links. A β-element can be viewed as containing flexible links which can be programmed, i.e., set to certain states, to provide desired communication paths from the inputs to the outputs of the β-element. This section develops a formal technique for concisely describing the topological structure of β-networks. Large and complex β-networks are typically constructed from smaller β-networks; the smallest being the 2 x 2 β-element. Frequently, many identical subnetworks are connected to form a large network with a regular interconnection structure. Two very general interconnection methods for β-networks are now discussed.

The dimension of an n x n β-network N is |N| = n. By numbering the inputs and outputs from top to bottom, the set of inputs of N can be denoted by two ordered sets $I(N) = (I_1,I_2,...,I_n)$ and $O(N) = (O_1,O_2,...,O_n)$, respectively. In an interconnected multicomputer system the inputs and outputs of N coincide, hence we can say that I(N) = O(N), which means that I_i is connected to O_i for i = 1,2,...,n. β-networks with the same dimension can be connected to form a cascade or series network; we now define this concept formally.

A β-network N is a cascade of β-networks $N_1,N_2,...,N_y$, denoted $N = N_1*N_2*...*N_y$, if $|N| = |N_1| = |N_2| = ... = |N_y|$, and $I(N) = I(N_1)$, $O(N_1) = I(N_2),...,O(N_{y-1}) = I(N_y)$, $O(N_y) = O(N)$. If all the subnetworks are identical, i.e., if $N_1 = N_2 = ... = N_y$, then we will write $N = N_1^y$. A cascaded β-network and all its subnetworks must have the same dimension. Many well-known β-networks are cascades of other networks.

Given two ordered sets of terminals, $X = (X_1, X_2,...,X_a)$ and $Y = (Y_1, Y_2,..., Y_b)$, the union of X and Y, denoted X U Y, is another ordered set of terminals $Z = (Z_1, Z_2,...,Z_c)$, such that c = a + b, and $Z_i = X_i$ for i = 1,2,...,a, and $Z_i = Y_{i-a}$ for i = a + 1, a + 2,..., a+b. We can now define another interconnection method involving the vertical composition or juxtaposition of networks.

A β-network N is a stack of β-networks $N_1, N_2,...,N_w$, denoted $N = N_1 + N_2 + ... + N_w$, if $|N| = |N_1| + |N_2| + ... + |N_w|$ and $I(N) = I(N_1)$ U $I(N_2)$ U ... U $I(N_w)$ and $O(N) = O(N_1)$ U $O(N_2)$ U...U $O(N_w)$. If all the subnetworks are identical, i.e., if $N_1 = N_2 = ... = N_w$, then we will write $N = wN_1$. The above two interconnection methods, cascade and stack, can be combined in the construction of complex β-networks.

The interconnection topology of the n links in a β-network can be conveniently described by a permutation of its terminals. An n x n permuter π is defined here as a network consisting of n fixed links connecting two sets of n terminals. The connections realized by the permuter π can be represented by a permutation of n elements thus

$$\pi = \begin{pmatrix} t_1 & t_2 & \cdots & t_n \\ \pi(t_1) & \pi(t_2) & \cdots & \pi(t_n) \end{pmatrix}$$

where t_i is connected to $\pi(t_i)$ for i = 1, 2,...,n. An n x n permuter can be considered to be a degenerate or empty n x n β-network, and hence can be used as a subnetwork in the construction of large networks.

Since a β-network is composed of β-elements and fixed links, β-elements and permuters can be considered as the most primitive elements used in the construction of β-networks. The two interconnection methods, cascade and stack, can be

defined as operators on the primitive elements. All β-networks of interest can be formed by applying the cascade (*) and stack (+) interconnections to a set of β-elements and permuters.

Two β-networks are isomorphic if they have the same labeled β-graphs. The actual symbols used for the labeling are insignificant. There exist one-to-one correspondences between the β-elements, links and terminal links of two isomorphic β-networks. Isomorphic β-networks also have the same connecting capability and network structure. However, the diagrams representing two isomorphic β-networks may not look identical. They can be made to look identical by rearranging the positions of the β-elements without breaking and reconnecting any link.

Every β-network has a unique (unlabeled) β-graph, but a β-graph can represent more than one β-network, depending on the labeling of its terminal edges. Hence an unlabeled β-graph represents a class of β-networks all having the same unlabeled β-graph. We define two β-networks to be BG-equivalent (β-graph equivalent) if they have the same unlabeled β-graph.

All the β-networks belonging to the same BG-equivalence class can be viewed as possible realizations of the same unlabeled β-graph. Each β-network corresponds to a specific labeling of the edges of the β-graph. In a β-graph of z vertices and 2z edges, there are 2^{2z} distinct ways of labeling its edges. Hence the β-graph represents a BG-equivalence class of 2^{2z} distinct β-networks, not all of which may have practical significance. All the edges in the β-graph of a single-stage β-network are terminal edges. Hence each β-graph represents a unique single-stage β-network.

We are interested in the fault tolerance characteristics of β-networks. These characteristics depend strictly on the structure of the β-networks, and not on the computing units. It appears that a β-graph captures all the useful structural properties of a β-network, including connecting and switching properties needed for fault-tolerance analysis. Thus β-networks in the same BG-equivalence class have the same fault-tolerance properties.

Let N be a β-network with z β-elements $B = \{b_1,b_2,...,b_z\}$. A (partial) state of N, denoted $s(B) = (s_1,s_2,...,s_z)$, is an assignment of each of the z β-elements to the T, X, or d state, where $s_i \varepsilon \{T,X,d\}$ denotes the state of the β-element b_i for i = 1,2,...,z. The β-elements which have been assigned the T or X states are called the specified β-elements. The residual network of a β-network N with respect to a (partial) state s, denoted N/s, is the β-network obtained from N by replacing all the specified β-elements of s by fixed links according to the specified states. The number of β-elements in N/s is equal to the number of unspecified β-elements in s. The residual network of N with respect to a completely specified state is simply a collection of links and is not of much interest. Figure 2a depicts a β-network N with seven β-elements $B = \{b_1,b_2,...,b_7\}$ connecting eight computing units. The residual β-network of N with respect to the partial state s(B) = (d,d,d,d,X,X,T), denoted M = N/s, is illustrated in Fig. 2b. We can extend this concept to β-graphs. If G is the β-graph of a β-network N, then the residual β-graph of G with respect to a state s, denoted G/s, is the β-graph of the residual β-network N/s. Figure 2c and Fig. 2d are the β-graphs of the β-network of Fig. 2a and its residual network M = N/s of Fig. 2b, respectively. It can easily be seen that residual networks of a β-network are also legitimate β-networks.

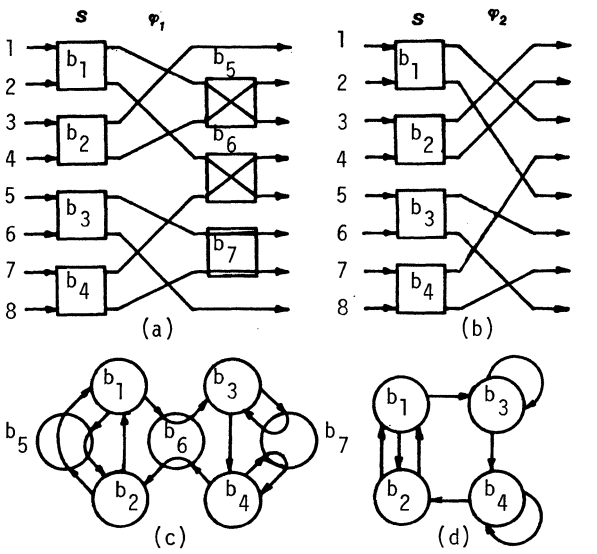


Fig. 2. Illustration of a residual network: (a) A β-network N; (b) A single-stage β-network, M = N/s, s = (d,d,d,d,X,X,T); (c) β-graph of Fig. 2a; (d) β-graph of Fig. 2b.

Many practical complex β-networks are constructed by systematically interconnecting a collection of smaller β-networks. By knowing the properties of the subnetworks and their interconnections we often can draw conclusions about the entire network. This is the approach taken here.

Theorem 1: If two β-networks N_1 and N_2 are BG-equivalent then N_1 has DFA if and only if N_2 has DFA.

Proof: Let G_1 (G_2) be the unlabeled β-graph of N_1 (N_2). A β-network has DFA if and only if its β-graph is strongly connected. Hence N_1 (N_2) has DFA if and only if G_1 (G_2) is strongly connected. Since N_1 and N_2 are BG-equivalent, G_1 must be identical to G_2, and hence N_1 has DFA if and only if N_2 has DFA. Δ

This theorem tells us that DFA is independent of the specific labeling of terminal links, i.e., all links can be considered as intermediate or terminal links. The DFA property of a β-network can be checked by inspecting any other β-network in its BG-equivalence class.

Theorem 2: If N_1 is a residual network of N with respect to a state s, i.e., N_1 = N/s, and N_1 has DFA, then N must also have DFA.

Proof: All the terminal links of N still exist in N_1. Since N_1 has DFA, there exists a connecting path between any pair of terminal links of N_1. Since N_1 is a residual network of N, all connecting paths in N_1 exist in N. Hence there exists a connecting path between any pair of terminal links of N. Therefore N must have DFA. Δ

Frequently, a residual network of a β-network has an obvious structure which facilitates fault-tolerance analysis. Theorem 2 allows us to analyze the residual network and draw certain conclusions about the original network. Clearly, the converse of Theorem 2 is not true. Every faulty β-network is a residual network of the fault-free β-network. A critical fault produces a residual β-network which does not have DFA. A β-network has the full-access property if each input terminal can be connected to every output terminal via exactly one pass through the network [6]. The proof of the following Theorem is straight forward and is omitted.

Theorem 3: Let the β-network N be a cascade of g subnetworks $N_1, N_2,..., N_g$, i.e., $N = N_1 * N_2 *...*N_g$. The network N has full access if any one of the subnetworks $N_1,N_2,...,N_g$ has full access. Δ

Clearly any β-network N having full access must have DFA. The above theorem says that if N is a cascade of subnetworks then any one of the subnetworks having full access will guarantee full access and DFA for N. We use I to denote the identity connection

$$I = \begin{pmatrix} t_1 & t_2 & \cdots & t_n \\ t_1 & t_2 & \cdots & t_n \end{pmatrix}$$

in which every terminal t_i is connected to itself via the network N. We say a network N contains the identity connection I if there exists a complete state s of N that realizes the identity connection. I will also be used to represent the residual network N/s.

Theorem 4: Let $N = N_1*N_2*...*N_g$. If some N_i has DFA and all the other N_i's contain the identity connection I, then N must have DFA.

Proof: Assume that N_i has DFA. Let $N' = N_1*N_2*...*N_{i-1}$ and $N'' = N_{i+1}*...*N_g$ so that $N = N'*N_i*N''$. Since each N_i for i = 1,2,...,g, contains I, both N' and N'' must also contain I. Let s' and s'' be complete states of N' and N'', respectively, such that N'/s' = I and N''/s'' = I. Let s be a partial state of N that results from setting N' and N'' to the states s' and s'', respectively. Clearly $N/s = I*N_i*I = N_i$. Since N_i has DFA, N/s must have DFA. According to Theorem 2, N must also have DFA. Δ

The above theorem implies that if a β-network N has DFA and contains the identity connection I, then any cascade of multiple copies of N must also have DFA. The foregoing results will be applied to the analysis of several well-known β-networks in subsequent sections. For convenience we assume in this paper that all β-networks have dimension 2^m where m is an integer, i.e., only $2^m \times 2^m$ β-networks are considered.

3. INVERSE SHUFFLE-EXCHANGE NETWORKS

An n x n β-stack ***S*** is an n x n β-network consisting of a stack of n/2 β-elements. Many n x n β-networks contain a single stage or multiple stages of β-stacks. A well-known β-network called the n x n shuffle-exchange network or SE network, ***P***, is the cascade of an n x n permuter and an n x n β-stack [3]. The n x n permuter σ used here, which resembles the perfect shuffling of a deck of cards, is called the perfect shuffle. If the terminals are numbered from 0 to n-1, then the perfect shuffle permutation

$$\sigma = \begin{pmatrix} 0 & 1 & \cdots & n-1 \\ \sigma(0) & \sigma(1) & \cdots & \sigma(n-1) \end{pmatrix}$$

can be defined as follows

$$\sigma(i) = (2i + \lfloor 2i/n \rfloor) \bmod n, \text{ for } i = 0,1,...,n-1.$$

The inverse of an n x n β-network N, denoted N^{-1}, is another n x n β-network that is the same as N except that the direction of all the links is reversed. The input terminals become the output terminals, and vice versa. Clearly the inverse of an n x n permuter is represented by the corresponding inverse permutation. The n x n inverse shuffle-exchange network, or ISE network, is the inverse of the n x n SE network. Figure 1a depicts the 8 x 8 ISE network. Although $N*N = N^2$, $N*N^{-1}$ is not always defined, unless N is a permuter. In fact if both N_1 and N_2 are permuters, then the cascade operator * used in N_1*N_2 becomes identical to the usual composition operator in permutation groups. It can be shown that $(N^t)^{-1} = (N^{-1})^t = N^{-t}$.

For convenience, we restrict our attention to $2^m \times 2^m$ ISE networks, where m is an integer. Each β-element in an ISE network can therefore be designated by an m-bit binary number $b_m b_{m-1}...b_1$, where $b_i \varepsilon \{0,1\}$. The top β-element is designated 00...0 and the bottom β-element is designated 11...1. Following the same convention, all the 2n links can be labeled from top to bottom by (m+1)-bit binary numbers $b_m b_{m-1}...b_0$, starting from 00...0 and terminating with 11...1, as illustrated in Fig. 1a.

In the above labeling scheme each vertex $b_m b_{m-1}...b_1$ has two incoming links labeled $b_m...b_1 0$ and $b_m...b_1 1$, and two outgoing links labeled $0b_m...b_1$ and $1b_m...b_1$. When β-element $b_m b_{m-1}...b_1$ is in the T-state, connections are established from $b_m...b_1 0$ to $0b_m...b_1$ and from $b_m...b_1 1$ to $1b_m...b_1$. If it is in the X-state, these connections are reversed. The labels for β-elements can be translated directly into β-graphs to identify corresponding vertices. The binary (m+1)-tuple labels for β-network links can be used to label edges in the β-graph and, thereby implicitly identifying the computing units. The labeled β-graph of the ISE network of Fig. 1a is shown in Fig. 1b.

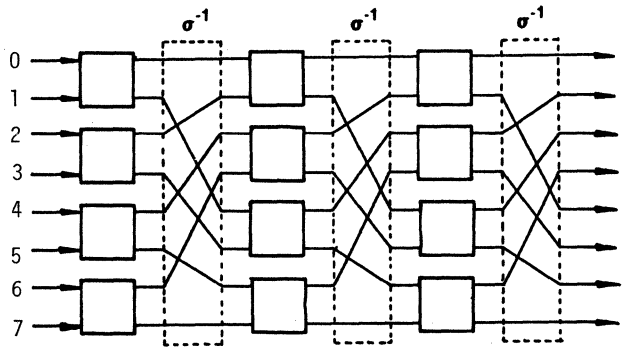


Fig. 3. The 8 x 8 omega network.

It has been shown that Pease's indirect binary m-cube is isomorphic to the omega network, which is actually a cascade of m stages of the 2^m x 2^m ISE network [10]. For example, the 2^3 x 2^3 omega network in Fig. 3 is a cascade of three identical 8 x 8 ISE networks. We know from Pease's work [5] that the indirect binary m-cube has the full access property, that is, every input terminal of the network can reach any output terminal via one pass through the network. By doing a space-to-time transformation, the i^{th} stage of the indirect binary m-cube can be mapped onto the i^{th} pass through the 2^m x 2^m ISE network. Hence if an input terminal of an indirect binary m-cube can reach any one of the output terminals in m stages, then any input terminal of the 2^m x 2^m ISE network should be able to reach any other terminal within the distance m. The communication-delay parameter d of the 2^m x 2^m ISE network must therefore be m or less. In other words, for the 2^m x 2^m ISE network, $d \leq m$. It has been shown in [7] that, given a β-network of n β-elements the lower bound for its CD parameter d is $\lfloor \log_2 n \rfloor + 1$. Since the 2^m x 2^m ISE network has 2^{m-1} β-elements, the lower bound for its d must be $\lfloor \log_2(2^{m-1}) \rfloor + 1 = m$. Hence the CD parameter of the 2^m x 2^m ISE network must be d = m. It is easy to see that the 2^m x 2^m ISE network is 0-FT. Both the top and bottom β-elements contain selfloops which constitute single critical faults [1]. The foregoing discussion leads to the following theorem.

Theorem 5: The FT and CD parameters of the 2^m x 2^m ISE network are k = 0 and d = m, respectively. Δ

The minimal communication delay of ISE networks makes them very desirable for systems requiring very fast communication. In addition, ISE networks require very simple control algorithms [5]. Clearly, a serious drawback of ISE networks is their lack of fault tolerance. We now propose a modified ISE network which is fault tolerant and still possesses the minimal communication delay.

The 2^m x 2^m modified ISE network, or MISE network, is the 2^m x 2^m ISE network with two of its links altered as follows. The top output from β-element 00...0 is connected to link 11...1 instead of to link 00...0. Similarly, the bottom output of β-element 11...1 is connected to link 00...0 instead of to link 11...1. Basically, in the MISE network, the destinations of the two original self-loop links are exchanged. Figure 4 illustrates the 2^3 x 2^3 MISE network. It can be shown that the 2^m x 2^m MISE network is 1-FT and still possesses the same minimal CD parameter of d = m. This result is stated in the following Theorem, whose proof is documented in [7].

Theorem 6: The FT and CD parameters of the 2^m x 2^m MISE network are k = 1 and d = m, respectively. Δ

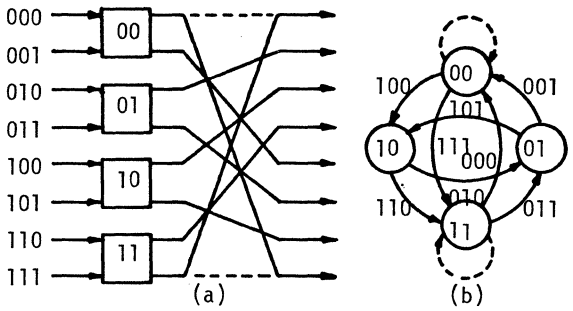


Fig. 4. (a) The 8 x 8 MISE network;
(b) Its β-graph.

The MISE networks are fault-tolerant β-networks with minimal communication delay. We have thus synthesized a fault-tolerant β-network by modifying a non-fault-tolerant β-network. This was accomplished without adding extra β-elements or increasing communication delay. The simple control algorithm used for ISE networks needs to be modified only very slightly for the MISE networks [1]. The β-graph of the 2^m x 2^m SE network is isomorphic to that of the 2^m x 2^m ISE network. Hence they both possess the same FT and CD parameters. Same is true for the modified SE network and the MISE network.

4. DOUBLE-TREE NETWORKS

The double-tree network was first proposed by Levitt, Green and Goldberg in their study of a class of β-networks called CPCU (complete permutation-complete utilization) networks [4]. A double-tree network consists of a right and a left "half." Each half of the network resembles a binary tree. The left and right trees are mirror images of each other, and each pair of mirror-image β-elements is connected by a link. Figure 5 illustrates the 2^3 x 2^3 double-tree network. The double-tree network has been investigated by three different research teams for three very different applications. Levitt et al designed the double-tree network as a fault correcting network. By cascading a double-tree network with a CPCU network, a single-fault correcting CPCU network is obtained. Hopper and Wheeler [11] considered using the double-tree network as a packet switching network for local computer networks. The double-tree network was one of two β-networks considered by Leung and Dennis [12] for implementing the distribution network of the MIT Data Flow Processor.

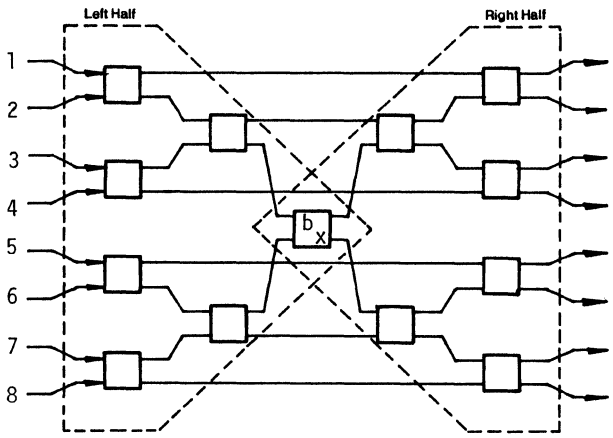


Fig. 5. The $2^3 \times 2^3$ double-tree network, or DOT network, D_3.

The structure of a $2^m \times 2^m$ double-tree network, or DOT network, denoted D_m, can be defined recursively as follows. The β- element is defined as the $2^1 \times 2^1$ DOT network. The $2^m \times 2^m$ DOT network D_m is obtained by cascading a stack of 2^{m-1} β-elements to the input side and a stack of 2^{m-1} β-elements to the output side of the $2^{m-1} \times 2^{m-1}$ DOT network D_{m-1} according to the following construction rules; see Fig. 6.

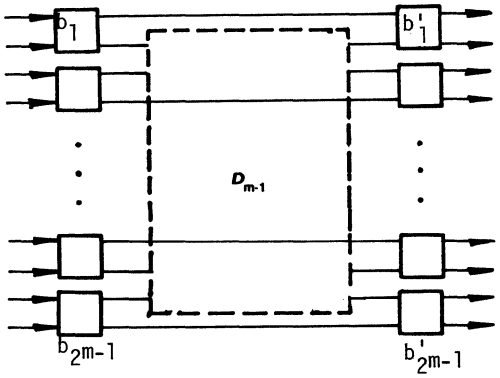


Fig. 6. The general structure of the $2^m x 2^m$ DOT network D_m.

1. Assign the labels $b_1, b_2, ..., b_{2^{m-1}}$ $(b'_1, b'_2, ..., b'_{2^{m-1}})$ to the β-elements to be cascaded to the input (output) side of D_{m-1}.

2. Connect one output of the new β-element b_j to input link I_j of D_{m-1} for $j = 1,2,...,2^{m-1}$.

3. Connect one input of the new β-element b'_j to output line O_j of D_{m-1} for $j = 1,2,...,2^{m-1}$.

4. Connect the remaining output of b_j to the remaining input of b'_j.

The inputs of the (b_j)'s and the outputs of the (b'_j)'s are the inputs and outputs, respectively, of D_m.

In general, D_m has 2^{m+1}-3 β-elements, and has 2m-1 stages of β-elements, with stage i and stage 2m-i each having exactly 2^{m-i} β-elements, for i = 1,...,m. For example, D_3 as shown in Fig. 5 has 5 stages and stages 1, 2, 3, 4 and 5 have 4, 2, 1, 2 and 4 β-elements respectively. The single β-element in the middle, i.e., stage m of D_m, is called the center β-element, and is denoted b_x. Based on the construction of D_m, a vertical symmetry and a horizontal symmetry can be identified in the network structure of D_m. The left half and the right half of D_m are symmetrical with respect to a vertical axis passing through b_x. Furthermore, the upper half and the lower half of D_m are symmetrical with respect to a horizontal axis passing thorugh b_x.

A DOT network is a multiple-stage β-network. Unlike a single-stage network, not every link of a multiple-stage network is a terminal link. Hence, not every edge in its β-graph is a terminal edge. The CD parameter d denotes the longest distance separating any pair of edges in the β-graph. Since the terminal edges represent computing units, for a multiple-stage network, it is useful to also consider the longest distance separating any pair of terminal edges in the β-graph. Hence, we define the terminal delay (TD) parameter t of a β-network as the longest distance between any pair of terminal edges in its β-graph. Consequently the TD parameter indicates the actual worst case communication delay between any two computing units. For single-stage networks t = d. Clearly, the terminal delay of a full-access β-network with t stages is equal to t.

Lemma 1: The $2^m \times 2^m$ DOT network D_m has TD parameter t = 2m-1 and CD parameter d = 4m-3.

Proof: Every input terminal of D_m can reach the center β-element b_x, and b_x can reach every output terminal. Hence D_m has full access. Since D_m has 2m-1 stages, the TD parameter must be t = 2m-1.

Let the two input links of b_x be a_1 and a_2, and the two output links of b_x be e_1 and e_2. The links e_1 and e_2 can reach every terminal link of D_m via exactly m-1 β-elements, and can reach every link of D_m via 2(m-1) or fewer β-elements. On the other hand, every link in D_m can reach e_1 and e_2 via 2m-1 or fewer β-elements. Hence the distance between any two links is at most 2(m-1) + 2m-1 = 4m-3.

To prove that d = 4m-3, we must show that there exist two links in D_m separated by the distance 4m-3. The distance from e_1 back to a_1 is equal to 2m-2, and the distance from a_1 to a_2 is equal to 2m-1. Since the upper and lower halves of D_m are joined only by the center β-element b_x, the shortest path from e_1 to a_2 must include a_1. Hence the distance from e_1 to a_2 is 2m-2 + 2m-1 = 4m-3. Therefore the CD parameter of D_m is d = 4m-3. Δ

It is easily seen that the center β-element b_x of D_m is critical. If b_x is s-a-T, then the network will be split into disjoint upper and lower halves. Hence the FT parameter of D_m is k = 0. To summarize the foregoing results we have the following theorem.

Theorem 7: Let D_m denote the $2^m \times 2^m$ DOT network. The FT parameter of D_m is k = 0. The CD parameter of D_m is d = 4m-3. The TD parameter of D_m is t = 2m-1. Δ

We now propose a modification of the DOT network to make it fault tolerant. We know that the center β-element b_x is critical; it can easily be shown to be the only critical β-element. We can remove the only single critical fault of b_x s-a-T by simply deleting the center β-element b_x. The modified DOT-network, or MDOT network, X_m is identical to the DOT network D_m except that the center β-element b_x is permanently set to the X-state. Figure 7 illustrates the $2^3 \times 2^3$ MDOT network.

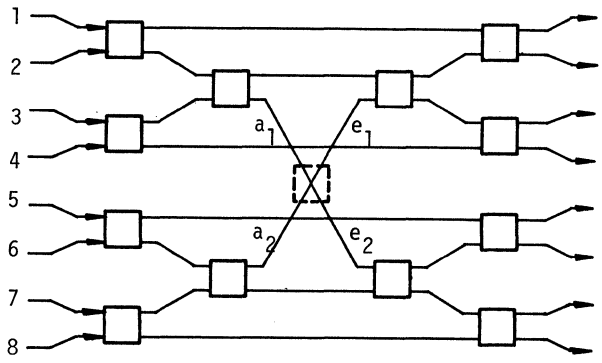


Fig. 7. The $2^3 x 2^3$ modified DOT network, or MDOT network, X_3.

Like the original DOT network, the MDOT network has full-access. However, the number of stages, and hence the TD parameter, has been reduced by one to t = 2m-2. Using the argument given in the proof of Lemma 1, it can be shown that the CD parameter for the $2^m \times 2^m$ MDOT-network is d = 4m-4, which is also one less than the CD parameter of the DOT network. Thus the MDOT network actually has better delay characteristics than the DOT network. It can be shown that the smallest minimal critical faults (MCFs) of the $2^m \times 2^m$ MDOT network consist of pairs of mirror-imaged β-elements being stuck at the same state. Hence, no single critical fault exists and the $2^m \times 2^m$ MDOT network is 1-FT. We have the following theorem, a formal proof of which can be found in [1].

Theorem 8: Let X_m denote the $2^m \times 2^m$ MDOT network. The FT parameter of X_m is k = 1. The CD parameter of X_m is d = 4m-4. The TD parameter of X_m is t = 2m-2. Δ

Interestingly, we have succeeded in modifying a non-fault-tolerant β-network to make it 1-FT, while decreasing its communication delay. As might be expected, the connecting capability of the MDOT network, in terms of the number of permutations that can be realized, is slightly less than that of the corresponding DOT network due to the absence of the center β-element. An alternate modification which does not sacrifice any connecting capability is simply to add a redundant β-element b_x' in tandem to b_x to correct the s-a-T fault of b_x.

5. INDIRECT BINARY m-CUBE NETWORKS

Various "cube" structures have been proposed for interconnecting large numbers of processors in computer systems. The binary m-cube structure has been frequently considered [5,13]. This network may be thought of as interconnecting 2^m processors which are placed at the vertices of an m-dimensional cube. Each edge of the cube represents a link connecting two processors, hence the name "binary" m-cube. One processor can be designated as the origin with an m-bit binary address 00...0. Other processors can then be identified by their corresponding coordinates in the m-dimensional space. The binary m-cube has a regular structure and is relatively simple to control. The average communication delay between any pair of processors is small.

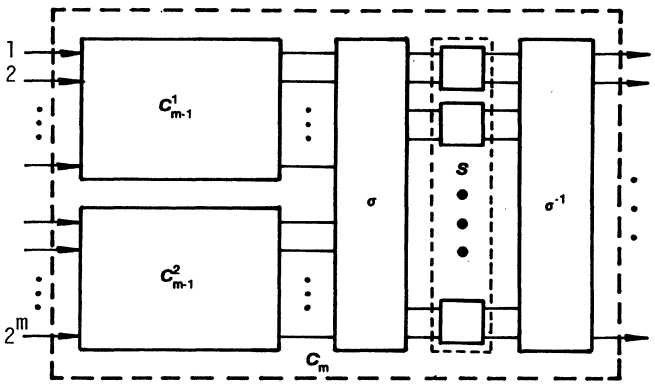


Fig. 8. The general structure of the mIBC network C_m.

The indirect binary m-cube, or mIBC network, denoted by C_m, is a $2^m \times 2^m$ β-network which is defined recursively as follows. A β-element is a 1IBC network. An mIBC network for $m \geq 2$ is constructed from two (m-1)IBC networks and a $2^m \times 2^m$ β-stack according to the following equation

$$C_m = (C_{m-1} + C_{m-1}) * \sigma * S * \sigma^{-1},$$

where σ is the perfect shuffle permuter, S is a β-stack, and σ^{-1} is the inverse perfect shuffle permuter. As before, the operator * denotes cascade or composition of permutations. Figure 8 shows the general structure of C_m. The indirect binary 3-cube C_3 is illustrated in Fig. 9.

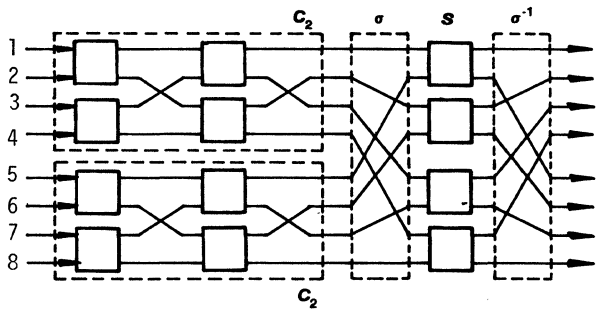


Fig. 9. The 3IBC network C_3.

The interconnections provided by the mIBC network emulate those of an m-dimensional binary cube [5]. The mIBC network has m stages of β-elements with 2^{m-1} β-elements per stage. The total number of β-elements in this network is $m2^{m-1}$. There are 2^m terminal links which correspond to the corners of the binary m-cube, while the β-elements correspond to the edges of the binary m-cube. The β-elements in any stage correspond to all the edges parallel to one of the axes. A β-element set to the X-state corresponds to a traversal of that edge in the binary m-cube. A simple algorithm, similar to those of the ISE and MISE networks exists for the control of the mIBC networks [5].

Two other well-known β-networks are actually isomorphisms of the mIBC network. The 2^m x 2^m flip network used in the Staran SIMD parallel processor manufactured by Goodyear Aerospace is structurally isomorphic to the mIBC network [8]. The two networks differ only in their control schemes. Unlike the mIBC network, in which each β-element can be individually controlled, there is only one control line for each stage of β-elements in the flip network. All the β-elements in the same stage are set to either the T-state or the X-state simultaneously, to accomplish either the "shift" or the "flip" operation. Lawrie devised a β-network called the omega network for accessing and aligning data in an array processor [9]. The omega network is typically placed between a set of processors and a set of memory modules. Processors access data in the memory via the omega network. An inverse omega network is an omega network in which the direction of all the links are reversed. It has also been shown that the 2^m x 2^m inverse omega network and the indirect binary m-cube are structurally isomorphic [10].

It is well-known that the mIBC network is a full access β-network, hence its terminal delay t is m, the number of stages. In fact there exists a unique path of length m from each terminal link to any other terminal link. Since a terminal link a_i can reach any other terminal link in distance m, a_i must be able to reach any link, terminal or intermediate, within the distance 2m-1. We now show that the CD parameter d of the mIBC network is 2m-1.

Two BG-equivalent β-networks have the same communication delay. A BG-equivalent of the mIBC network can be obtained by cyclically rotating the stages, i.e., by replacing stage 1 by stage m, stage 2 by stage 1, etc. Since the mIBC network is isomorphic to the inverse omega network, which is a cascade of identical stages, any cyclic rotation will produce an isomorphic β-network. Consequently, the links in any stage can be made the terminal links by the cyclic rotation. Hence any link whether terminal or intermediate can reach any other link within the distance 2m-1. In one pass an input terminal link can reach all the output links of the m^{th} stage, but only half of the input links to the m^{th} stage. The unreached links can be reached in a second pass. Consequently, there exist links separated by the distance 2m-1, hence the TD parameter t of the 2^m x 2^m mIBC network is m and the CD parameter d is 2m-1.

The recursive structure of the mIBC network suggests that we can determine its fault-tolerance parameter by induction. For this purpose we first develop several lemmas.

Lemma 2: Let C_m be an mIBC network. By setting all the β-elements in the m^{th} stage to the X-state we obtain a residual network C'_m having the structure illustrated in Fig. 10.

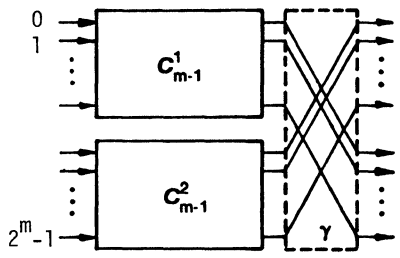


Fig. 10. A residual network C'_m of the mIBC network C_m.

Proof: The structure of C_m is defined in Fig. 8. We need to show the permuters σ and σ^{-1} and the β-elements in the m^{th} stage of C_m combine to form the permuter γ of Fig. 10. If we label the links in a stage from top to bottom with the address numbers $0,1,...,2^m-1$, then the permuter γ can be defined by the following permutation:

$$\gamma(a) = a + 2^{m-1} \pmod{2^m}$$

where $a \in \{0,1,...,2^m-1\}$ is the address of a link. If we represent a by the equivalent binary number $a_m a_{m-1}...a_1$, then we can write

$$\gamma(a_m a_{m-1}...a_1) = \overline{a}_m a_{m-1}...a_1$$

where $\overline{a}_i = 0$ (1) if $a_i = 1$ (0). Effectively, the permuter γ connects the i^{th} output of the upper C_{m-1} to the i^{th} input of the lower C_{m-1} and vice versa. The perfect shuffle σ and the inverse perfect shuffle σ^{-1} can be similarly defined as follows:

$$\sigma(a_m a_{m-1}...a_1) = a_{m-1}...a_1 a_m$$

and

$$\sigma^{-1}(a_m a_{m-1}...a_1) = a_1 a_m...a_2.$$

σ effectively corresponds to a cyclic left shift of the binary address $a_m a_{m-1}...a_1$, and σ^{-1} corresponds to a cyclic right shift of this address.

The permuting effect of a β-element can also be defined in this way. Let the permutation realized by a β-element in the T-state be denoted by E^T and that realized by a β-element in the X-state be denoted by E^X. Then

$$E^T(a_m a_{m-1}...a_1) = a_m a_{m-1}...a_1$$

and

$$E^X(a_m a_{m-1}...a_1) = a_m a_{m-1}...\overline{a}_1 \quad .$$

Now

$$\sigma * E^X * \sigma^{-1}(a_m a_{m-1}...a_1) = E^X * \sigma^{-1}(a_{m-1}...a_1 a_m)$$

$$= \sigma^{-1}(a_{m-1}...a_1 \overline{a}_m)$$

$$= \overline{a}_m a_{m-1}...a_1$$

$$= \gamma(a_m a_{m-1}...a_1)$$

from which it follows that $\sigma * E^X * \sigma^{-1} = \gamma$. Δ

Corollary 1: The residual β-network C'_m of Lemma 2 is BG-equivalent to a β-network C''_m obtained by cascading two (m-1)IBC networks C^1_{m-1} and C^2_{m-1}. Δ

If, instead of setting all the β-elements in the m^{th} stage to the X-state, we set them to the T-state, another interesting residual network is obtained.

$$\sigma^* E^T * \sigma^{-1}(a_m a_{m-1} \ldots a_1) = E^T * \sigma^{-1}(a_{m-1} \ldots a_1 a_m)$$

$$= \sigma^{-1}(a_{m-1} \ldots a_1 a_m)$$

$$= a_m a_{m-1} \ldots a_1.$$

Hence $\sigma^* E^T * \sigma^{-1} = I$, where I is the identity permuter. This leads to the following lemma.

Lemma 3: Let C_m be an mIBC network. By setting all the β-elements in the m^{th} stage to the T-state we obtain a residual network C^*_m which is a stack of two (m-1)IBC networks C^1_{m-1} and C^2_{m-1}. Δ

The two residual networks C'_m and C^*_m will be useful in computing the fault tolerance of C_m. As shown in Corollary 1, C'_m is essentially a cascade of two (m-1)IBC networks, while C^*_m is a stack of two (m-1)IBC networks. Since the mIBC network has loops of length m, and hence has elementary circuits of length m in its β-graph, its fault tolerance k must be less than m. We show next by induction, that k is indeed m-1.

Lemma 4: The FT parameter k of the mIBC network C_m is m-1, for $m \geq 2$.

Proof: The 2IBC network is clearly 1-FT because its β-graph contains a Hamiltonian circuit and no self-loop [1]. It remains to be shown that for $m > 2$, if C_{m-1} is (m-2)-FT, then C_m must be (m-1)-FT.

Let f be any set of m-1 faulty β-elements in C_m. Since C_m has m stages, there must exist a stage which does not contain any faulty β-element. We can assume without losing generality, that this fault-free stage is the m^{th} stage. Hence all the faulty β-elements of f are in the first m-1 stages, i.e., they are contained in the two subnetworks C^1_{m-1} and C^2_{m-1} of C_m. Since all the β-elements in the m^{th} stage are fault-free, they can all be set to the X-state or the T-state to obtain the residual networks C'_m or C^*_m respectively. We want to show that the C_m network containing the fault f or, equivalently, that the residual network C_m/s_f, where s_f is the partial state representing the fault f, still has DFA.

Case 1: (One subnetwork is fault-free) All m-1 faulty β-elements of f are in one subnetwork, say C^1_{m-1}. The other subnetwork C^2_{m-1} must then be fault-free. We know that a fault-free IBC network has full access. Hence C^2_{m-1} must have full access. Since C'_m is BG-equivalent to a cascade of the two subnetworks (Corollary 1), it can be concluded from Theorem 3 that C'_m/s_f must still have full access. Since C'_m/s_f has DFA and is a residual network of C_m/s_f, by Theorem 2, C_m/s_f must also have DFA. Therefore C_m is (m-1)-FT.

Case 2: (Both subnetworks are faulty) The m-1 faulty β-elements of f are distributed in both C^1_{m-1} and C^2_{m-1}. In this case each subnetwork can contain at most m-2 faulty β-elements. Since by assumption C^1_{m-1} and C^2_{m-1} are (m-2)-FT, then both C^1_{m-1}/s_f and C^2_{m-1}/s_f must still have DFA. The upper and lower halves, C^1_{m-1} and C^2_{m-1}, of the residual network C^*_m are disjoint. Since both C^1_{m-1}/s_f and C^2_{m-1}/s_f have DFA, a link in the upper (lower) half of C^*_m/s_f can reach all the links in the upper (lower) half of C^*_m/s_f. For a link in the upper (lower) half to reach all the links in the lower (upper) half, we can set the fault-free β-elements in the m^{th} stage to the X-state to obtain C'_m/s_f. Consequently any link in C_m/s_f can reach any other link in C_m/s_f. Hence C_m/s_f must have DFA and C_m must be (m-1)-FT. Δ

Theorem 9: Let C_m denote the $2^m \times 2^m$ mIBC network. The FT parameter of C_m is k = m-1. The CD parameter of C_m is d = 2m-1. The TD parameter of C_m is t = m. Δ

Unlike many of the β-networks presented earlier, mIBC networks have FT parameters which are not a constant, but a function of the size parameter m. mIBC networks exhibit the best FT and CD combination of the β-networks considered so far. If the number of β-elements of C_m is denoted by n, then the fault tolerance of C_m is approximately $\log_2 n$ and the communication delay is approximately $2\log_2 n$.

6. BENES' REARRANGEABLE NETWORKS

One of the earliest studies of connecting networks was performed by Clos on nonblocking switching networks [14]. Clos presented a class of nonblocking networks, now called Clos networks, consisting of three stages of crossbar switches. Benes presented a special class of the three-stage Clos network and showed that it is rearrangeable [6]. A network in this class has 2 x r input and 2 x r output terminals and consists of only square crossbar switches. The first and the third stages each contain r 2 x 2 crossbar switches, or β-elements, while the middle stage contains two r x r crossbar switches. It has also been proven that this network can be further decomposed by replacing each of the r x r crossbar switches in the middle stage by another three-stage rearrangeable network of the same structure, as illustrated in Fig. 11. This process can be continued until all the square crossbar switches in the network are β-elements. The resultant network is a rearrangeable $2^m \times 2^m$ β-network consisting of 2m-1 stages of β-elements. We refer to this β-network as the Benes' rearrangeable network or BRS network. The fault-tolerance properties of these BRS networks are the topic of this section.

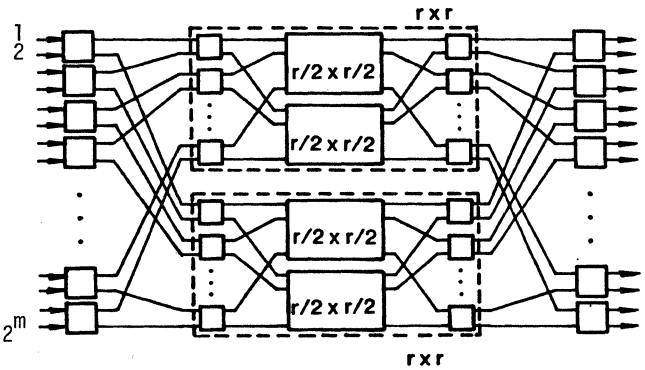


Fig. 11. Decomposition of Clos's rearrangeable network as presented by Benes.

The smallest 2 x 2 BRS network is the β-element. The 2^m x 2^m BRS network B_m for m $\geq$ 2 is defined recursively as follows.

$$B_m = S_m * \sigma^{-1} * (B_{m-1} + B_{m-1}) * \sigma * S_m$$

where S_m denotes the 2^m x 2^m β-stack, σ denotes the perfect shuffle permuter, and B_{m-1} denotes the 2^{m-1} x 2^{m-1} BRS network. Figure 12 depicts the structure of the 2^m x 2^m BRS network. The 2^3 x 2^3 BRS network B_3 is illustrated in Fig. 13. The general BRS network B_m has 2m-1 stages of β-elements and is symmetrical with respect to the middle stage. By the definition of rearrangeability, this network is capable of realizing all $2^m!$ possible connections. The BRS network is then the most powerful β-network, in terms of the connecting capability, that we have considered so far. It is also the most difficult to analyze.

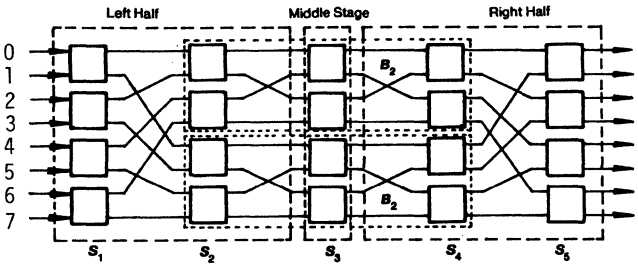


Fig. 13. The 2^3 x 2^3 BRS network, B_3.

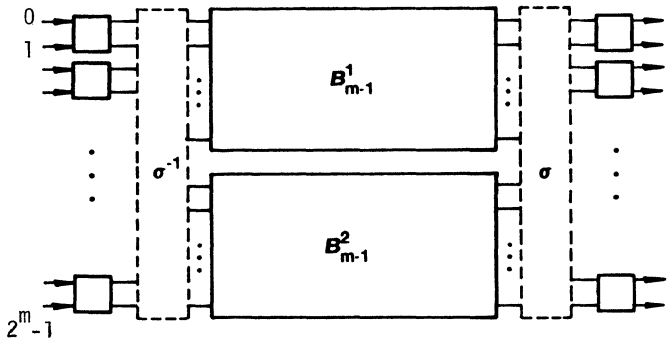


Fig. 12. The general structure of the $2^m x 2^m$ BRS network B_m.

BRS networks have been extensively studied by many researchers, including Joel [15] and Opferman and Tsao-Wu [16]. Previous work has focused on the analysis of the network complexity, and implementation of efficient algorithms for network control. Opferman and Tsao-Wu have also studied the diagnosis of faulty β-elements which are stuck at the T- or the X- states. They assume that the state of each individual β-element is not accessible, and have shown how to derive a very small set of connections, or permutations of the terminals, which correspond to an efficient set of test patterns.

As shown in Fig. 12,. the 2^m x 2^m BRS network B_m contains a stack of two 2^{m-1} x 2^{m-1} BRS networks. We denote the upper one by B^1_{m-1} and the lower one by B^2_{m-1}. Let S_i denote the stack of β-elements in the i^{th} stage of B_m whose stages are numbered 1,2,...,2m-1, from left to right. Consequently, S_m denotes the middle stage, and the network B_m is symmetrical with respect to S_m. We call the subnetwork to the left (right) of S_m the left (right) "half" of B_m. Hence the network B_m is the cascade of the left half, the middle stage S_m, and the right half as illustrated in Fig. 13. Wu and Feng designed a full-access β-network called the baseline network [17] which is isomorphic to the cascade of the left half of B_m and the middle stage S_m.

Since the BRS network is rearrangeable, it clearly has full access. Hence its terminal delay t is the number of stages in the network, which is 2m-1. The communication delay parameter t is much more difficult to derive than the terminal delay.

Lemma 5: The CD parameter of the 2^m x 2^m BRS network B_m is d = 4m-3.

Proof: The baseline network has full access, hence a terminal link of B_m can reach all the links in the right half of B_m in one pass. It can reach any link of B_m in a second pass, or within the distance d_1 = (2m-1) + (m-1) = 3m-2.

The middle stage S_m cascaded with the right half of B_m is isomorphic to the inverse baseline network. It has been shown that the inverse baseline network is isomorphic to the baseline network [17], so S_m cascaded with the right half of B_m also possesses full access. Hence any non-terminal link in the left half of B_m can reach all the terminal links within the distance d_2 = 2m-2, and can reach any link of B_m within the total distance d_3 = d_2 + (2m-2) = 4m-4. Similarly, any non-terminal link in the right half of B_m can reach at least two terminal links within the distance d_4 = m-1, and hence can reach any link of B_m within the total distance d_5 = $d_4 + d_1$ = 4m-3. Since $d_5 > d_3 > d_1$, every link must be able to reach any other link within the distance d_5 = 4m-3. Hence the communication delay d of the BRS network B_m is at most d_5. If we can show there exist two links separated by the distance 4m-3, then the communication delay parameter must be exactly d_5 = 4m-3.

Let b denote a β-element in the upper half of the middle stage S_m. Let k_o denote the upper outgoing link of b and k_i denote the lower incoming link of b. Because the structure of the BRS network, an upper outgoing link of a β-element in S_m can only reach the upper half terminal links of B_m in one pass through the network. Hence after the first pass, k_o can only reach all the upper terminal links. Furthermore, in a second pass k_o cannot reach any of the lower incoming links to the β-elements of S_m; see Fig. 13. Only in a third pass can k_o reach k_i. Hence the distance from k_o to k_i is equal to (m-1) + (2m-1) + (m-1) = 4m-3 = d_5. Therefore, the communication delay parameter d of the 2^m x 2^m BRS network is 4m-3. Δ

Some upper bounds on the fault tolerance parameter k of B_m can be easily obtained. Let the 2^m terminals of B_m be numbered 0,1,...,2^m-1 from top to bottom. If the top (bottom) β-element in

each stage is set to the T-state, a critical fault consisting of 2m-1 β-elements results, which isolates the terminal 0 (2^m -1). Hence the fault tolerance parameter k of the 2^m x 2^m BRS network must be less than or equal to 2m-2.

The 2^m x 2^m BRS network contains two major subnetworks B^1_{m-1} and B^2_{m-1}. Each of these 2^{m-1} x 2^{m-1} BRS networks in turn contains two 2^{m-2} x 2^{m-2} BRS networks, etc. We call the terminal links 0,1,...,2^{m-1}-1 of B_m its upper terminal links, and call the remaining links 2^{m-1},..., 2^m-1 the lower terminal links. In the left half of B_m any upper (lower) terminal link can only reach those links which are upper (lower) input links of the subnetworks of B_m. Consequently, if all 2^{m-1} β-elements of the middle stage S_m are set to T, the upper and lower terminal links of B_m are disconnected, and B_m is decomposed into two identical subnetworks. Hence another upper bound for k is 2^{m-1}-1. Combining the two upper bounds we can conclude that the FT parameter k of the 2^m x 2^m BRS network B_m is bounded above by min{2m-2 , 2^{m-1}-1}.

It is conjectured that k is actually equal to this upper bound.

Conjecture 1: The 2^m x 2^m BRS network B_m has FT parameter k = min{2m-2 , 2^{m-1}-1}. Δ

For m ≤ 3 the above conjecture is known to be true. If it is true for m = 4, then the conjecture can be proven inductively using the similar approach as that of Theorem 9.

Theorem 10: Let B_m denote the 2^m x 2^m BRS network. The FT parameter of B_m is k ≤ min{2m-2 , 2^{m-1}-1}. The CD parameter of B_m is d = 4m-3. The TD parameter of B_m is t = 2m-1. Δ

BRS networks appear to have fault tolerance and communication delay similar to those of mIBC networks. If the number of β-elements of B_m is n, then the fault tolerance of B_m is approximately $2\log_2 n$, and the communication delay is approximately $4\log_2 n$. BRS networks appear to be fault tolerant with respect to full access as well.

ACKNOWLEDGEMENT

The author would like to acknowledge the helpful comments and superb guidance provided by Professor John P. Hayes.

REFERENCES

1. J.P. Shen, Fault Tolerance of β-networks in Interconnected Multicomputer Systems. Ph.D. Dissertation, Dept. of Elec. Engineering, Univ. of Southern Calif., Aug. 1981. Also available as USCEE Tech. Report No. 510.

2. J.P. Shen and J.P. Hayes, "Fault tolerance of a class of connecting networks," Proc. 7th Ann. Symp. Computer Architecture, pp. 61-71, 1980.

3. H.S. Stone, "Parallel processing with the perfect shuffle," IEEE Trans. Computers, vol. C-20, pp. 153-161, Feb. 1971.

4. K.N. Levitt, M.W. Green and J. Goldberg, "A study of the data commutation problems in a self-repairable multiprocessor" Proc. Spring Joint Computer Conf., pp. 515-527, 1968.

5. M.C. Pease, "The indirect binary n-cube microprocessor array," IEEE Trans. Computers, vol. C-26, pp. 458-473, May 1977.

6. V.E. Benes, Mathematical Theory of Connecting Networks and Telephone Traffic, New York, Academic Press, 1965.

7. J.P. Shen and J.P. Hayes, "Synthesis of fault tolerant beta- networks," Proc. 12th Symp. on Fault Tolerant Computing, June 1982.

8. K.E. Batcher, "The Flip network in STARAN," Proc. Parallel Processing Conf., pp. 65-71, Aug. 1976.

9. D.H. Lawrie, "Access and alignment of data in an array processor," IEEE Trans. Computers, vol. C-24, pp. 1145-1155, December 1975.

10. D.S. Parker, "Notes on shuffle/exchange-type switching networks," IEEE Trans. Computers, vol. C-29, pp. 213-222, March 1980.

11. A. Hopper and D. J. Wheeler, "Binary routing networks," IEEE Trans. Computers, pp. 699-703, October 1979.

12. C.K.C. Leung and J.B. Dennis, "Design of a fault-tolerant packet communication computer architecture," Proc. Tenth Fault-Tolerant Computing Symp., pp. 328-335, Oct. 1980.

13. H. Sullivan and T.R. Bashkow, "A large scale homogeneous fully distributed parallel machine, I" Proc. 4th Ann. Symp. Computer Architecture , pp. 105-117, March 1977.

14. C. Clos, "A study of non-blocking switching networks," Bell Sys. Tech. Journ., Vol. 32, No. 2, pp. 406-424, March 1953.

15. A. Joel Jr., "On permutation switching networks," Bell Sys. Tech. Journ., Vol. 47, No. 5, pp. 813-822, May-June 1968.

16. D.C. Opferman and N.T. Tsao-Wu, "On a class of rearrangeable switching networks, part II: enumeration studies and fault diagnosis," Bell Sys. Tech. Journ., pp. 1601-1618, May-June 1971.

17. C.L. Wu and T. Y. Feng, "Fault-diagnosis for a class of multistage interconnection networks," Proc. Parallel Processing Conf., pp. 269-278, Aug. 1979.

A FAULT-TOLERANT CONNECTING NETWORK FOR MULTIPROCESSOR SYSTEMS

L. Ciminiera
CENS - Istituto di Elettrotecnica Generale
Politecnico di Torino
Corso Duca degli Abruzzi, 24
10129 TORINO - ITALY

and

A. Serra
Istituto Elettrotecnico
Facoltà di Ingegneria
Università di Catania
Via Andrea Doria, 6-95100 Catania - Italy

This paper presents a new interconnection network, referred to as F-network, which is able to correctly handle the communications between the connected devices, even if some nodes within the network are faulty. The routing algorithm presented in this paper provides a fast procedure for rerouting a message; hence, the redundant paths can also be used to enhance the network bandwidth. It is also shown that the rerouting properties are still valid when broadcasting is used. Analytical models show that the MTBF of a F-network is unreacheble by using several non-fault-tolerant networks in parallel. Finally, this paper presents the modularity properties of the F-network, which lead to a LSI or VLSI implementation cheaper than for a pair of parallel delta networks.

1. Introduction

One of the most promising approaches to the implementation of large multiprocessor systems is based on the use of special switched networks, connecting the processors with themselves and/or with the memory banks. In the past few years, many papers on interconnection networks have appeared in the literature [1] ÷ [7]. Almost all of them deal with the functional properties, the performance issues and the implementation of the networks discussed.

Little attention has however been paid to the fault-tolerance capabilities of interconnection networks. Fault-tolerance can be achieved in several ways; the most classical methods include the use of self-checking and correcting codes for the data transmitted through the network and the fault tolerant design of network switches and control units. These techniques rely directly on the network implementation. A third way consists in providing multiple alternative paths for the transmission of messages; in this case, the network topology and the routing and re-routing algorithms greatly influence the network fault-tolerance.

When the latter approach is used to enhance the fault-tolerance, the following criteria may be used to judge the network design:

1) the mean life time of the network should be substantially higher than in the other networks, and it should be better than that achieved by duplicating or multiplying other networks;
2) the control algorithm should be as simple as possible; furthermore, dynamic rerouting should be used, since it allows simple recovery procedure and increases the network performances;
3) the network should be able to perform all the switching functions performed by the commonest networks, without any penalty;
4) the number of active devices required for implementing the new network should be kept as low as possible.

In [11] a multiple path routing scheme for the banyan networks is presented; however it is not well suited for circuit switching, hence it partially violates criterion 3. In [8] ÷ [10], the rerouting capabilities of the ADM and IADM networks are studied; such networks are not able to reroute every message, hence they violate criterion 2. This paper presents a new network, referred to as the F-network, which performs well with respect to all the criteria listed above. Furthermore, it is modular, hence it is suitable for low-cost LSI or VLSI implementation.

In section 2, the F-network definition is presented and the routing algorithm is formulated. In section 3, the rerouting capabilities of the F-network are shown, when broadcast communications are used.

In section 4, analytical reliability models for the F and multiple delta networks are presented, and the results obtained are discussed. In section 5, the modularity properties of the F-network are illustrated and their impact on the implementation costs are discussed.

2. F-network definition

The basic element of the network presented in this paper is a switch with 4 inputs, 4 outputs and capacity 1. A block diagram of a single switch is shown in Fig. 1. Although this diagram can be used as a suggestion for implementation, it is provided hence only to explain the switch behaviour clearly.

0190-3918/82/0000/0113$00.75 © 1982 IEEE

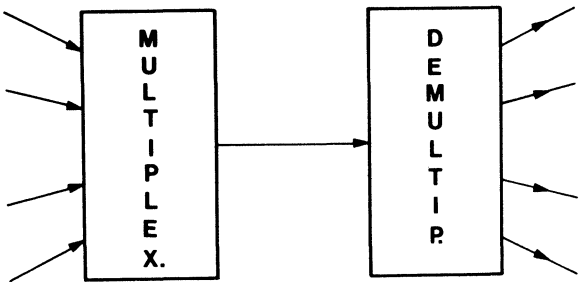


Fig.1. Block diagram of the switch used.

Let N be the number of input and output devices and n be equal to $\log_2 N$. The the whole F network is constituted by n+1 stages composed of N nodes. While each node in a middle stage is a full switch, the nodes located in the stages 0 and n are constituted by only the right or left half of a full switch. Hence, the overall complexity, is $N\log_2 N$ times the complexity of the swicth shown in Fig.1.

The nodes within each stage are numbered from 0 to N-1 and from top to bottom, while the stage are numbered from 0 to n and from left to right. The input devices are connected to the nodes in stage 0 and the output ones are connected to the nodes in stage n. The two sets of input and output devices may or may not be coincident. In the rest of this paper, a node of the network will be referred to as P_j, where P ($0 \leq P < N$) indicates the number within the stage and j ($0 \leq j \leq n$) indicates the stage number.

One vector of bits $(P_{j,n-1},\ldots,P_{j,0})$ can be associated to each node; each vector is calculated so that the following relation holds:

$$P_j = \sum_{k=0}^{n-1} P_{j,k}\, 2^k \qquad P_{j,k} \in \{0,1\} \qquad (1)$$

In other words, $(P_{j,n-1},\ldots,P_{j,0})$ is the representation of the number P in the binary number system. The interconnections between the nodes in the F-network are defined using the following rules.

Definition 2.1.

The topology of an F network with N input and N output devices can be obtained by connecting the four outputs of a generic node P_j ($0 \leq j < n$) to the nodes P_{j+1}, Q_{j+1}, R_{j+1}, S_{j+1} where the number of these nodes are expressed by the following strings of bits:

$$P_{j+1}=(P_{j,n-1},\ldots,P_{j,j+1},P_{j,j},P_{j,j-1},\ldots,P_{j,0})$$

$$Q_{j+1}=(P_{j,n-1},\ldots,P_{j,j+1},\bar{P}_{j,j},P_{j,j-1},\ldots,P_{j,0})$$

$$R_{j+1}=(\bar{P}_{j,n-1},\ldots,\bar{P}_{j,j+1},\bar{P}_{j,j},P_{j,j-1},\ldots,P_{j,0})$$

$$S_{j+1}=(\bar{P}_{j,n-1},\ldots,\bar{P}_{j,j+1},P_{j,j},P_{j,j-1},\ldots,P_{j,0})$$

It is worth noting that the F network is a superset of the binary cube network; in fact, the former can emulate the latter, by using only the connections to P_{j+1} and Q_{j+1}. The F network with N=8 is shown in Fig.2.

Although the network topology seems very complicated the routing algorithm is very simple. In fact the F network belongs to the "digit controlled" class of networks [4]; that is, the routing at each stage is performed only on the basic of a single digit within a routing tag. In our case, since the message entering a node can be routed to one out of four outputs, the routing tag $T=(t_{n-1},\ldots,t_0)$ is composed of n four-valued digits t_j ($0 \leq j < n$). The four possible values of t_j are 0, 1, 2, 3; the choice of the set of the values for t_j is merely conventional. The path-finding process of a message is based on the use of a special function f defined below.

Definition 2.2.

The function f (P , t_j) accepts a string of bits P and a four-valued digit t_j and produces a string of bits (i.e. a number) defined by the following relation.

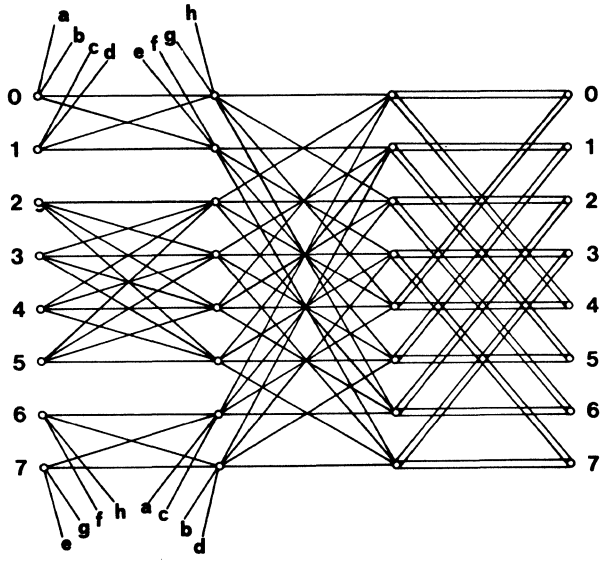


Fig.2. The 8x8 F-network.

$$f(P,t_j)=\begin{cases}(P_{n-1},\ldots,P_{j+1},P_j,P_{j-1},\ldots,P_o) & \text{if } t_j = 0\\ (P_{n-1},\ldots,P_{j+1},\bar{P}_j,P_{j-1},\ldots,P_o) & \text{if } t_j = 1\\ (\bar{P}_{n-1},\ldots,\bar{P}_{j+1},P_j,P_{j-1},\ldots,P_o) & \text{if } t_j = 2\\ (\bar{P}_{n-1},\ldots,\bar{P}_{j+1},\bar{P}_j,P_{j-1},\ldots,P_o) & \text{if } t_j = 3\end{cases}$$

Once the right routing tag has been calculated, a request for the output $D=(d_{n-1},\ldots,d_o)$ generated by the input $S(s_{n-1},\ldots,s_o)$ is routed according to the following recursive procedure:

$$M_o = S$$

$$M_{j+1} = f(M_j, t_j), 0 < j \leq n \qquad (4)$$

$$D = M_n$$

where M_j $(0 \leq j \leq n)$ is the node at the stage j involved in the past between S and D.
The most important feature of the F - network is that it gives 2^n possible destinations, while $4^n = 2^{2n}$ different routing tags are allowed. From the pathfinding procedure defined by (4), it can be established at once that the sequence of nodes M_j composing a path is altered, even if only one digit composing the related routing tag is altered. Hence different paths correspond to different routing tags.
Furthermore, since the number of the possible routing tags is larger than the number of the possible destinations, several paths exist in a F-network, connecting an input to an output node. For example, if a message for the output 6 is generated at the input node 4, in the network shown in Fig.2, it can be routed by using on e of the following tags: (0,1,0) (1,0,2), (0,2,2), (1,3,0).
The following theorem provides us with two important results.
It shows how a message can be routed through the network and how the redundant paths can be used to circumvent faulty nodes. Moreover, the algo — rithm presented needs a routing tag representation using only n+1 binary digits rather than 2n.

Theorem 2.1. Let $C = (C_{n-1},\ldots, C_o) = S \oplus D$, where the symbol $\oplus$ indicates the bit-wise exclusive-or operation performed on the two vectors of bits S and D. Let r be a binary variable initially set to 0 and r(j) be the value of r after the completion of the j-th step of the routing algorithm described by the following recursive procedure:

$$M_{j+1}=f(M_j,(C_j \oplus r^{(j)})), \quad r^{(j+1)}=r^{(j)} \qquad (5)$$

or, alternatively

$$M_{j+1}=f(M_j,2+(C_j \oplus r^{(j)})), \quad r^{(j+1)}=\bar{r}^{(j)} \qquad (6)$$

with $M_o=S$ and $M_n=D$.

Proof

In order to prove the thesis of the theorem, it is sufficient to show that the j-th bit of M_{j+1} is equal to d_j and that the bits form 0 to $j-1$ are equal in M_{j+1} and M_j. In fact, once such a result is proven, the k-th bit of M_n is equal to the k-th bit of M_{k+1}, that is d_k; hence $M_n=D$.
In general, when the routing algorithm reaches the step j a certain number of routing steps making use of the equation (6) have already been esecuted.
The two cases of a even and an odd number of such steps are considered separately:

a) even: $r^{(j)}=0$ and $m_{j,j}=s_j$, since, from (6), an even number of complementations have been performed on these bits and their initial values where 0 and s_j, respectively. From Definition 2.1. we derive that the following relation holds:

$$m_{j+1,j+1}=m_{j,j} \oplus (r^{(j)} \oplus C_j)= s_j \oplus C_j \qquad (7)$$

since C_j was computed as $s_j \oplus d_j$, the (7) becomes

$$m_{j+1,j+1}=s_j \oplus s_j \oplus d_j = d_j \qquad (8)$$

b) odd: $r^{(j)}=1$ and $m_{j,j}=\bar{s}_j$, since, from (6), an odd number of complementations have been performed on these bits, and their initial values where 0 and s_j, respectively. Using the same arguments as in the case a, it is possible to show that the following relation is valid:

$$m_{j+1,j+1}=\bar{s}_j \oplus 1 \oplus d_j \oplus s_j = d_j \qquad (9)$$

In both cases the bits from the 0-th to the (j-1)-th are copied from M_j into M_{j+1}.
At each step of the routing algorithm illustrated in Theorem 2.1., it is possible to calculate the next node of the path, using one out of two possible formulas. In other words, there are always at least two different paths starting from a node within the F-network and leading to the same destination node. Only when the message reaches a node in stage n-1, the alternative paths lead to the same destination node different merely from a formal point of view. In fact, while either equation (5) or equation (6) can be used, the result is always D. This feature derives from the assumption that a different device is connected at each node in stage n. Theorem 2.1. assumes that the routing is performed on the basis of a binary variable r and a vector of bits C, by applying a sequence composed of a mix of two types of steps. By using all the possible patterns of steps, all the paths interconnecting the same input-output

pair are generated. Given an input node, there are 2^n possible tags leading to the same output node; however, only 2^{n-1} distinct paths exist, since the tags differing only in t_{n-1} produce identical paths. Since 2^n possible destinations exist, the 2^{2n-1} paths starting from an input node are equally distributed among all the possible destinations. Furthermore, the F-network is designed so that alternative paths exist at each stage. This feature allows an on-the-fly rerouting of a message, when some nodes in a network are faulty. In fact, if at step j of the routing algorithm, the next node selected by using equation (5) is faulty, the message can be routed to the node selected by using equation (6), and vice versa. The on-the-fly re-routing can be usefully employed to enhance the network bandwidth, since the nodes previously acquired must not be released, and the re-routing is accomplished in a short time interval.

3. Broadcasting

Broadcasting capability is an important issue for a connecting network, since the algorithms executed on multiprocessor systems often require that the result of some computation should be sent to a pool of processor and/or memory banks. Broadcasting in the F-network is discussed in this section.

In general, in an interconnection network, the path, for a multi-destination message is established by duplicating, on different output links of some switch, either the arriving packet (packet switching) or the request-to-connection (circuit switching). The F-network also works in this way; furthermore, since more than two outputs per node are available, broadcasting in the network presented here has the same rerouting properties as the point-to-point transmission.

The routing of a multi-destination message is performed on the basis of the routing tag C, the bit r, defined in section 2, and an additional n bit broadcasting mask $B=(b_{n-1},\ldots, b_0)$, used to identify the switches which should duplicate the arriving message on their outputs. In fact, a node in stage j, finding $b_j = 1$, duplicates the message, while, if $b_j = 0$, the node behaves as in a point-to-point connection. The duplication of a message on two different output links of a node in stage j can be seen as the superimposing of two point-to-point connections, one with $C_j=0$ and the other with $C_j=1$. Hence, by applying formulas (5) and (6), either of the following nodes in the stage j+1 can be selected for the copy of the message corresponding to $C_j=0$

$$M_{j+1}=f(M_j,0) \quad r^{(j+1)}=r^{(j)} \qquad (10)$$
$$M_{j+1}=f(M_j,2) \quad r^{(j+1)}=\overline{r}^{(j)}$$

while, for the copy of the message corresponding to $C_j=1$, either of the following nodes can be used:

$$M_{j+1}=f(M_j,1) \quad , \quad r^{(j+1)}=r^{(j)} \qquad (11)$$
$$M_{j+1}=f(Mj,3) \quad , \quad r^{(j+1)}=\overline{r}^{(j)}$$

Thus, in the F-network, two alternative paths exist for each copy of a message, which is duplicated at a node to allow broadcasting transmission. Note that the two copies of a message may be routed independently, hence in general the value of $r^{(j+1)}$ is not equal for both copies.

It is easy to proof that if a message is duplicated only once, in stage j, it will reach only two outputs, whose numbers differ in only the j-th bit. When many duplication of the same message occur, the final result obtained is equivalent to the superposition of the results of each single message duplication. Hence, the following theorem may be easily proven.

Theorem 3.1

A source node $S=(s_{n-1},\ldots, s_0)$ can broadcast to the 2^i destination nodes $D=(d_{n-1},\ldots,d_0)$. Whose numbers are obtained by taking all the 2^i combinations for the bits $d_{k_{i-1}},\ldots,d_{k_0}$ and fixing a value for the other n-1 bits. The complete routing tag should be computed as follows:

$$r^{(0)} = 0$$
$$C = D \oplus S$$
$$B = (b_{n-1},\ldots,b_0)$$

where

$$b_j = \begin{cases} 1 & j=k_0,k_1,\ldots,k_{i-1} \\ 0 & \text{otherwise} \end{cases}$$

Theorem 3.1 states that the cardinality of the set of the destination nodes should always be a power of two. In effect, it is possible to apply to the F-network a broadcasting scheme presented in [10] for IADM and ADM networks; in addition, it is possible to eliminate the constraint, imposed in the original presentation of this method, on the contiguity of the stages duplicating the message.

The first step is to compute the difference between the number of destinations and the largest power of 2 less than that number. The binary representation of such a difference can be embedded into C, by using the bits $C_{k_{i-1}},\ldots, C_{k_0}$, which are not used by the stages duplicating the message, as shown by the previous discussion.

(C.AND.B) is a vector of bits, where the bits in position $k_{i-1},\ldots,k_0$ express the current value of the count, while the others are meaningless and

set to 0. When a decrement of 2^{k_j} is performed on the (C.AND.B) number, the bits of the result in position $k_{i-1},\ldots,k_0$ express the value of the input count decremented by 2^j, while the other bits are still meaningless. In order to obtain the correct output value of the whole vector C, the bits of the result $(x_{n-1},\ldots,x_0)$, obtained by decrementing (C.AND.B), and the bits of C must be merged according to the following rule:

$$C_j = \begin{cases} x_j & , \quad j=k_0,k_1,\ldots,k_{i-1} \\ C_j & \text{otherwise} \end{cases} \qquad (12)$$

This decrement and merge operation is used by the routing algorithm performed by the control units of the nodes which should duplicate the arriving message. This algorithm is described by the procedure shown in Fig. 3.

```
if b_i=0
    then if (R.AND.B) and b_j=0, ∀j>i
        then do not duplicate the message and
             send the single copy as if C_i + r^(i) =0
        else substract 2^i from (R.AND.B)
             set the counter to 0 if the result
             is negative merge the result of the
             substraction and C
             if count≠0
                then route the copy with modified
                     C as if C_i ⊕ r^(i) =1 as if
                     C_i ⊕ r^(i) =0;
                else do not duplicate the message
                     and send the single copy as
                     if c_i ⊕ r^(i) =0;
```

Fig.3. Procedure executed by a control unit of a switch which should duplicate the arriving message.

Note that the binary broadcast subtrees pruned are always those leading to the highest numbered outputs.
Finally, it should be noted that the F-network allows a message to be duplicated and both copies to the rerouted, at the same node, even when the algorithm described in Fig.3 is applied. In fact, the bits of C are never changed by the rerouting procedure, hence the bits of C.AND.B entering a control unit are always correct.
In conclusion, the F-network is able to perform the most sophisticated broadcasting techniques allowed by other similar networks; in addition, it is able to combine such broadcasting properties with the dynamic rerouting capabilities discussed in section 2.

4. Reliability modelling

The primary goal of the F-network design was to obtain an interconnection network able to correctly handle the communications between its input and output devices, even if some nodes are faulty. The final result expected is the enhancement of the network reliability. The results presented in this section give an estimation of the reliability enhancement achieved. The analysis is based on the following assumptions;

a) the faults occur, independently, only within the network nodes;
b) the nodes in the stages 0 and n are considered fault-free;
c) each kind of fault prevents the correct execution of any node operation, hence a faulty node is totally unavailable;
d) the whole system is considered faulty, when the number and the location of the faulty nodes prevent the communications between at least one input-output pair.

Hypothesis a is obvious, since only network reliability should be studied. Hypothesis b derives from the assumption that each input or output device communicates with the network by only one port. In this case, the failure of the first switch connected to one of these ports prevents every communications with the corresponding device. Hence, the faults within the nodes in stages 0 and n are not recoverable by using a suitable network topology, but their effect should be avoided only by the reliable implementation of such switches.
Hypothesis c leads to a conservative analysis, since, in general, a fault does not destroy all node functionalities. Finally, under hypothesis d, the occurrence of non-critical faults does not prevent any system operation.

Theorem 4.1.

The minimum number of faults leading to a system failure is 2.
The proof derives directly from the routing algorithm, which allows two alternative paths at each stage.

Theorem 4.2.

The maximum number of faults possible without causing a system failure is $\frac{N}{2}((\log_2 N)-1)$, where N is the number of input and output devices.

Proof

From the proofs of Theorem 2.1., it follows that

all the messages requiring the use of the node $(P_{n-1},\ldots,P_j,P_{j-1},\ldots,P_o)$, at the stage j, may be rerouted only to the node $(P_{n-1},\ldots,\bar{P}_j,P_{j-1},\ldots,P_o)$, and viceversa. Hence, the N nodes whithin a stage can be divided into N/2 subets of 2 nodes, referred to as β subsets. Since the network does not fail until both nodes in any β subset are faulty, the maximum number of faults in a non-faulty network is $\frac{N}{2}\log_2 N$.

Theorem 4.1. and Theorem 4.2 provide the lower and upper exact bounds on the number of faults, which cause a system failure. However, it can easily be realized that both the best and the worst cases occur only when some particular pattern of faults is found. Hence, the network reliability characterization provided by the results of the previous theorems is too poor. Since the fault location is random, the number of faults causing the system failure is a random variable. Thus, it is important to evaluate its mean value, k. The general expression for k is the following one:

$$\bar{k}=\sum_{i=2}^{L} i\,P(i) \qquad , \qquad L=\frac{N}{2}((\log_2 N)-1) \qquad (13)$$

where P(i) is defined as follows:

$P(i) = Pr\{$ the i-th fault causes the system failure$\}$

This probability can also be expressed by the following formula:

$$P(i)=Q(i-1)\,R(i) \qquad (14)$$

where:

$Q(i-1)=Pr\{$ i-1 faults do not cause the system failure$\}$

and

$R(i)= Pr\{$ a fault causes the system failure $|$ i-1 faults have already occurred and the system is not faulty$\}$

In a NxN F-network, L β subsets exist, and the whole system is not faulty until both nodes in the same β subset are faulty. Hence, the fault patterns preserving the system functionalities are constituted by nodes belonging to different β subsets. The number of the groups of i-1 different subsets is $\binom{L}{i-1}$; 2^{i-1} fault patterns correspond to each group, because it is possible to choose independently for each subset the faulty node; since all the fault patterns are equally probable, Q(i-1) is given by the following expression

$$Q(i-1)=2^{i-1}\binom{L}{i-1}\Big/\binom{2L}{i-1} \qquad (15)$$

Given a non faulty F-network with i-1 faults, the i-th failure of a node causes a system failure if and only if the new faulty node belongs to the same β subset as one of the previosly failed nodes. Since such nodes belong to i-1 different subsets, there are i-1 nodes out of 2L-(i-1), the failure of which will cause a system failure. Hence, R(i) can be expressed as follows:

$$R(i)=\frac{i-1}{2L-i+1} \qquad (16)$$

Note that Q(1)=1 and R(1)=0, as required by Theorem 4.1., and Q(L+1)=0 and R(L+1)=1, as required by Theorem 4.2. By using the equations (13),(14), (15) and (16), it is possible to compute k.
Fig.4 shows the value of k for different network sizes. It is worth noting that for Delta, IADM and ADM networks k is always 1. In fact, the Delta networks provide only one path between a input-output pair, hence the failure of a single node will cause a system failure.
The IADM and ADM networks in general provide multiple paths between an input device and an output one; however, for some input-output pairs, there is only one path. Since each node within such networks is involved in at least one of these unique paths, a single failure will cause a system failure.
Let us compare the reliability of the F-network with that of a system of h parallel 2x2 delta networks like that shown in Fig.5 where a network is switched off as soon as one of its nodes fails.
For the sake of uniformity, it will be assumed that each delta network is implemented replacing each 2x2 crossbar switch with a fully connected bipartite graph with 4 nodes, each one constituted by a switch like that shown in Fig.1, with 2

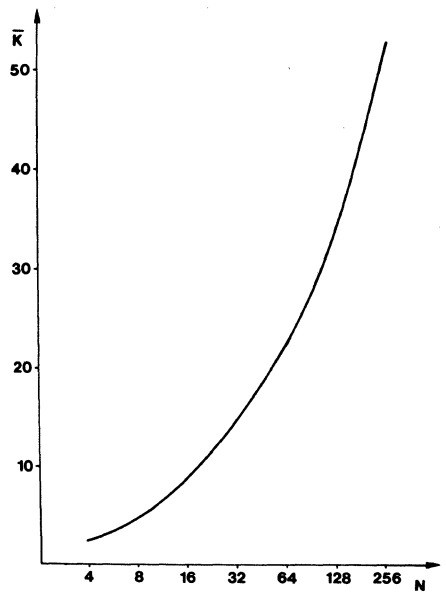


Fig.4. Average number of faults leading to the system failure for different network sizes.

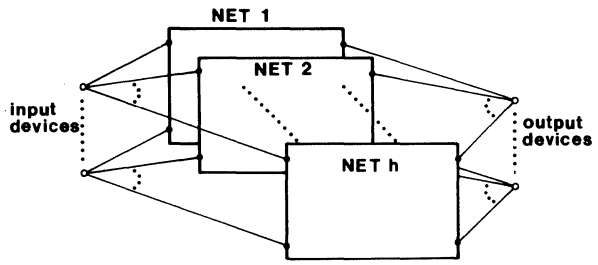


Fig.5. Redundant interconnection system composed by h parallel networks.

inputs and 2 outputs. Furthermore, it will be assumed that the time to failure of each node is a random variable with negative exponential distribution and mean value λ^{-1}. Since each node of the F-network is twice as complex as that of a delta network, it is assumed that the failure rate for a switch of the F-network is 2λ.
At this point, it is possible to compute the mean time before the failure (MTBF) for the network proposed here and the configuration shown in Fig.5. The MTBF of a F-network, $MTBF_F$, can be computed as follows:

$$MTBF_F = 0.5\,\lambda^{-1} \sum_{i=1}^{L} \left(\sum_{j=0}^{i-1} (L-j)^{-1} \right) P(i) \qquad (17)$$

For a single delta network, the failure is caused by a single node failure, because there is only one path between each input-output pair. Since the network has identical nodes, the MTBF is given by the following formula:

$$MTBF_\Delta = (2\lambda L)^{-1} = (\lambda N \log_2 N)^{-1} \qquad (18)$$

It is worth noting that the life time of a delta network also has a negative exponential distribution. Hence, it is easy to compute by using formulas of classic reliability modelling the MTBF for a system with h identical parallel networks where each single network is switched off when one of its nodes fails. The final result is given by the following expression:

$$MTBF_{h\Delta} = (\lambda N \log_2 N)^{-1} \sum_{i=1}^{h} i^{-1} \qquad (19)$$

The plots of the MTBF for the F-network and for a system with h parallel delta networks are shown in Fig.6.
It can be seen that the parallel delta networks achieve the same MTBF as the F only when the size of the network is small, for realistic values of h. However, interconnection networks are intended for very large multiprocessor systems, hence the range of interest is shown on the right side of Fig.6. After simple calculation, it can be seen that for

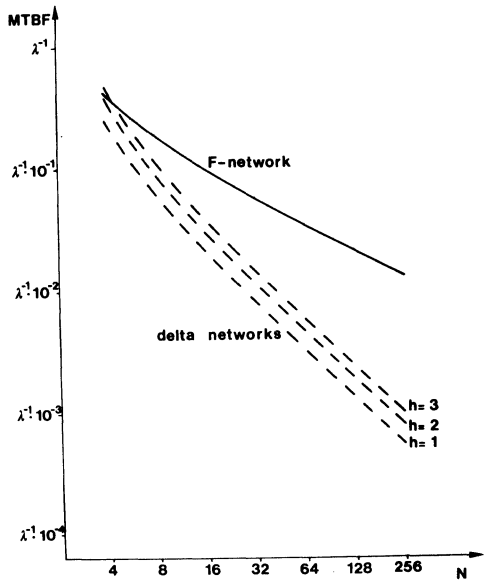


Fig.6. MTBF for the F and parallel delta networks.

N=32, more than 1000 delta networks are needed to achieve the same MTBF as a single F network; this number goes up to above 10^6, when N=128 is considered.
In other words, the previous analysis shows that the MTBF attained by a single F network cannot be reached by a system where the redundancy is obtained by using several delta networks in parallel.

5. Network modularity

Previous work on the LSI and VLSI implementation of interconneting networks [12] ÷ [14] has shown that switches belonging to different stages must be integrated in a single circuit, in order to obtain the minimum chip count for implementing a given network. Unfortunately, the use of basic blocks composed by switches of different stages imposes some constraints on the network topology.
Hence it is not always possible to define such basic building blocks, for each interconnecting network. The modularity issues of the F-networks are discussed in this section. The goal is to show how a F-network of a given size can be built, interconnecting several smaller multistage subnetworks. The most important feature of these subnetworks should be the limited number of interconnections. In fact, smaller the number of input and output signals, the smaller the number of pins required for implementing each subnework in a single chip. Since the pin count rather than the area is the main limiting factor for the integration of large subnetworks, the basic builing block with the minimum number of interconnections gives us the best VLSI implementation.

Definition 5.1

A SUBF network with $M=2^m$ ($m \geq 2$) inputs, referred to as SUBF (M), is a network obtained from an M input F-network, by using switches with 4 inputs and 2 outputs in the last stage.

From Definition 5.1. if follows that a SUBF (M) has 2M output links.

Each output node has 2 outlets, which will be distingnished by referring to them as the "dashed" link and the "solid" link, with an obvious reference to their representation in the figures of this paper. Hence, each SUBF (M) has M "solid" and M "dashed" outlets.

In order to allow the connection of several SUBF(M), it is assumed that the implementation of the output nodes of such networks is in accordance with the block diagram shown in Fig.7. The selection of the "dashed" or the "solid link" is performed by enabling the appropriate three-state buffer; in this way, several output links of a SUBF network can be tied to the same input link of a different SUBF network, without extra logic.

A NxN F- network can be obtained by using (N/M) $\log_M N$ SUBF(M) networks, arranged in $\log_M N$ stages of N/M subnetworks. The first stage should perform all the routing functions of the first m stages of the F-network.From the routing algorithm presented in section 2, it can be deduced that a message entering the F-network from input $p=(p_{n-1},\ldots,p_0)$ can reach either node in stage j, whose number is expressed by $(p_{n-1},\ldots,p_j,x,\ldots,x)$ or by $(p_{n-1},\ldots,p_j,x,\ldots,x)$ where a string of x stands for any binary string of the same length.

Hence, in order to preserve such a behavior for $o \leq j \leq n-1$ it is necessary for each SUBF(M) to group the input expressed by $(p_{n-1},\ldots,p_{m-1},x,\ldots,x)$ and by $(p_{n-1},\ldots,p_{m-1},x,\ldots,x)$; varying the string $p_{n-1},\ldots,p_{m-1}$ all the N/M pairs of groups are generated. Moreover, since a message occupying the node $(p_{n-1},\ldots,p_{m-1},d_{m-2},\ldots,d_0)$ at stage m-1, can reach either node $(p_{n-1},\ldots,p_m,d_{m-1},d_{m-2},\ldots,d_0)$ or node $(\bar{p}_{n-1},\ldots,\bar{p}_m,d_{m-1},d_{m-2},\ldots,d_0)$ at stage m; it is necessary to use the two output links to reach all the possible nodes. In particular, the "dashed" link of the output $(p_{n-1},\ldots,p_m,d_{m-1},d_{m-2},d_0)$ is connected to the "solid" link of the output $(p_{n-1},\ldots,p_m,d_{m-1},d_{m-2},\ldots,d_0)$. In this way, the choice of the value of d_{m-1} is performed at the last stage of the SUB(M), while the adjustment of the most significant n-m bits in performed by choosing either the "solid" or the "dashed" link.

The second stage of SUBF networks is able to perform the same operation as the first one; however, since it should operate on the second block of m bits, there is a m-unshuffle permutation, which causes an m-position right rotation of the bits expressing the node number. Each subnetwork groups the inputs expressed by $(d_{m-1},\ldots,d_0,p_{n-1},\ldots,p_{2m-1},x,\ldots,x)$ and by $(d_{m-1},\ldots,d_0,p_{n-1},\ldots,p_{2m-1},x,\ldots,x)$. Since the most significant m bits cannot be changed, the connection between "solid" and "dashed" links at the output of the second stage must not influence such bits.

Repeating the same process shown above and taking account of the increasing number of most significant bits which cannot be changed, all the $\log_M N$ stages can be laid out. A final m-unshuffle permutation is required to obtain the correct ordering of the outputs. A 16x16 F-network built using eight SUBF (4) is shown in Fig.8.

Each SUBF(M) allowing the transmission of w parallel signals per input, requires 3wM connections with the outside world. Whereas, each MxM subnetwork proposed in [12] for a class of delta network requires 2wM connections.

Let us consider the problem of implementing a fault-tolerant network with N inputs and N outputs and B parallel signals per input. Two alternatives with similar costs are considered: the use of a NxN F-network, the use of a pair of NxN modular

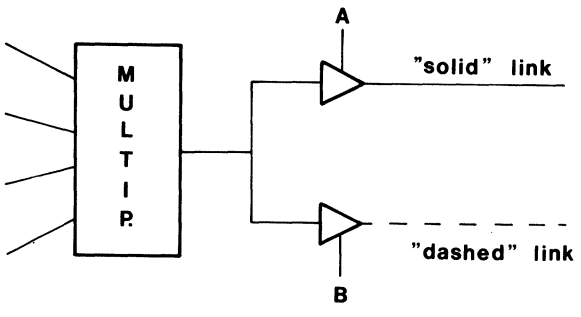


Fig.7. Output switch for a SUBF network.

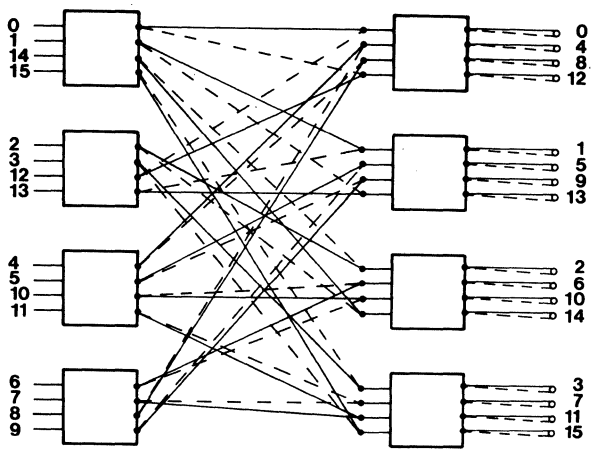


Fig.8. 16x16 F-network built using 8 SUBF(4)networks.

delta networks in parallel.
In the first case, the number of modules to be used, b_F, is given by:

$$b_F = \frac{B}{w} \frac{N\log_2 N}{M_F \log_2 M_F} = \frac{B}{w} \frac{N\log_2 N}{e_f} \propto \frac{1}{e_F} \quad (20)$$

while in the second case, the number of the subnetworks, is expressed by

$$b_\Delta = \frac{B}{w} \frac{2N\log_2 N}{M_\Delta \log_2 M_\Delta} = \frac{B}{w} \frac{N\log_2 N}{e_\Delta} \propto \frac{1}{e_\Delta} \quad (21)$$

The value of M is limited by the number of pins per package allowed, such a limitation does not depend on the type of the network. Hence, if the limited number of pins imposes a maximum of Z_0 connections per subnetwork, the following expressions must hold:

$$M_F \leq \frac{Z_o}{3w} \quad (22)$$

$$M_\Delta \leq \frac{Z_o}{2w} \quad (23)$$

Assuming $M_F = Z_0/3w$ and $M = Z_0/2w$ the following relation can be obtained:

$$b_F/b_\Delta = e_\Delta/e_F = 0.75 + 0.4387 \log_2 M_F \quad (24)$$

From (24) it can be deduced after trivial computation, that for $M_F > 3.3$., the number of chips required by a F network is smaller than that required by a duplicated delta network. Taking account of the relation $Z_0 = 3M_F w = 2M_\Delta w$, it will be seen that b_F and b_Δ decrease by increasing the values of M_Δ and M_F. Hence, the minimal chip count is achieved by making M_F (or M_Δ) as large as possible; in general, such a condition leads to $M_F > 3.3$. Thus, the chip count for implementing a F-network is less than required for a duplicated delta network, although the former has better reliability and performances.

6. Conclusions

Let us now compare the results presented in the previsions sections with the four criteria listed at the beginning of this paper. The analytical models presented in section 4 have shown that the mean lifetime of an F network is so high, that a similar result cannot be achieved by using several networks in parallel, where each network does not have intrinsic fault-tolerance properties. Hence F-network performs very well under criterion 1.
The F-network achieves its high level of reliability by introducing multiple redundant paths between each input-output pair. The routing algorithm is only O(log N), since the F is one of the so-called "digit-controlled" networks [4], hence it allows the control functions to be distributed among several units; furthermore, the selection of alternative paths may be performed on-line, as required by criterion 2, so that simple recovery procedures are allowed and rerouting can be used to enhance the system performances.
F is the only network presented in the literature which holds all the rerouting properties when broadcast communications are considered, even if sophisticated broadcasting techniques are used. Furthermore, since the F-network is a superset of the multistage cube network, it is also able to perform all the other switching functions of the most popular networks, as required by criterion 3. Finally, the number of active devices required by an F network is about equivalent to that required by redundant network composed of two delta networks in parallel. Moreover, if the chip count rather than the number of active devices in considered as a cost function, the F network becomes cheaper than two delta networks in parallel, although the former configuration has better reliability, performance and switching capabilities than the latter.

References

[1] Batcher K., "The flip network in STARAN", 1976 Int. Conf. Parallel Processing, Aug. 1976, pp. 65-71

[2] Goke G., Lipovski G.J., "Banyan networks for partitioning multiprocessor systems", 1st Symp. Comp. Arch., Dec. 1973, pp.21-28.

[3] Lawrie D., "Access and alignment of data in an array processor", IEEE Trans. on Computers, Vol. C-24, 12(1975) pp.1145-1155.

[4] Patel J., "Performance of processor-memory interconnection for multiprocessors", IEEE Trans on Computers, vol.C-30, 10(1981), Oct. 1981, pp.771-780.

[5] Pease M., "The indirect binary n-cube microprocessor array", IEEE Trans.Comp.,Vol.C-26, 5(1977), pp.458-473.

[6] Siegel H.J., "A model of SIMD machines and a comparison of various interconnection networks", IEEE Trans.Comp.,Vol.C-28, 10(1979), pp.907-917.

[7] Feng T., "Data manipulating functions in parallel processors and their implementations", IEEE Trans. on Computers, Vol. C-23,3(1974), pp.309-318.

[8] Siegel H.J. and McMillen R., "The use of augmented data manipulator in PASM", Computer, Vol.14, n.2, Feb. 1981, pp.25-31.

[9] McMillen R. and Siegel H.J. "MIMD machine communication using the augmented data mani

pulator network" 7th Ann. Symp. on Computer Architecture, May 1980, pp.51-58.

[10] McMillen R. and Siegel H.J., "Dynamic rerou ting tag schemes for the augmented data manipulator network", 8th Ann. Symp. on Computer Architecture, May 1981, pp.505-516.

[11] Tripathi A. and Lipovski G.J., "Packet swit ching in banyan networks", 6th Ann. Symp. on Computer Architecture, Apr. 1979, pp. 160-167.

[12] Ciminiera L. and Serra A., "Modular intercon nection networks with asynchronous control", 14th Hawaii Int. Conf. on System Sciences, Jan. 1981, pp. 210-218.

[13] Smith S.D., "LSI design considerations for multistage interconnection networks for pa rallel processing systems", 14th Hawaii Int. Conf. on System Sciences, Jan. 1981, pp.219-227.

[14] Franklin M.A. and Waun D.F., "Pin limita — tions and VLSI interconnection networks", Int. Conf. on Parallel Processing, Aug.1981, pp.253-258.

A FAULT TOLERANT INTERCONNECTION NETWORK USING ERROR CORRECTING CODES

J. Edward Lilienkamp, Duncan H. Lawrie, and Pen-Chung Yew
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

ABSTRACT

A method for constructing a fault tolerant interconnection network is described. It uses error correcting codes to correct errors due to all single and many multiple failures of both switching elements and links, and requires O(Nw) encoders and decoders, where N is the network size and w is the size of the packet in bits, and less than w additional check bits. Also discussed is a method for isolating the failed component after one is detected. This result contrasts with previous results because it allows the network to continue operation while the fault is being located, rather than performing off-line testing.

Introduction

Much research has been done in the area of interconnection networks in multiprocessor systems, but not enough attention has been paid to making such networks fault tolerant. This paper addresses that issue and applies an old technique to the problem.

This fault tolerant network design attempts to provide a cost effective method that allows the network to operate properly in the presence of some set of failures, including all single failures, while maintaining a straightforward routing algorithm.

Previous approaches have attempted to solve the problem by providing alternate paths from each source to each destination, allowing a failed component to be bypassed. Falavarjani and Pradhan [FaPr81] and Adams and Siegel [AdSi82] propose using a standard network with an extra stage. With this extra stage, there is more than one path to each destination. Shen and Hayes [ShHa80] use a simple fault model, and explore the fault tolerant capabilities of conventional configurations. Yew [YewP81] proposes using multiple layer networks, which are by default fault tolerant, since a failure in any particular layer could be avoided by deactivating that layer.

These results have several limitations. First of all, many authors assume an unreasonably optimistic fault model, such as a switching element will only fail by becoming stuck at one of its valid states. Second, many fault models do not include failure of a data link, and all faults are assumed to be permanent and not transitory. Third is the extra complexity added to the routing algorithm. In order to avoid a faulty section of the network, the location of the fault must be known by the sender prior to transmitting the packet. Finally, several passes through the network may be required to pass a permutation when a failure occurs.

The approach here is to apply the techniques of error coding to correct errors caused by a single fault, and to analyze the failure data to locate the faulty component. The use of error correction codes in communication systems is not new [PeWe72]. Pradhan and Stiffler [PrSt80] discuss using error codes to achieve fault tolerance in many computer applications, such as ALU and memory design, but they do not mention interconnection networks specifically. This paper discusses using such codes to achieve a fault tolerant network design.

The Network and the Fault Model

The network to be considered here is the omega network of size $N=2^n$ for packets of w bits routed in parallel, although the technique is applicable to other networks as well. The packets can be buffered in a switching element to allow improved performance [CLYP81] [Chen82], and the network can be operated in one of two modes, SIMD or MIMD.

The network described above can be viewed in three dimensions. The x dimension is the direction of packet flow. The y dimension is the different processors in the system. The z dimension is the parallel bits of the packets sent by the processors. One conventionally views a network in the x-y plane.

A packet consists of many bits, but pin limitations in VLSI technology require that only s bits per package be allowed. Consequently, at least w/s packages are required to implement a single switching element. The fault tolerance scheme presented here will take advantage of the multiple package requirement.

A virtual switching element is defined as a single column of physical switching elements and its associated control. This functions as a single w bit switching element. Similarly, a virtual link is the column of wires that implement a w bit link. Finally, a physical switching plane is defined as a single x-y plane of physical switching elements and their associated links, which with the control implement an s bit layer of the NxN network.

The fault model is general and will encompass almost any single and some multiple failures of real components. It is assumed that any single interconnection link can fail, independently of any other link. An example of a link failure is one that is permanently stuck at logic level 0 or 1, or some level outside the normal logic domain. It is also assumed that any single physical

This research was supported by the Department of Computer Science, University of Illinois at Urbana-Champaign, the National Science Foundation under Grant No. MCS 80-01561, and the United States Department of Energy under Grant No. DOE DE AC02-81ER10822.

0190-3918/82/0000/0123$00.75 © 1982 IEEE

switching element package can fail independently of all of the others, and that such a failure causes some or all of the outputs of that switching element to be invalid. Link and switching element failures can be either permanent or transitory.

Fault Tolerance by Coding Each Packet

The principal of this design is to take advantage of the package redundancy caused by the large word size, and use an error code to correct errors caused by single failures. This requires great care, since faults in the network control could potentially route a packet to the wrong destination.

As mentioned previously, several physical switching elements are required to implement one virtual switching element. As currently described, however, a single fault in the output of the control section could cause all of the physical switching elements in the virtual switch to route the outputs to the wrong port, which is undesirable. If the control section generated three control signals independently and the physical switching elements contained voting logic for these signals, then a single fault in the control section would not cause any of the switching elements to misroute the data. Hence the worst case failure would be a switching element failure. If an error correcting code were used to correct the error caused by a single switching element failure, then the fault would not cause system failure. Furthermore, the same code can be used to correct errors from link failures.

The previous discussion said little about the error correction code required, and did not address the problem of corruption of routing bits. When a switch failure occurs, then a contiguous run of s bits aligned on an s bit boundary could be in error. Any code used would have to be able to correct for such a failure. Furthermore, the destination tag portion of the packet must be corrected in the control section of every stage, so if an error destroys a bit in the tag, the packet will still be routed properly.

Although codes to correct large bursts of errors exist [PeWe72], they are cumbersome to use, particularly in the coding and correcting operations. An alternative is to use a single error correction code on pieces of the data, and distribute the bits of the code words to different physical switching elements. Then a switching element package failure would only destroy single bits of several code words, all of which can be corrected.

One could use a $(2^m-1, 2^m-m-1)$ Hamming code [PeWe72] for such a purpose. The bits of data are divided into parcels of 2^m-m-1 bits each. These are coded in parallel, resulting in $k=w/(2^m-m-1)$ groups of 2^m-1 bit code words, each group containing 2^m-m-1 data bits and m parity bits. Each bit of a code word is sent to a different switching element. One possible assignment is to route bit i of group j to switching element $(i+j-2) \bmod (2^m-1)+1$. The output of the switches at the final stage are unscrambled, and the error correction procedure is applied.

It was initially stated that the scheme was to allow all single faults in the network. Additionally, the errors from several common multiple failures can be corrected, so it is desirable to classify those multiple failures that do not disrupt correct network operation. The only multiple failure that would prevent successful network operation is one that destroys more than a single bit of a code word. A single physical switching plane can fail totally and the network will continue to operate. Thus if a failure occurs and the physical switching plane or some portion of it is implemented on a single card, then the failed section could be removed and replaced while the network continues to operate. Only multiple failures in the same virtual switching element or in different physical switch planes but common to at least one path would cause incorrect operation. Similarly, multiple link failures are allowed if they do not happen to destroy two bits of the same code word.

The network can be made more resilient to error by applying the correction to the code words between every stage. Then the input to each stage is known to be correct, and the network can tolerate single faults in each virtual switching element. This change would require $\log N$ times as many decoders as the original scheme.

The main disadvantage of this technique is the large increase in the number of bits, which is $m/(2^m-m-1)$. The number of encoders and decoders required is $Nw/(2^m-m-1)$. The limitation on the maximum size code is the packet size (including routing bits) and the number of bits in a physical switching element. The total packet size must be at least $s(2^m-m-1)$ bits, or some of the bits in the switching elements will not be utilized.

The most optimal code for arbitrary w can be selected by choosing the smallest value of m such that $2^m-m-1 \geq w/s$. Then the number of parity bits required is ms, and the parity bit overhead is ms/w. Since $m \approx \log(w/s)$, the overhead is $O(s \log(w/s)/w)$, plus $O(Nw)$ encoders and decoders. This is preferable to triple modular redundancy.

Fault Location

This section describes a way to analyze the failure data to locate the failures. The technique is simple and can be performed by an auxiliary processor while the network continues to operate normally. This contrasts with the techniques proposed by Feng and Wu [FeWu81], which require the network to cease normal operation while a series of tests are used to locate the failure.

The error correction codes described in the previous section determine which bits have been corrupted, and thus isolate the fault to a particular physical switching plane. If a single physical switching element fails, then as many as s bits are incorrect, whereas a link failure will only destroy a single bit.

The correction circuits for each data word cooperate together in locating a fault. If an error is corrected, then the source and destination tags are sent to a fault location

processor. The fault location processor accumulates failure information over several cycles of network operation. If all corrections are to the same single bit of the data words, then a link error is suspected. Otherwise, a switching element is suspect. Depending on the type of error, the fault location processor determines at which switching element or link the paths of the failed packets intersect.

Determining the intersection of the paths can be accomplished by comparing the source and destination tags of the corrected packets. Lawrie showed [Lawr75] the path used by a packet is uniquely determined by its source and destination tags. A packet travels through switching elements in stage i

$$\begin{cases} S_{n-1}S_{n-2}\cdots S_1 & i=1 \\ S_{n-1}S_{n-2}\cdots S_{n-i}D_1\cdots D_{i-1} & 2\leq i\leq n-1 \\ D_nD_{n-1}\cdots D_2 & i=n. \end{cases}$$

The intersection of two paths can be determined by comparing the concatenated source and destination tags of the paths, and noting the common bits. The absence of a continuous run of identical bits (not including S_n and D_1) indicates the paths do not intersect at any switching element. The paths from 100 to 110 and 110 to 000 do not intersect at any switching element. On the other hand, the paths from 001 to 011 and 111 to 001 do intersect, because S_1D_{n-3} is 10 in both pairs. Since it matches two bits from the left, the paths intersect at the second stage and switching element 10. The technique is similar for locating link failures, except a link requires n consecutive bits.

In the presence of multiple faults, fault isolation can be difficult. If the network fails due to multiple faults then on-line fault isolation is impossible. If the multiple faults allow continued operation, then the data could be analyzed as before. This will result in a list of suspect locations, but they will not be completely accurate. If two packets go through two separate failed components in early stages of the network and happen to go through a common switching element, then that element could be flagged as faulty. For such cases off-line testing as described in [FeWu81] would be required.

Conclusion

This paper has presented a method for achieving a fault tolerant interconnection design using error correcting codes. The technique utilizes the multiple packages required to implement many parallel bits in the packet. The packets are encoded using several Hamming codes, and the bits of the code are distributed to different physical switching elements. At the destinations the packets are corrected for errors. The error caused by any single failure, and a large class of multiple failures, can be corrected by the codes. The network requires O(Nw) extra hardware in the form of packet encoders and decoders, where N is the network size and w is the packet size. Also, additional hardware is required in the form of extra bits to be transmitted, although that is dependent on the particular Hamming code chosen.

Also presented is a technique for comparing the source and destination tag bits of the packets requiring correction to isolate the fault in the specific switching element or link, while the network continues normal operation. This is different from previous fault location techniques, which require removing the network from the system for special testing.

There are two problems with this technique. First of all, it presumed the existance of fault tolerant encoders and decoders for the error correction code. Secondly, the scheme requires many bits to be routed in parallel. For a bit serial method of transmission some other method for achieving fault tolerance be required.

REFERENCES

[AdSi82] George B. Adams III and H. J. Siegel, "A Multistage Network with an Additional Stage for Fault Tolerance," Fifteenth Hawaii International Conference on System Sciences, Jan. 1982.

[CLYP81] P-Y. Chen, D. H. Lawrie, P-C. Yew, and D. A. Padua, "Interconnection Networks Using Shuffles," Computer, Vol. 14, No. 12, Dec. 1981.

[Chen82] Pin-Yee Chen, Multiprocessor Systems: Interconnection Networks, Memory Hierarchy, Modeling and Simulations, Ph.D. Thesis, University of Illinois at Urbana-Champaign, January, 1982.

[FaPr81] K. M. Falavarjani and D. K. Pradhan, "A Design of Fault-Tolerant Interconnection Networks," submitted for publication, 1981.

[FeWu81] T. Feng and C. Wu, "Fault-diagnosis for a class of multistage Interconnection Networks," IEEE Transactions on Computers, vol. C-30, Oct. 1981.

[Lawr75] Duncan H. Lawrie, "Access and alignment of data in an array processor," IEEE Transactions on Computers, Vol. C-24, Dec. 1975.

[PeWe72] W. Wesley Peterson and E. J. Weldon, Error-Correcting Codes, The MIT Press, Cambridge, MA, 1972.

[PrSt80] D. K Pradhan and J. J. Stiffler, "Error-Correcting Codes and Self-Checking Circuits," Computer, Vol. 13, No. 3, March 1980.

[ShHa80] John P. Shen and John P. Hayes, "Fault Tolerance of a Class of Connecting Networks," Seventh Annual Symposium on Computer Architecture, May 1980.

[YewP81] Pen-Chung Yew, On the Design of Interconnection Networks for Parallel and Multiprocessor Systems, Ph.D. Thesis, University of Illinois at Urbana-Champaign, March, 1981.

DDSP--A DATA FLOW COMPUTER FOR SIGNAL PROCESSING

Eugene B. Hogenauer, Richard F. Newbold and Yul J. Inn

ESL Incorporated
A Subsidiary of TRW Inc.
495 Java Drive
Sunnyvale, CA 94086

Abstract -- ESL Incorporated is presently developing a high speed data flow computer designated the Data Driven Signal Processor (DDSP). Intended primarily for signal processing applications, DDSP is designed to be programmable and modular. The use of data flow architecture provides a natural way of expressing parallelism in algorithms; DDSP maps this parallelism onto a multiprocessing system that can be expanded without software modification. The maximum configuration of 32 processors occupies four chassis and has an execution rate of about 71 MFLOPS. Hardware and high order language designs were coordinated resulting in a compiler that generates extremely efficient code.

1.0 INTRODUCTION

The Data Driven Signal Processor (DDSP) is being developed by ESL Incorporated to meet requirements for a digital signal processor that is cost effective, programmable, modular, and easily interfaced with other digital hardware. Data flow techniques are used because they provide an effective method for programming algorithms in a multiprocessor environment; data flow exposes the fine grain parallelism in algorithms without explicit software directives. This parallelism can then be spread out over several processors to increase overall performance. Data flow is also a natural way of expressing signal processing problems, which engineers typically represent using data flow graphs.

DDSP has been designed for ease of programming with a high order language capable of generating efficient machine code; it is modular, with a variety of possible configurations ranging from one to 32 processors; it is fast, with a full configuration operating at about 71 million floating point operations per second (MFLOPS); and it interfaces with a variety of devices allowing for concurrent data and I/O processing.

Currently, the best way of getting low cost computing power is to use array processors such as the Floating Point Systems AP-120B. Over the past few years, array processors have permitted the solution to a wide class of vector oriented problems, with a cost effectiveness unobtainable with conventional computers. However, this cost effectiveness can be lost when new software is required. Array processors are usually programmed using horizontal microcode that is difficult to write and maintain. The need for a high order language is born out in statistics on programming productivity developed by R. S. Bucy and K. D. Senne, and reprinted in (12). Bucy and Senne measured the number of man-months to program a nonlinear filtering problem on several computers. Their results indicated that programming took 0.5 man-months using Fortran on the VAX-11/780, as compared to 6.0 man-months for microprogramming the Floating Point Systems AP-120B. This factor of 12 is typical of the microprogramming experience at ESL using other array processors. Although array processor hardware is inexpensive, the high software costs can greatly reduce their effectiveness. Floating Point Systems has tried to alleviate high software costs by developing a version of Fortran called AP-Fortran. This approach is effective in improving programmer productivity but, for one benchmark (12) it generated code that was four times slower than a hand-coded version. The Data Driven Programming Language (DDPL) developed for DDSP, mitigates these problems by providing a high order language that is capable of generating efficient machine code. This has been achieved because of the inherent expressive power of data flow techniques, and because hardware and language designs were closely coordinated from the earliest conceptual stages.

Data flow computers operate on an entirely different principle than conventional von Neumann computers: A von Neumann computer executes instructions one at a time (or sequentially), whereas a data flow computer executes nodes when the data for those nodes becomes available. Nodes are like the instructions used in von Neumann computers, except that they can perform more than one operation. These nodes are logically connected by arcs which are used as pathways between nodes. Arcs are used for sending tokens that contain the values used in computations. Arcs are one-directional pathways that have a source and a destination and are connected between the output port of one node to the input port of another. A node is activated when all required tokens have arrived at input ports and is executed when a processor is available. When a node executes, the input tokens to the node are consumed. A data flow program follows a single assignment rule which states that a token can only have one destination. This rule permits the orderly allocation and deallocation of values as they are created by one node and subsequently consumed by another and, it exposes the parallelism in algorithms.

The current work in data flow computers has concentrated on experimental designs that are applicable to the broad range of computational problems. Our approach has been to look at a specific class of problems (i.e. digital signal processing), and to use data flow techniques that result in a cost effective processor as compared to current methodology (i.e. using array processors). An overview of data flow concepts is beyond the scope of this paper; however, good introductions are given by T. Agerwala and Arvind (7), and J. B. Dennis (8). Also of importance is a survey of data flow languages by W. B. Ackerman (9). Our design has been greatly influenced by the work of J. R. Gurd and I. Watson (1,2,3,4), and Arvind and K. Gostelow (5,6), especially in regards to the concept of dynamic tagged data flow. In DDSP a label field is appended to the data tokens in order to distinguish between different instances of the same token. A matching store resembling that proposed by Gurd and Watson is used to match-up pairs of tokens with the same label. The matching process is implemented using a hash algorithm similar to that described by T. Ida and E. Goto (11). The work by J. B. Dennis and K-S. Weng (10) on the use of data streams in data flow computers has also been influential to the DDSP

0190-3918/82/0000/0126$00.75 © 1982 IEEE

design, because of the similarities between data streams and time series data used in digital signal processing.

The next section establishes some of the basic characteristics of DDSP including how it compares with other data flow processors and with array processors. Section 3.0 covers DDSP architecture with descriptions of matching store, the processing element and the interconnection network. An introduction to the Data Driven Programming Language (DDPL) is given in Section 4.0 with explanations on token labeling, the skew algorithm and a special data structure used for communication. Finally, Section 5.0 gives the results of a discrete time simulation for two signal processing benchmarks, Section 6.0 presents an application of DDSP to sonar signal processing, and Section 7.0 reports on the current status of DDSP development.

2.0 DDSP CHARACTERISTICS

DDSP systems can be configured with one processor or expanded to a system having up to 32 processors without software modification. DDSP can meet a wide range of performance requirements starting with a single processor operation at 2.22 MFLOPS, and extending up to a 32 processor system operating at 71 MFLOPS. A single DDSP processor is packaged on two large printed circuit cards. Up to 8 processors can be packaged in a chassis along with a bus controller, diagnostic hardware, and one or more I/O controllers. The maximum system configuration of 32 processors is packaged in four chassis. As a performance benchmark, today's crop of array processors operate at 5 to 12 MFLOPS and can be matched by DDSP systems with 3 to 6 processors. Large DDSP systems exceed the processing capability of the CDC STAR-100 and CRAY-1, two of the world's fastest supercomputers.

Several unique characteristics of DDSP set it apart from other data flow computers. These include:

- a skew algorithm for routing data among processors
- a special data structure called the data driven communication (DDC) structure used for transmitting data between procedures
- generalized labels for multidimensional indexing.

The skew algorithm makes use of a token's label field to direct the token to a specific processor. Using this algorithm, it is possible to have a uniform distribution of processing for a wide class of array and signal processing problems. For many of these problems, the skew algorithm has the added benefit of localizing communication to nearby processors.

In most data flow programming languages, procedures are called with parameter lists similar to those used in conventional programming languages (13,14). For DDSP we have developed a more flexible approach using a type of linked-list that we call a data driven communication (DDC) structure. A procedure is called simply by passing the procedure a pointer to the DDC structure. The structure not only contains data used in the computations, but also control information such as array dimensions and return pointers.

Generalized labels are used in order to give DDSP a powerful method of indexing multidimensional data. All nodes are assumed to operate on arrays of data where dimensions are determined at the point where the corresponding tokens are generated.

DDSP is being designed as an alternative to array processors, by providing low cost computing power with much more system flexibility. The following points demonstrate this flexibility:

- Multi-tasking system. Any number of tasks can proceed in parallel. The hardware changes contexts (switches from one task to another), without any processor overhead.
- Automatic data management. Data storage is allocated when a data value is calculated and is deallocated when the data value is used by subsequent computations. An associative memory allows direct access into a data array even if some elements of the array have not been allocated.
- Continuous data streams. This feature allows digital filtering operations to proceed without the usual overhead associated with indexing across block boundaries.
- Data dependent branching. Unlike array processors, DDSP handles data dependent branching with the ease of conventional computers.
- Data feedback. Infinite impulse response (IIR) filters, adaptive filters, and phase lock loops can be implemented using normal programming techniques. The time required for feedback can be utilized by other tasks that are pending execution. By contrast, array processors have problems with feedback, often resulting in an underutilization of processing resources.
- Macro compiler. Programmers can develop their own libraries of often used functions. In addition, system macros are available for standard functions such as I/O and control.

3.0 DDSP ARCHITECTURE

The need for a programmable, high speed signal processor was the primary reason in choosing a data flow architecture for DDSP. Although digital signal processing involves a high proportion of vector operations, there are enough scalar, conditional and other "non-regular" operations to give data flow architecture a decided advantage over pipelined approaches used in many array processors. DDSP architecture is similar to that developed by Gurd and Watson at the University of Manchester, whose primary motivation has been to build an experimental machine for data flow research. As a result, instruction times for their machine have been kept relatively slow so the operation of the processor can be easily redefined.

DDSP implements a dynamic tagged data flow model where tokens are tagged with a label field determined at run-time. A DDSP system consists of several processors that are closely coupled though an interconnection network. As shown in Figure 1, a processor includes an input queue for temporarily saving tokens, a matching store for associating pairs of tokens, and a processing element for performing high speed integer and floating point computations. The processing element receives a continuous stream of token pairs

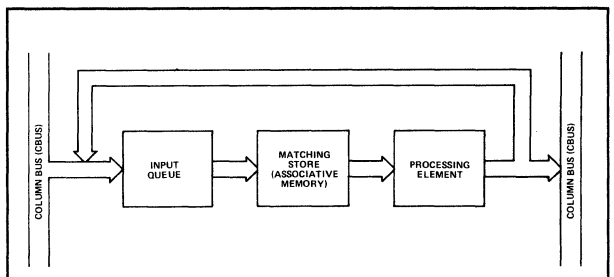


Figure 1. DDSP processor block diagram

for typical applications by designing matching store to equal or exceed the speed of the processing element.

There is an important trade-off in designing a multiprocessing system in regard to the speed of an individual processor vs. the number of parallel processors in a system. Some data flow researchers (8) have indicated that thousands of relatively slow processors can be connected in a data flow system using serial I/O. We feel, however, that there is a point of diminishing return in regards to finding problems with enough parallelism to fully utilize such a large collection of processors. As a result we have designed DDSP with a relatively fast processor that can perform floating point operations in about 450 ns (2.22 MFLOPS). Because of the high speed, it requires a relatively small number of processors to equal the speed of today's supercomputers, and for a given level of throughput, less parallelism in the problem is required to keep all of DDSP's processors fully utilized.

The interconnection network (like the rest of DDSP), has been strongly influenced by the nature of signal processing computations. Much of these computations are vector oriented, requiring highly localized communication. For the most part, communication is local to the originating processor or to its nearest neighbors. Although long distance communication is required, it is usually for transmitting input values and summary results; these require an order of magnitude less communication bandwidth than local communication.

The architecture of the processor is shown in Figure 1. A data token enters the processor from the bus to the left of the processor and is placed in the input queue. The input queue provides load leveling within the processor and temporary storage when processing large volumes of data. The queue is organized as a first-in-first-out buffer with input from the bus and output to matching store.

3.1 Matching Store

The matching store is a high speed associative memory used to match pairs of tokens having identical keys. A key consists of an 11-bit node address identifying the node used for token processing, and a 16-bit label field, providing the token with attributes such as index and iteration numbers. When a match is found, the pair of tokens and the key are sent to the processing element where the node definition is executed. If a match cannot be made then the unmatched token is stored in memory until a matching token comes along. Tokens used in unary operations don't require matching and are simply passed through matching store.

Matching store is implemented by using a parallel hash algorithm devised by T. Ida and E. Goto (11). The algorithm works on the same principle as hash techniques used by compiler designers. Ida and Goto's contribution has been the design of a high speed hardware algorithm that accesses hash tables in parallel and provides a means of deleting table entries when two tokens are matched. In our design, two parallel hash tables are implemented, each capable of holding 16K tokens. The hash algorithm is controlled using a state transition sequencer implemented in firmware. A matching store operation starts by using a hash function to transform the 27-bit key into a 14-bit hash address. The contents of the two parallel hash tables are checked for a match, and if one exists, then the matching token is deleted from the hash table and the resulting token pair is sent to the processing element. If there isn't a match then an attempt is made to store the token at the hash address; however, it is possible for both table entries to be filled resulting in table overflow requiring special processing. In DDSP table overflow conditions are handled entirely within the parallel hash hardware. This contrasts with Gurd and Watson's approach (4), where an independent overflow unit is implemented using a relatively slow speed microprocessor.

DDSP's matching store has a variable cycle time of between 110 and 150 ns; when hash tables are about half full, an average of 250 ns are required for a matching operation including the time for overflow processing. The approach in DDSP is to operate matching store with a faster cycle time than required by the processing element and to allow for a relatively high proportion of overflow processing. In this manner, we are able to implement matching store with two parallel hash tables as compared to the eight tables used by Gurd and Watson.

3.2 Processing Element

A token pair and the corresponding key are input to the processing element when it becomes available. The processing element includes a microprogram sequencer that controls two processing units: an arithmetic processor and a label processor. For the most part, these units operate independently although they can be tied together in order to share resources. The arithmetic processor includes an arithmetic logic unit (ALU) and a high-speed multiplier used for processing both floating point mantissas and integers, an 8-bit ALU for floating point exponent processing, and a memory unit to store constants and intermediate results. The label processor is used for creating new labels and performing various index operations.

Integer operations are performed in one microcycle while floating point operations take 2 cycles for multiplies and an average of 4.5 cycles for adds. Label processing is performed in parallel with these operations and can usually be completed without adding to the overall processing time. Additional overhead is usually required for testing iteration numbers resulting in an average floating point speed to about 450 ns (2.22 MFLOPS). This same type of overhead results in average integer times of about 250 ns (4 MOPS).

3.3 Interconnection Network

Data tokens coming out of the processing element's output queue are output to the interconnection network shown in Figure 2. The network is essentially a linear arrangement of processors with wrap-around from the last pair of processors to the first pair. This arrangement is augmented by a three level tree used for long distance communication. The processors are closely coupled with a minimum amount of network overhead required

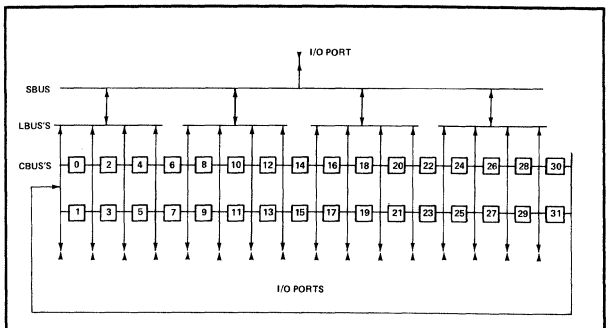


Figure 2. DDSP interconnection network

to pass tokens between processors. At the bottom level of the network are column buses (C-buses) used for local communication along the base of the tree. Tokens can be output onto one of the two C-buses on either side of the originating processor depending on the token's destination. The network is organized like a packet switching network, where packets simply consist of a single token. Each token has its own network destination used in routing itself to any one of the processors in the system or to an I/O port. For signal processing problems, the vast majority of the communication is with local processors; thus, column buses are used the majority of the time. To support longer moves the basic linear arrangement of processors is augmented by a three level tree structure with communication between levels performed through bidirectional queues.

4.0 DATA DRIVEN PROGRAMMING LANGUAGE (DDPL)

One of the primary reasons for using the Data Driven Signal Processor is the ease of programming, as compared to microprogrammed array processors such as the Floating Point Systems AP-120B. DDSP is programmed using the Data Driven Programming Language (DDPL), a high order language with syntax modeled after ADA and language constructs designed for data flow computing.

The programmer designs algorithms for DDSP using a simple conceptual model of parallel processing, without regard to the actual number of processors in the system. The hardware configuration is specified as compile-time parameters, and when the configuration changes, the program is simply recompiled with new parameters. The program itself remains unaltered, allowing for the development of configuration independent software.

A DDPL program has a block structure consisting of a program block containing one or more procedure blocks; a procedure defines a logical group of actions that have a common purpose. Procedures communicate with one another by sending and receiving data driven communication (DDC) structures containing both data and control information; these structures are in the form of linked lists where list members may consist of values or sub-structures. Values include data used in computations, as well as control information used to invoke procedures and route output data. A procedure contains node definitions which are the basic units of data flow computation; a node definition is similar to a task on conventional computers, because it can execute independently of the other software in the system. Node definitions contain all the executable code in a DDPL program including assignment, output, and nested if-then-else statements. In the Data Driven Programming Language, a node definition can have from one to four input ports, and may produce an unlimited number of output tokens. For nodes with three or four inputs, the compiler generates equivalent node definitions with the one and two inputs supported by hardware.

As a data flow program executes, the same node definition may be activated thousands of times. In order to keep track of these various activations, a generalized label is appended to the tokens; for a node to be activated, all the input ports must have tokens with identical labels. The programmer defines the label field on a procedure by procedure basis by subdividing the label into label index fields with programmer defined meanings such as "sample number", "filter number", or "user identification". When writing assignment statements the programmer continually makes references to these index fields in a manner analogous to array subscripting.

A unique skew algorithm is used to map index numbers into specific processor destinations in a manner that provides effective load balancing among processors. This skew algorithm allows the programmer to write software without knowledge of the specific DDSP configuration. For example, an algorithm can be debugged on a DDSP system with 4 processors, and subsequently used on a 32 processor system without software modification, with the skew algorithm automatically spreading the computations over the larger set of processors.

A powerful feature of the Data Driven Programming Language is the ability to define macros and then to expand these macros at compile time using macro substitution and conditional compilation. The motivation for using macro substitution in DDPL is to give the programmer a great deal of flexibility and still allow the efficient generation of machine code. The programmer can build macro libraries for often used operations, using the same syntax as DDPL programs. In addition, there are system macro libraries that include various utility functions for DDC structure creation and manipulation, I/O device control, and various arithmetic functions.

4.1 Destinations and Labels

Tokens are primarily generated within node definitions when an output statement or destination statement is executed. In addition, when the program starts execution, initial token values are generated based on information in the token declarations. No matter which method is used, a token must be given a destination and label. At the machine level, this requires the specification of a network destination, a local destination, and a label field. The network destination specifies the processor or I/O device where the token is being directed. The local destination indicates the destination node within that processor together with the specific port into that node. In the high order language, the programmer specifies a local destination simply by referencing the node input by its symbolic name. The label is specified in a manner analogous to array subscripting, and the network destination is derived without programmer intervention by a combination of compile and run time operations on the label field using the skew algorithm.

The manner in which labels are interpreted is based on a generalized label declaration provided at the start of each procedure. This declaration specifies how the label field is subdivided into label index fields, provides a symbolic name for each index, and indicates the index range. In addition, the declaration specifies how the label indices are used to generate network destinations. Once a decision is made on label field use, a programmer's effort can be concentrated on the programming task itself without any further regard as to how tokens are routed among processors. Generalized labels are more flexible than the fixed format labels proposed by Gurd and Watson (13) because they can be formatted by the programmer to meet specific application requirements.

The label declaration can be described using the following example declaration:

```
label (TIME,COUNT,USER:5,8,3) using (1,-2,0);
```

Here the label field which has a total of 16 bits is divided into three index fields with symbolic names TIME, COUNT and USER. The index fields are assigned a specific number of bits: the index TIME is assigned the high order 5 bits followed by COUNT with 8 bits and USER with 3 bits. The indices are considered unsigned magnitudes so indexing starts at zero and goes in the positive direction. Thus the 8-bit index COUNT has a range from 0 to 255 (i.e. 2**8-1).

The DDPL programmer can think of the process-

ors as being arranged linearly with the last processor connected back to the first processor. In this scheme, higher numbered processors are to the right and lower numbered processors are to the left. As a result, when a token generated in the last processor is routed one processor to the right, it in fact, ends up at the first processor. It is because of this wrap-around feature that the programmer does not have to know specifically how many processors are in the object DDSP system.

A token is routed in the network based on the skew algorithm applied to the token label. Since the same algorithm is used in all processors, tokens with the same label will be routed to the same processor no matter where they were generated. In the above example, the using phrase specifies routing constants corresponding to each index. These constants are used by the skew algorithm in the following manner: Assuming that a node is executing in one of the processors, if the TIME index is incremented, then the resulting token is routed +1 (e.g. one processor to the right). If COUNT is incremented the routing will be -2 (e.g. two processors to the left). If USER is changed, it will have no effect on routing because the corresponding routing constant is zero. As another example, if both TIME and COUNT are incremented, then the routing is the sum of the routings for the individual indices.

To be more specific, the network destination (i.e processor number) for a new token is computed as a function of the token's label indices and the procedure's routing constants using the following skew algorithm:

```
NDEST := (R1*I1 + R2*I2 + ... + Rn*In) mod NPROC
```

where R1, R2, ..., Rn are the routing constants, I1, I2, ..., In are the label index values, and NPROC is the number of processors in the DDSP configuration. The resulting value, NDEST, is the logical processor number in the network. The compiler is optimized to compute the function at compile time if at all possible. As an example of how this algorithm is used, suppose the label declaration is

```
label (I,J,K,L:2,2,2,10) using (1,-1,1,3)
```

and the new label has the index values

```
I - 0
J - 1
K - 2
L - 12
```

Assume that this program is being compiled for a four processor DDSP system, then the current logical processor number is 1*0+(-1)*1+1*2+3*12 mod 4 = 1.

When a node definition is activated, it has an input label field associated with it referred to as the current label that contains the current indices; the processor where the node is activated is referred to as the current processor. Within the confines of this activation, current index values are referred to by their symbolic names as specified in the label declaration. These values can be used in generating labels for output tokens, or used as operands in arithmetic and conditional operations.

The programmer has several ways of specifying a destination. The most common way is to use the symbolic name for a node input followed by an index list. The index list is an ordered list that corresponds one-to-one with the index identifiers in the label declaration. With reference to the above example, a destination might have the form:

```
XDATA (TIME + 3, COUNT - 1, USER)
```

In this example, the TIME index is incremented by +3 and the COUNT by -1 relative to the current indices. XDATA is the symbolic name for the destination and, based on the routing constants above, the destination is 3*1 + (-1)*(-2) = +5 (e.g. five processors to the right).

4.2 Data Driven Communication (DDC) Structures

The primary means for communicating between procedures is by sending and receiving a special data structures called data driven communication (DDC) structures. DDC structures are used in DDPL (rather than the parameters lists common to most high order languages) because of their versatility. For example:

- DDC structures can be created without specific knowledge about the indexing scheme used by the receiving procedure.
- The sending procedure doesn't have to have a complete set of data before it starts to send a structure.
- DDC structures can be created with parallel sub-structures. These sub-structures can be sent independently of each other, thus creating parallel transmission paths between procedures.
- DDC structures can be combined and separated simply by manipulating the pointers to these structures.
- The actual movement of data values at the base of a structure, only occurs on the basis of data availability on the part of the sending procedure, and data demand on the part of the receiving procedure. This two-way data control allows for an orderly flow of data between procedures.
- DDC structures may include not only the data to be processed but control information such as the array dimensions, the type of data in the structure (integer or floating point), or where the results should be sent. The structure can also specify which procedure should be used in processing the data.
- DDC structures can be used to time-share a procedure among several calling routines. Independent DDC structures are used by each of the callers so that they can easily be identified within the receiving procedure.

The programmer creates and manipulates DDC structures by using a set of system macros. The macros generate node definitions that do the actual data manipulation. In Figure 3, an example is given on how DDC structures can be used for matrix multiplication. A matrix multiply procedure is sent a structure consisting of two sub-structures representing the matrices to be multiplied. The dimensions of the matrices are included as part of the sub-structures, and the columns are separated into their own sub-structures so that they can be transmitted in parallel. The matrix multiply procedure receives the DDC structure, performs the computations, and creates another structure as the final result.

The actual manner in which a DDC structure is transmitted is illustrated in Figure 4. The sending procedure outputs a call to the receiving procedure in the form of a pointer indicating where the structure is being created. The call enters a first-in-first-out queue where it stays until the receiving procedure has an activation name available. The number of activation names for a procedure is specified by the programmer and determines the number of calls that can execute simultaneously. One of the index fields defined for the procedure is used to hold the activation name. When an activation name is assigned to the

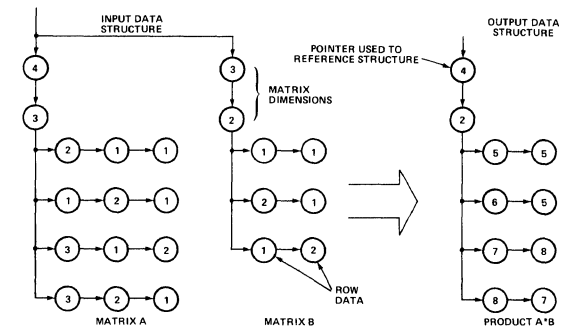


Figure 3. Example data driven communication (DDC) structure

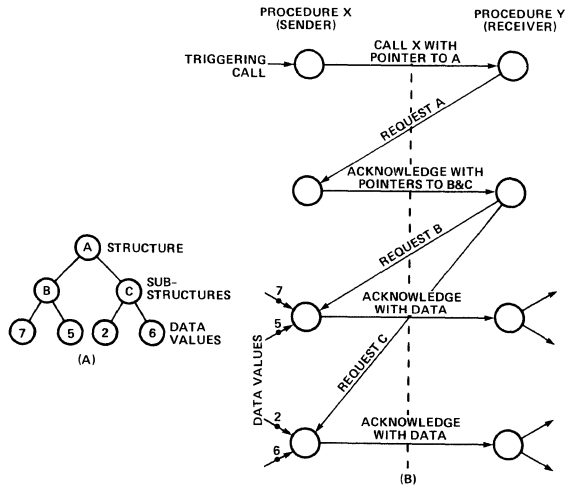


Figure 4. Transmission of DDC structures between procedures: (A) DDC structure to be sent between procedures X and Y. (B) Sequence of events in sending DDC structures between procedures X and Y.

calling routine, the receiving procedure makes a request to the sender which includes a pointer indicating where the structure is to be used. The sender, in turn, sends an acknowledge in the form of a pointer to the next level in the structure. This request/acknowledge process continues until the bottom levels of the structure are reached. At this point, the receiving procedure makes a request for data values from the sender, and the sender waits for data to become available. The data is subsequently sent to the receiving procedure and consumed as part of the computations.

4.3 Node Definitions

Node definitions contain all the executable statements in a DDPL program. These nodes are similar to tasks used by conventional operating systems: They represent a segment of code that

```
  label (TIME,COUNT,USER:5,8,3) using (1,-2,0);
  A, B, SYNC: token integer;
  TEMP: local integer;
    .
    .
    .
node A, B is
  TEMP := A - B;
  if TEMP >= 0 then
    SYNC(0,0,0) := TEMP;
  else
    SYNC(TIME+1,COUNT,5) := COUNT+1;
  end if;
end node;
```

Figure 5. Example node defintion

can be executed independently of all other software in the system. In DDPL a node has a relatively small number of input ports (from one to four inputs). Because of this, node definitions tend to be short and thus result in DDPL procedures consisting of many independent nodes. Each of these node definitions represent an opportunity for parallel activation, and DDSP achieves its high speed as a result of spreading these activations over several processing elements.

One of the features of DDSP, is that there is no processor overhead associated with the transition from one node activation to the next. As a result, the number of node activations has little effect on the overall execution time. In fact, good DDSP programming practice requires that node execution time be kept to a minimum in order to allow as many independent activations as possible. To achieve this, nodes are kept short and program loops are implemented by multiple node activations. This involves the output of a control token with an incremented index field and feeding it back to the node at the start of the loop. This process causes independent and sometimes parallel activation of the loop iterations. It also allows parallel execution of other nodes that are not part of the loop.

Statements that can be used in a node definition include output, assignment, destination and if-then-else statements. Output statements perform basic integer, floating point and logical operations, and output tokens to specified destinations. Assignment statements perform the same types of computations, except the results are normally used within the current node activation. Destination statements are used in conjunction with pointers in order to return results to destinations determined at run-time. Availability of the if-then-else statement represent one of the key advantages of DDSP as compared to array processors. The fact that DDSP can perform data dependent branching means that different software can be used in processing the elements of an array, depending on the element values themselves. This can be achieved with no loss in processor efficiency.

In Figure 5, an example node definition is presented, identifiers are declared at the program and procedure levels. The node executes when tokens with identical labels are available at input ports A and B; TEMP is computed and stored as a local variable and the if-then-else statement determines which of the two output statements to execute. The second output statement uses the current index value, COUNT, as an operand.

5.0 DDSP SIMULATION RESULTS

An important part of DDSP development has been the implementation of a discrete time simulator. The simulator models asynchronous operations

Table I. DDSP Simulation Results for FIR Filter

Number of Processors	Type of Arithmetic	Efficiency (%)	Execution Rate
1	integer	99.99	3.80 MOPS
2	"	99.42	7.58
4	"	97.98	14.98
8	"	95.10	29.02
16	"	86.78	53.12
1	floating point	99.99	2.03 MFLOPS
2	"	99.80	4.15
4	"	99.24	8.24
8	"	97.68	16.16
16	"	92.62	30.72

Notes:
(1) 32 tap finite impulse response (FIR) filter.
(2) 64 samples processed in parallel.
(3) MOPS: Million operations per second.
(4) MFLOPS: Million floating point operations per second.

Table II. DDSP Simulation Results for FFT Algorithm

Complex FFT Size	Number of Processors	Efficiency (%)	Time per FFT (ms)	Execution Rate (MFLOPS)
256	1	95.28	3.76	2.72
"	2	96.00	1.867	5.48
"	4	97.31	0.921	11.12
"	8	97.46	0.460	22.26
1024	1	94.05	19.05	2.69
"	2	95.33	9.40	5.45
"	4	96.89	4.62	11.08
1024 for AP-120B	-	-	4.8	10.7

Notes:
(1) Floating point arithmetic used.
(2) Number of samples processed in parallel increase with the number of processors.

involving interprocessor and matching store-to-processor communication down to the register level. Execution of node definitions are modeled simply by establishing the elapsed time between the start of execution and when tokens are output to the interconnection network. The simulator has been an indispensable aid in the design of DDSP by allowing the design team to perform trade-offs of performance vs. hardware complexity. It has also provided a means for evaluating DDSP performance for some typical signal processing applications. Results for two such applications, a finite impulse response (FIR) filter and a fast Fourier transform (FFT) algorithm are presented in this section.

Table I indicates the execution rate and efficiency of a 32-tap FIR filter. The execution rate indicates the average number of arithmetic operations being performed, excluding index and bookkeeping operations; efficiency indicates the proportion of time that the processing element is performing useful work. The results indicate a slight decrease in efficiency as the number of processors increase, primarily because the number of parallel operations is held at a constant level.

Table II shows similar results for 256 and 1024 point complex FFTs. In these experiments, the efficiency actually increased for larger processors configurations, mainly because the number of parallel operations was allowed to increase along with the number of processors. Results for the 1024 point complex FFT, indicates that a four processor DDSP system (about half a chassis) is comparable to a Floating Point Systems AP-120B.

6.0 DDSP APPLICATIONS

DDSP applications cover the fields of signal, sonar and image processing. DDSP implements basic functions such as digital filters, phase lock loops and FFTs. In addition, its programmability permits more specialized functions such as adaptive filters, synchronous video integrators, and signal search/recognition processors.

DDSP can be used as an attached processor to a host computer or for dedicated applications in a totally self-contained configuration. In Figure 6, one such application is shown for an adaptive array beam former (15) used in sonar signal processing. This example shows the versatility of DDSP to handle a large number of time-shared functions in real-time. In Figure 6, data is collected from 16 independent transducers. The data streams are formed into a beam by adapting to a reference signal generated by the pilot tone generator. The adaption is performed by a least-mean-squares (LMS) algorithm that compares the incoming signal streams with delayed versions of the desired signal. The bandwidth of the desired signal fixes the bandwidth of the main beam and the front end delay determines the beam look direction. Two sets of FIR filters are used in the processes: The first, implements the adaptive LMS algorithm, resulting in continuously updated filter coefficients. The second, applies the updated coefficients to the original data to form the desired beam. The resulting data stream is transformed into the frequency domain by an FFT and then averaged to enhance its spectral components. Display formatting and control are also performed in DDSP, resulting in a completely self-contained system.

The hardware for this application consists of a DDSP system with eight processors, packaged in a single chassis. The processor utilization is about 50 percent, allowing for other time-shared applications.

7.0 DDSP STATUS

Our goal is to have a working DDSP prototype up and running in late 1983. The prototype will consist of four processors with complete software support. It will be interfaced to a VAX-11/780 and a high speed digital recorder. The VAX will be used for compiling and loading software. Data input/output will be performed with either VAX or the high speed recorder.

The hardware design consists of four unique designs. These include the matching store, processing element, bus controller, and I/O controller. The first two designs are repeated for each processor in a DDSP system. The bus controller is repeated for each set of four processors. At this time, the functional design is complete for all but the I/O controller. The detailed circuit design is complete for the match-

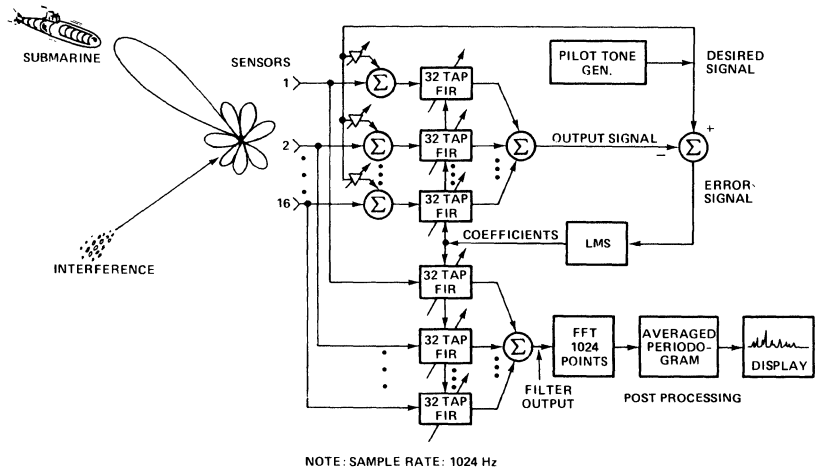


Figure 6. A DDSP application: adaptive array beam former

ing store and is nearing completion for the bus controller.

The software support consists of four major modules including the compiler, assembler, code compactor and diagnostic software. The detail software design is complete except for the diagnostic software. Code and test is complete for the compiler.

8.0 CONCLUSIONS

DDSP's primary attraction is that it solves a total system problem that involves the programming of multiple processors to achieve very high speeds, thereby providing a flexibility unobtainable with array processors. It has several unique characteristics that set it apart from other data flow computers including an efficient algorithm for routing data among processors, a special data structure used for transmitting data between procedures, and a generalized labeling scheme for multidimensional indexing. Simulation results show that processor efficiency is extremely high, even for large DDSP systems with figures well above 90 percent in most cases.

ACKNOWLEDGMENTS

The authors would like to acknowledge Craig A. Peterson for his design of matching store, Danley M. Carlson for implementing the microassembler, and Gloria S. Hogenauer for editing this paper.

REFERENCES

1. I. Watson and J. R. Gurd, "A Prototype Data Flow Computer with Token Labelling," AFIPS Conference Proceeding, NCC, June 1979, pp. 623-628.
2. J. R. Gurd and I. Watson, "Data Driven System for High Speed Parallel Computing -- Part 1: Structuring Software for Parallel Execution," Computer Design, June 1980, pp. 91-100.
3. J. R. Gurd and I. Watson, "Data Driven System for High Speed Parallel Computing -- Part 2: Hardware Design," Computer Design, July 1980, pp. 97-106.
4. I. Watson and J. R. Gurd, "A Practical Data Flow Computer," Computer, Vol. 15, No. 2, Feb. 1982, pp. 51-57.
5. Arvind and K. P. Gostelow, "A Computer Capable of Exchanging Processors for Time," Proc. IFIP Congress 77, Aug. 1977, pp. 849-853.
6. Arvind and K. P. Gostelow, "The U-interpreter," Computer, Vol. 15, No. 2, Feb. 1982, pp. 42-49.
7. T. Agerwala and Arvind, "Data Flow Systems," Computer, Vol. 15, No. 2, Feb. 1982, pp. 10-13.
8. J. B. Dennis, "Data Flow Supercomputers," Computer, Vol. 13, No. 11, Nov. 1980, pp. 48-56.
9. W. B. Ackerman, "Data Flow Languages," Computer, Vol. 15, No. 2, Feb. 1982, pp. 15-25.
10. J. B. Dennis and K-S. Weng, "An Abstract Implementation for Concurrent Computation with Streams," Proc. 1979 Int'l Conf. Parallel Processing, Aug. 1979, pp. 35-45.
11. T. Ida and E. Goto, "Performance of a Parallel Hash Hardware with Key Deletion," Proc. IFIP Congress 77, Aug. 1977, pp. 643-647.
12. W. J. Karplus and D. Cohen, "Architectural and Software Issues in the Design and Application of Peripheral Array Processors," Computer, Vol. 14, No. 9, Sept. 1981, pp. 11-17.
13. J. R. Gurd and I. Watson, "A Multilayered Data Flow Computer Architecture," Dept. Of Computer Science, University of Manchester, March 1980.
14. Arvind, K. P. Gostelow and W. Plouffe, "An Asynchronous Programming Language and Computing Machine," Tech. Report 114A, Dept. of Information and Computer Science, University of California, Irvine, Dec. 1978.
15. B. Widrow, P. E. Mantey, L. J. Griffiths, B. B. Goode, "Adaptive Antenna Systems," Proceedings IEEE, Vol. 55, No. 12, Dec. 1967, pp. 2143-2159.

SUMMARY OF A HYBRID DATA FLOW SYSTEM

Gary N. Fostel

Intermetrics Inc.

Abstract

Recent years have seen an explosive growth in research on high capacity systems incorporating large numbers of processors, producing solutions to varied aspects of the total problem. This paper explores the thesis that a selection from the available techniques together with a synergistic combination of device, architecture and programming technology can yield a very powerful, reliable and usable data flow system for a good price.

Introduction

Is there a pot of micro processors at the end of the rainbow? How big is the pot and what shape is it? This paper summarizes one answer to the second; a more detailed discussion can be found in [10]. The main ideas behind this development are:

- o The programming methodology must be substantially improved; Data Flow languages provide a point of departure.
- o The system's architecture must be "two dimensional" to be consistent with current mass production technology (chips and boards).
- o Hardware must be matched to the software for efficient execution and human comprehension.

Multi-level Programming

One methodology currently under development by the Navy for signal processing algorithms (EMSP [7]) embraces three such levels: low level coding for computational fragments, a data flow (DF) language for organizing those fragments, and a HOL for control programs which monitor and control the DF graph (DFG). This split allows natural expression of the three different, but related facets of the problem.

While multi-level expression might seem to invite incoherent and unanalyzable systems, program verification results suggests just the opposite. Gutag argues that a single formalism will not adequately span the verification requirements of any real world problem [9] and presents a three layer methodology of local, organizational, and system specifications. The is a compelling match between the pragmatics of programming in EMSP and the abstract issues of certification.

The Web programing methodology similarly embraces three levels: machine code, DFG's, and ACTORs. The nature of the machine code depends on the processing elements employed and might be systolic machine descriptions or hand tuned ASM code for side-effect free primitve computations. DF languages are compatible with notations used by engineers and are nearly ideal in that context for organizing the primitives defined by the machine code. A variety of data flow languages exist and are well described elsewhere [1]. ACTOR systems have object oriented, message passing semantics: a generalization of the DF model with greater expressive power owing to greater flexibility. The combination of expressive power and data packages makes such a top level control a ideal extension of DFG's.

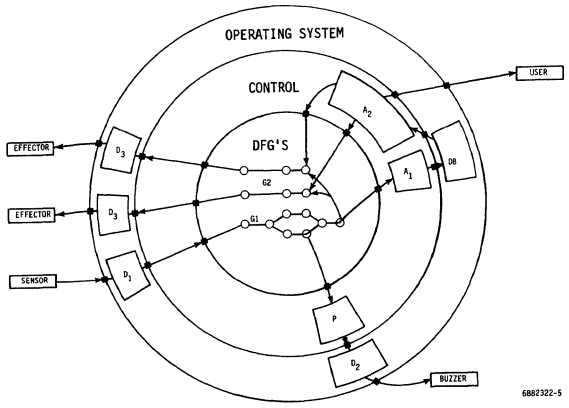


Figure 1.

Figure 1 shows an application with intensive input reduction (G1) and output conditioning (G2). A probe is used for a monitor operation (P). Persistent data is maintained by ACTORs (A1 and DB) and analyzed out of band by ACTOR A2. A2 detects a need to create two instances of G2 and the output of G1 is replicated and fed (as input) to the G2's. The OS provides device interfaces D1, D2, and D3, as well as graph loading, execution, replication of G1 output, and linkage with G2. The ACTORs serve as "intelligent glue" to hold DFG's together as a larger system. The computation in Figure 1 might be a military system which detects a threat, launches two missiles, and then guides them to their targets.

0190-3918/82/0000/0134$00.75 © 1982 IEEE

Web Architechture

VLSI chips will provide cheap performance only if they are part of a larger system which is itself easy to build. For example, many to many connection networks [2] solve some problems, but may be costly to manufacture. Dennis [1] has noted that the "complexity, as measured by total wire length, grows as O(n**2)" for physical layouts of components in many to many networks. Indeed, the connection topology is the heart of the Web design; the name "web" reflects the similarity of spider webs and sketches of the inter-connection topology. There are three specialized processors (chips) to match the three levels of the methodology.

- P - Processor for low level computational fragments. (P-ipeline)
- M - Management of the data flow graphs and interface to the ACTORs. (M-emory)
- G - ACTOR control systems and Web communications and OS. (G-ateway)

There are two interconnect patterns in the Web, one between P's and M's and another between the M's and G's. The M's thus serve as the interface between the processor and control much as a shared memory in a more conventional model. While the M will indeed contain large areas of on-chip memory, significant improvement over a passive device is possible with addition of on-chip intelligence and application tuning.

The M-P grid is principally devoted to the data paths needed to support high bandwidth DFG operation. The application DFG's will be mapped onto the network illustrated in Fig 2. The most critical properties of this model are homogeneity to simplify mapping, high connectivity to resist single failures and allow high bandwidth and lack of any connections which cross to simplify fabrication. The lack of universal interconnectivity implies trade-off in the quality of the mapping of the DFG onto the M-P grid

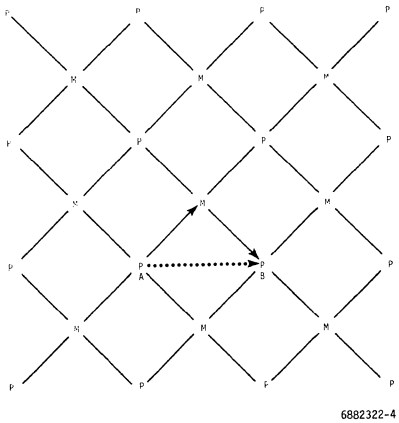


Figure 2.

The M-G grid is responsible for communications among ACTORs and with the external environment. The requirements here are distinct from those of the M-P grid: delay is more critical, volume of data is lower. Larger grid distances are likely in order to maintain global control of the Web. The low level view of the M-G grid is a number of local networks with a small number of M's, bounded by G's which act as gateways to other such local networks. One such might be "G-M-M-M-M-G"; the M's are said to be an M-string. The high level topology of these strings is illustrated in Figure 3, with the intersection points representing G's and the connecting lines the M strings. The result is a tree network with additional circular links added to reinforce the fault tolerance and delay characteristics of such trees, yet containing no crossed lines. These networks are discussed in [5].

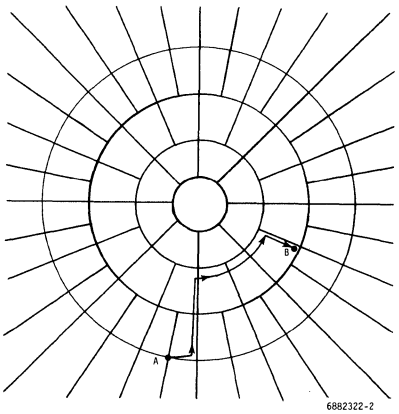


Figure 3.

Implementation Problems

First, the the DFG to M-P mapping must be optimized; further the control system must be able to keep up with the processing. system in the M-G grid will be hard pressed to control the application. Second, the M, P and G chips must be designed and built; the goals are aggressive: million bit chip technology and miniaturized, HOL architecture with store and forward network interfaces in each chip. An optimization strategy is proposed (Figure 4) which integrates pre-compile, compile, load and run-time techniques.

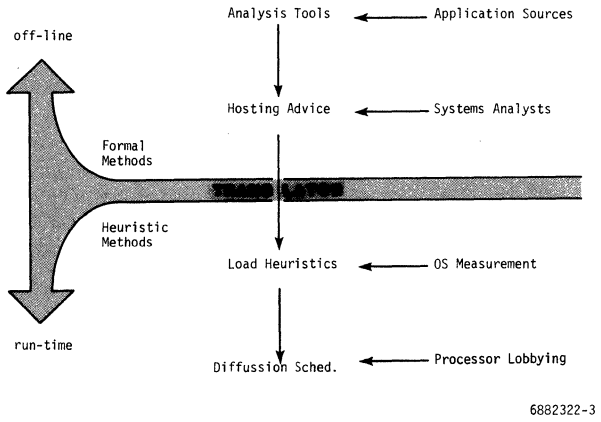


Figure 4.

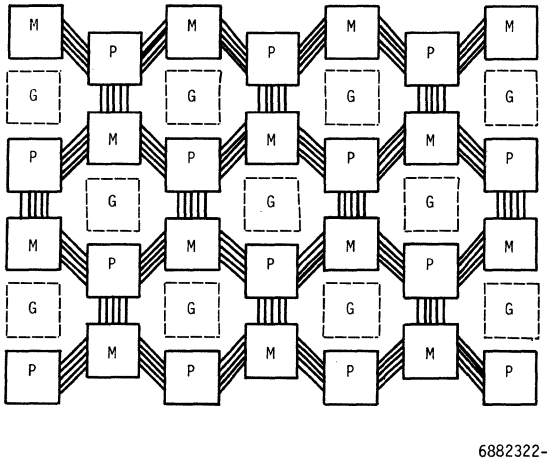


Figure 5.

Graph theoretic techniques can identify clumping properties of DFG's to reduce the volume of scheduling decisions made at run-time. Most appropriate to the model in Figure 4 is work by Stryker [8] on deriving global properties of DFG's from local properties of the nodes in the DFG, especially consumer-producer relationships. Informal analysis can produce additional results which is passed to the translator and run-time system as semi-formal advice, in the manner of pragmas in Ada.

Applications are loaded via mapping tricks, e.g. identity node creation, spliting, clumping and replication to heuristically squeeze the tasks into the "best" spot. If the "best" is not good enough, or if a chip fails, overloads result which must be gradually spread over nearby processors. A simple rule to achieve this is: Processors constantly grab as much work as they can find, with whatever reserve capacity they have to look for it. This is similar to "diffusion scheduling" used by Ward and Halstead [4] for control of the Munet.

The rectangular grid of the Web presents a problems. There must be four independent connections on each M and P chip. Each of these connections must be high bandwidth, leading to a P with 128 pins! The M is worse. While not unheard of, such pin counts are expensive in silicon area devoted to pin out loads. In any event, a dense rectangular packing of the M-P grid allows no space for the G chips. A very nice solution was proposed by Miller [6], using a hexagonal grid, which leads to the layout shown in Figure 5. All interconnections for the M-P grid can be achieved in a single layer, with extremely short connections. The longer radial and circular connection of the M-G grid require only one more connection layer.

References

[1] J. Dennis, Data Flow Super Computers, IEEE Computer Magazine Nov 1981.

[2] Tse-yun Feng, Survey of Interconnection Networks, IEEE Computer Magazine Dec 1981.

[3] H. Stone, S. Bokhari, Control of Distribuited Processes, IEEE Computer Magazine July 1978.

[4] R. Halstead Jr., S. Ward, The Munet: A Scalable Decentralized Archtitecture for Parallel Computation, Seventh Annual Symposium on Computer Architecture.

[5] A. Despain, D. Patterson, X-tree: A Tree Structured Multi-Processor Computer Architechture, Fifth Annual Symposium on Computer Architecture.

[6] J. Miller, private discussions of Jude's personal multi-processor project.

[7] US Navy, Enhanced Modular Signal Processor (EMSP), Procurement Request N00024-81-PR-28807, April 1981.

[8] D. Stryker, Graph Analysis in Data Flow Signal Processing, Intermetrics Internal R&D Technical Note, April 1981.

[9] J. Gutag, J. Horning, J. Wing, Some Remarks on Putting Formal Specifications to Productive Use, Forthcoming Xerox PARC Technical Report, April 1982.

[10] G. Fostel, A Hybrid Data Flow System, Intermetrics Internal R&D Technical Note, Feb 1982.

FUNCTION SHARING IN A STATIC DATA FLOW MACHINE[a]

Kenneth W. Todd
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract — Sharing a single copy of the body of a function among its invocation points in a program has been an important means of keeping down the size of large programs and thus enabling them to run on conventional computers. To do the same for programs run on a static data flow machine is also desirable but not easy because of the nature of the machine. This paper presents a scheme at the machine level of a static data flow machine for sharing a function among its activation sites which can be further modified to accommodate a variety of constraints. For programs using this scheme, space consumption is reduced but at the cost of an increased execution time.

The Static Data Flow Machine

With the advance in VLSI technology, it has become easier to make smaller and cheaper custom computer components. That, coupled with the current research on distributed systems and the desire to exploit parallelism in programs, has made data flow based computation attractive.

Unlike traditional computers based on von Neumann architecture, a data flow machine has no "program counter". Instead, an instruction executes when the values for all its operands have arrived. After execution, its result is sent to other instructions, possibly making some of these ready. Hence, instruction execution sequencing is based on the data dependencies among them and not on their location in the program memory. A high degree of parallelism is also obtainable since at any instant more than one instruction can be ready.

Data flow computers can be classified as either *static* or *dynamic*. Both classes base their execution on data flow principles, but the specifics vary. This paper is concerned with the static machine, which for the purposes of this paper differs from the dynamic machine in three ways. First, the static machine lacks a runtime loader — a program is loaded in its completed form. Second, an instruction can have at most one activation at any instant. Third, instructions and their operand values are stored together, making them not "pure". A more in-depth discussion on a static data flow machine can be found in [5]. For details on a dynamic data flow machine see [1] and [2].

Figure 1 shows the configuration of the static data flow machine currently under development by the Computation Structures Group of the Laboratory for Computer Science at MIT. It is constructed from two types of components: *processing elements* (PEs), which hold both program and data, and *2 X 2 routers*, which allow the PEs to communicate with each other. At present this prototype consists of four PEs and a network of four routers. By the adding more components the potential amount of parallelism obtainable can be arbitrarily increased.

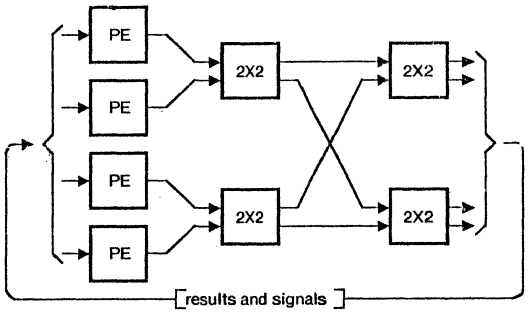


Figure 1. A Static Data Flow Machine

Instruction Cells

The basic unit of execution in the static data flow machine is an *instruction cell*. A graphical representation of a simplified instruction cell is shown in Figure 2 as a box with several fields. The top field contains the *opcode* of the cell. Directly below it are the *(initial) signals-needed value* field and the *signals-reset value* field, whose functions are deferred until later. At the bottom of the cell are fields that hold the *operand values*. From the right of the cell extend *result arcs* and *signal arcs*. A result arc, represented by a solid line, is used by a cell to send copies of the result to operand fields of other cells. Signal arcs, represented by dashed lines, are used by cells to simply signal each other. This example shows a cell that computes "*B* := *A* + 2". The actual cell used by the prototype is more complex and hence more powerful than the one presented here and is described in detail in [8].

A cell cannot execute (or *fire*) until it is *ready*. For this simplified version of a cell two conditions must be met: (1) the value of each operand must be present, either as a constant or a value received via a result arc from another cell; and (2) the signals-needed value must be zero. When a cell meets both conditions, its number (or address) is placed on a queue of ready cells maintained by its PE. Eventually the cell is fired, consuming the values of all non-constant operands in the process, sending copies of the result to operands of other cells as indicated by the results arcs, signaling other cells as indicated by the signal arcs, and overwriting the signals-needed field with the the signals-reset value.

It is often necessary to prohibit one cell from firing until after another cell has fired. For example, if cell *X* sends its result to an operand of cell *Y*, *X* should not fire again until *Y* has fired and is thus ready for another value from *X*. To insure this, the signals-reset value of *X* is set to 1 and a signal arc is established from *Y* back to *X*. When *X* fires, its signals-needed value is set to 1, and it cannot fire again until this value returns to zero. With each signal reception, its signals-needed value is decremented. When *Y* fires it signals *X* via the signal arc, causing the signals-needed value of *X* to turn zero, meaning that *X* can then safely refire. This example is so common that in order to keep the graphs readable, a signal arc that is associated with a result arc will be abbreviated by omitting the signal arc and replacing the arrow head of the result arc with a solid one.

As an example, Figure 3 shows three snapshots of the instruction cell program graph calculating "*C* := 2 * *A* + *B*". Snapshot #1 shows the state of the graph when values for *A* and *B* have arrived. Snapshot #2 shows the graph after the IMUL cell has fired. Snapshot #3 shows the graph state at the end of the computation when the IADD cell has fired.

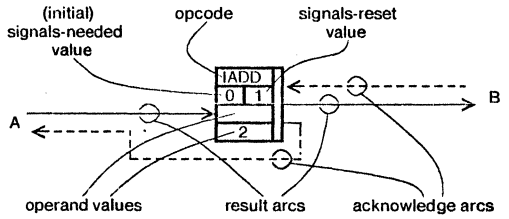


Figure 2. An Instruction Cell

(a) This research was supported by the National Science Foundation under grant no. MCS-7915255 and the Department of Energy under contract no. DE-AC02-79ER10473.

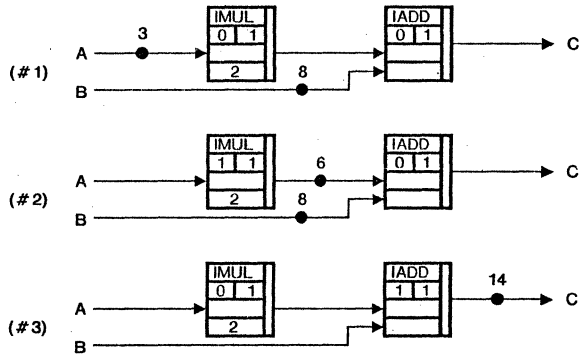


Figure 3. Three Shapshots of an Instruction Cell Program Graph

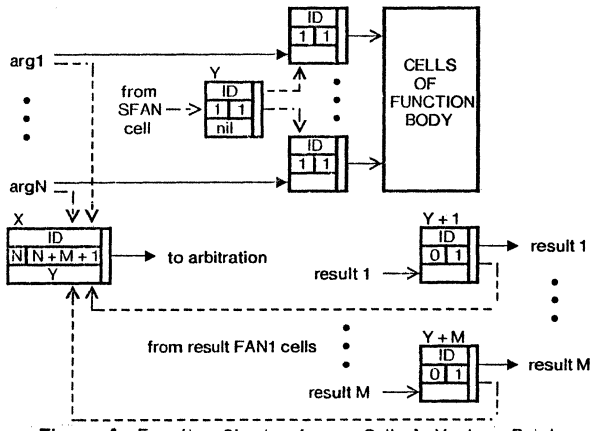


Figure 4. Function Sharing from a Caller's Vantage Point

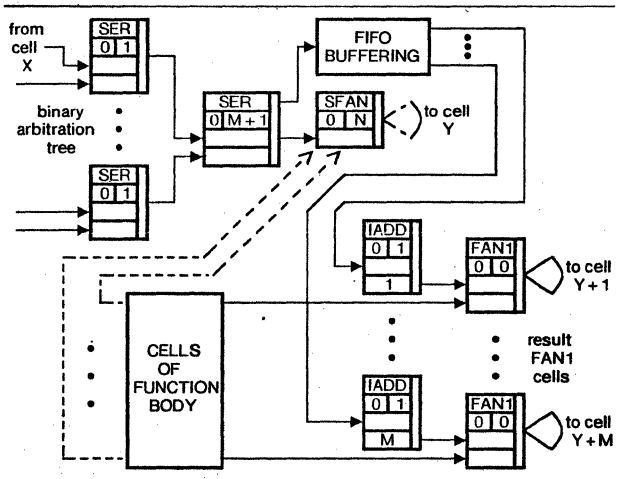


Figure 5. Function Sharing from the Function's Vantage Point

Functions

The high level source language from which instruction cells are generated for the prototype is VAL [3]. VAL is a functional, side effect free language designed primarily for numerical computations. Because instruction cells are functional in nature and since side effects would place restrictions on the sequence of instruction execution, VAL is ideal as a data flow source language.

VAL, like most high level languages, provides for function definitions. A survey of the methods used in both dynamic data flow and conventional machines to implement functions reveals that none can be successfully applied to the static data flow machine. The code of a function in conventional machines was originally impure and worked as long as the function had no more than one activation at a time. When recursion was added, the code needed to be pure and this was accomplished by moving the data and return address to a frame on a runtime stack. As for dynamic data flow machines, some link and load a fresh copy of the function body at the time of the call while others use colored tokens that permit multiple activations of cells.

A straightforward procedure for implementing function calls in the static data flow machine is to insert the body of the function at each point of invocation, the same process that a time-optimizing compiler for any language would do with a small side effect free function. However, as the number of invocation points increases and as the size of the function grows, such a process can result in a rather large instruction cell program generated from a comparatively small VAL program. Since a single cell consumes more memory than a corresponding instruction in a conventional computer (32 bytes in the prototype), it would not take long for this process to fill the memory of a data flow computer. In cases like this, a scheme for implementing the sharing of one copy of a function body among many or all of its invocation points can be advantageous since it would significantly reduce the number of cells generated.

To accomplish this sharing, four problems must be dealt with. First, arbitration must be performed among different invocation points simultaneously calling the function. Second, the data associated with each invocation must be kept separate. Third, a way must be established to determine to which invocation point to send the results. Fourth, deadlock must be avoided.

Figures 4 and 5 show how this sharing process can be implemented for a function that takes N arguments and returns M values. Figure 4 shows what is needed from the vantage point of a caller while Figure 5 shows what must be done on the part of the function. Starting from an invocation point in Figure 4, as each argument value of the function becomes available, cell number X is signaled and the value is stored in an ID (identity) cell where it waits for a signal to proceed into the function body. When all argument values are ready, the signals needed value of X is zero and it fires. This implies that the function call is *strict* in that an activation does not start execution of the body until all argument values are ready. To allow otherwise might result in mixing data of different activations.

When X does fire it sends the value "Y" to the SER (serializer or non-determinate merge) cell in Figure 5. The SER cell is a special case for the rules that determine when a cell is ready to fire because it needs only one operand, and the result it produces is the value of that operand. In case both operands are present, the SER arbitrarily chooses which one to use this firing and selects the other one next time. Using a binary tree of SER functions, arbitration between any number of simultaneous function calls can be achieved.

Eventually, the value "Y" reaches the root of the SER tree and is sent to the SFAN (signal fan) cell. This cell has the effect of creating a temporary signal arc between itself and the cell number specified by its lone operand, as indicated by the fan-shaped object in the figure. Thus, when this cell fires, a signal is sent to cell number Y.

Referring again to Figure 4, once Y has received this signal it fires and signals the ID cells holding the argument values. These cells in turn fire, sending their values into the function body. When the cells that receive the arguments have fired, they send signals back to the SFAN cell of Figure 5 as shown. This prevents a different invocation point from sending its argument values to the function body before it is safe to do so.

While the function is executing, it needs to remember where to send the results. Also, other invocations should be allowed to proceed with their function call when the body is ready. For this the first-in-first-out (FIFO) buffering of Figure 5 is introduced. This FIFO can be a chain of ID cells or some other construct that behaves like a queue. Because the function body also exhibits

FIFO-like behavior, if invocation *i* starts before invocation *j* then invocation *i* will terminate before invocation *j*. This will correctly match each set of function results with the corresponding value through the FIFO buffer.

After the results have been produced they must be sent to the caller. To accomplish this, the FANI (fan to operand 1) cell is used, a cell much like the SFAN except that the arc it creates is a result arc from itself to the first operand of the cell specified by the FANI's first operand, and the value sent over this arc is the FANI's second operand. For each result there is a FANI cell, and when the return value has been produced, it is sent to the second operand of its FANI cell as shown in Figure 5. There is also an ID cell at the activation point that is used as a receiver for this value and an IADD cell at the FIFO's end that is used to calculate the number of that ID cell. Thus, for the i^{th} result, "*i*" is added to the "*Y*" that eventually exits the FIFO and this sum is sent to the i^{th} FANI cell along with the i^{th} return value. When the FANI cell fires it sends this return value to cell *Y* + *i*.

One drawback to this scheme is that a single invocation point is prohibited from having concurrent activations. Suppose for example that invocations *A* and *B* share the same function with the results of *A* being fed to *B*, possibly indirectly. If *A* generates results faster than *B* can consume them, this stream of values will eventually extend back to the function body itself and thus prevent the completion of any more calls. If *B* then attempts a call, it too will not be able to complete. This would result in deadlock since *B* is waiting for *A* to complete its current calls while *A* is waiting for *B* to use the values it has sent. By including the signal arcs in Figure 4 from cell numbers *Y* + 1 through *Y* + *M* to *X*, a second activation of *A* will not start until the first one has completed and thus cleared the result receiver cells. This will prevent the backlog of values from stopping the output of the function body.

Pratical Considerations

Unfortunately, this last restriction degrades the efficiency of this function sharing scheme when used in a pipeline. If it can be determined at translation time that an invocation using a shared body is independent of all other invocations of that body, then this restriction can be lifted for that invocation.

If the function takes only one argument, a slight optimization can be performed by combining *Y* with the ID cell that holds the lone argument value. An increase in the maximum rate at which the function body could handle activations would result.

If many simultaneous calls to the function body are expected, then it will be desirable to make the body a maximal pipeline to achieve a high throughput rate. For simple expressions and conditionals, [4] describes an algorithm that achieves this by using buffering ID cells. For complex VAL constructs such as loops, a description of more complicated techniques required can be found in [7].

There maybe a limit to the signals-needed value; in the prototype it is 15. Because of cell number *X*, the sum of the number of arguments and return values is limited to 14, which is not too confining. The number of result and signal arcs might also be limited; in the prototype it is 6. This limits the number of arguments to 6 because of cell number *Y* — again, not a serious drawback. Functions that exceed either of these two limits should reduce its number of arguments (results) by combining them into a single record and passing (returning) the record pointer.

If addresses of cells are used instead of cell numbers, then cell number *Y* + *i* would be found by adding *i* times the size of a cell to the address of cell number *Y* instead of simply *i*, assuming that all cells are the same size. If not, then it may not be possible to calculate the return cell numbers at run time. In this case the original scheme can be modified by augmenting the function to take *M* additional arguments. The values of these arguments would be the addresses of cells *Y* + 1 though *Y* + *M* which are compile time computable constants and they would be sent via FIFO buffering to their respective FANI cells.

An example

A test of this sharing scheme has been performed on the following example:

$$tan(x) = sin(x) \,/\, cos(x) = cos(x - \pi/2) \,/\, cos(x)$$

Since the *cos* function is invoked from two different points, it is a candidate for function sharing.

For the prototype, two translations — shared and unshared — were derived for the *tan* function. A summary of these translations and the results of computing *tan*(1.0) are given below:

	Non-Sharing Translation	Sharing Translation	Sharing Improvement
Cells Generated	115 cells	75 cells	35%
Cells Executed	91 cells	114 cells	-25%
Passes Performed	33 passes	45 passes	-36%

(A *pass* is the simultaneous execution of every cells that is currently ready. It would be the order of the execution time of a program if each cell fired as soon as it was ready.) As the table shows, there is a time-space tradeoff involved. The size of the program is significantly reduced when sharing is used but at the cost of an increase in both the number of cells executed and passes performed. It should be noted that a large part of this increase is because the two *cos* calls are simultaneous.

Conclusion

Function sharing is not for every application. It can be a bottleneck in pipelines and in general increases the execution time of a program. However, it can significantly reduce the size of a program. In some cases the size reduction obtained would allow a program to be translated for and run on a data flow machine that would otherwise be too large.

Acknowledgments

The ideas of this paper were originally presented by Joe Stoy in [6]. I have developed them further both in [7] and this paper and have successfully applied them to the prototype. My thanks go to Professor Jack Dennis and to Dean Brock, Willie Lim, and Bill Ackerman for their helpful comments and suggestions in the preparation of this paper.

References

[1] Arvind and V. Kathail, "A Multiple Processor Dataflow Machine that Supports Generalized Procedures," *The Eighth Annual Symposium on Computer Architecture* (May, 1981), pp. 291-302.

[2] Arvind and K. P. Gostelow, "A Computer Capable of Exchanging Processors for Time," *Information Processing 77: Proceedings of IFIP Congress 77* (August, 1977), pp. 849-853.

[3] W. B. Ackerman. and J. B. Dennis, *VAL — A Value-Oriented Algorithmic Language Preliminary Reference Manual*, Laboratory for Computer Science, MIT, TR-218, (June, 1979).

[4] J. D. Brock and L. B. Montz, " Translation and Optimization of Data Flow Programs," *Proceedings of the 1979 International Conference of Parallel Processing*, (August, 1979), pp. 46-54.

[5] J. B. Dennis, "Data Flow Supercomputers," *Computer* (November, 1980), pp. 48-56.

[6] J. E. Stoy, "Functions in the Form 1 Data Flow Machine," Private communication (August, 1979).

[7] K. W. Todd, *High Level VAL Constructs in a Static Data Flow Machine*, Laboratory for Computer Science, MIT, TR-262, (June, 1981).

[8] K. W. Todd, *An Interpreter for Instruction Cells*, Laboratory for Computer Science, MIT, CSG Memo 208, (July, 1981).

SERFRE : A GENERAL-PURPOSE MULTI-PROCESSOR REDUCTION MACHINE.

F.-Y. VILLEMIN

DEPARTEMENT DE MATHEMATIQUES ET D'INFORMATIQUE CNAM 292 RUE SAINT MARTIN 75141 PARIS CEDEX03, FRANCE

ABSTRACT

FP, John Backus' applicative language naturally expresses concurency in programs. SERFRE evaluates FP programs by exploiting their built-in concurency as much as possible. In the single user version it has one I/O-processor (that either updates memories of the C-processors when a data or a program definition is given, or initiates a program evaluation and returns the result to the user), and many C-processors (that evaluate the programs). They are organized in modules, a module being a small number of C-processors and a strictly non-blocking communication device having at least two more ports than the number of C-processors in the module. When evaluating a program a C-processor detects concurency, calls for non-busy C-processors, and if any are available, initiates evaluation of concurent sub-programs on different C-processors (or else behaves like a sequential processor).

INTRODUCTION

An FP language (see J. Backus [Bac] for details) consists in a set X of objects, a set F of basic functions mapping objects into objects, a set A of function names and a definition operation def (def a = f means a is the name of the function f), a set C of function constructors forming new functions by combining objects, existing functions, and names of defined ones (Δ denotes composition) and, an execution command : (f : x means f is executed on the object x).
An FP program is such a function.
Some constructors can express concurency in programs,e.g. : construction , [$f_1,\ldots,f_n$] means $f_1,\ldots,f_n$ are to be executed concurently.
Programs in von-Neumann languages (FORTRAN, PASCAL ...) can be translated into FP programs [Vil], revealling their built-in concurency.

STRUCTURE OF SERFRE

SERFRE is a multi-processor command-driven (string reduction) machine having only a few different components (it is a VSLI architecture). It directly executes a FP language, trying to have sub-programs executed on different processors. It is a dynamic loosely-coupled system using direct communication with storage of messages.
Figure 1 describes the architecture of a possible, single-user implementation of SERFRE, and, figure 2 the structure of a module.
The I/O-processor either updates memories of the C-processors when a data or a program definition is given, or, initiates a program evaluation and returns the result to the user, or, takes care of local memories overflows by swapping on secondary storages.
Each C-processor has it own memory (working as a ROM for him) containing definitions of data and of programs (defined functions).
C-processor exchange messages of the form:<receiver address, program, data, sender address>. When a C-processor has to evaluate a function formed by a constructor involving concurency, it calls the (strictly non-blocking) communication device of its module which eventually calls other modules ones, for non-busy C-processors, and, sends them the concurent sub-programs and the data to execute and waits for them to return their results, if any are available, or else evaluates them sequentially.
A C-processor consists of a register for the return address (sender), a stack for the program (a place for each Δ-composed function), registers (variable-length arrays) for the data, and, a reduction engine which first takes the top of the program stack, then checks whether this is a basic function (and evaluates it on the data and puts the result in the data registers), or the name of a defined function in the memory (and puts its definition on the top of the program stack and carries on), or function formed using a constructor involving concurency (and calls for C-processors for evaluating the concurent sub-programs and waits for their results). It contains a stack for intermediate results in case of sequential evaluation of recursively defined functions, and, of concurent sub-programs.
When the program stack is empty, it calls the C-processor corresponding to the return-address and sends him the message < return address, empty, data, his address>.
A full description of several proposed implementations of SERFRE is given in the report submitted to the French Office of Patents.

OTHER DESIGNS

TRELEAVEN and al. review [T1,T2] the proposed demand-driven architectures.
SERFRE compares to TRELEAVEN and MOLES design [T2] but, it does not have the global memory bottle-neck and has a more powerfull mean of communication between processors.
MAGO'S design [MAG] is a tree organized system and seems to waste a lot of time in communication between processors,what we have tried to minimalize.

0190-3918/82/0000/0140$00.75 © 1982 IEEE

REFERENCES

[Bac] BACKUS, J. "Can programming be liberated from the von-Neumann style ? A functionnal style and its algebra of programs", Com. ACM 21, 8 (Aug. 78), 613-641.

[MAG] MAGO, G.A. " A network of microprocessors to execute reduction language" Int'l J. Computer and Information Sciences 8, 5 pp 349-385 and 6 pp 435)471 (1980).

[T1] TRELEAVEN, P. "VLSI Processor Architectures" Computer 15, 6 (June 82), 33, 45.

[T2] TRELEAVEN, P, BROWN BRIDGE D, and, HOPKINS, R. "Data-driven and Demand-driven Computer Architecture, Comp. Survey 14, 1 (Mar 82), 91, 138.

VIL VILLEMIN F.-Y. "Translation of FORTRAN programs into FP programs" Report CNAM-GRIP 81/03 (Revised version submitted for publication), 1981.

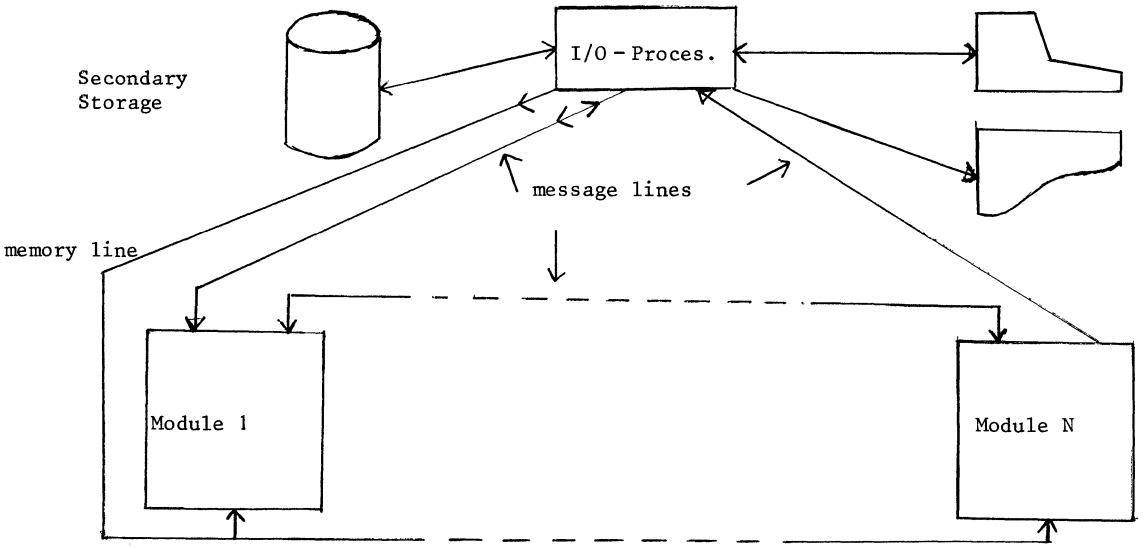


FIGURE 1

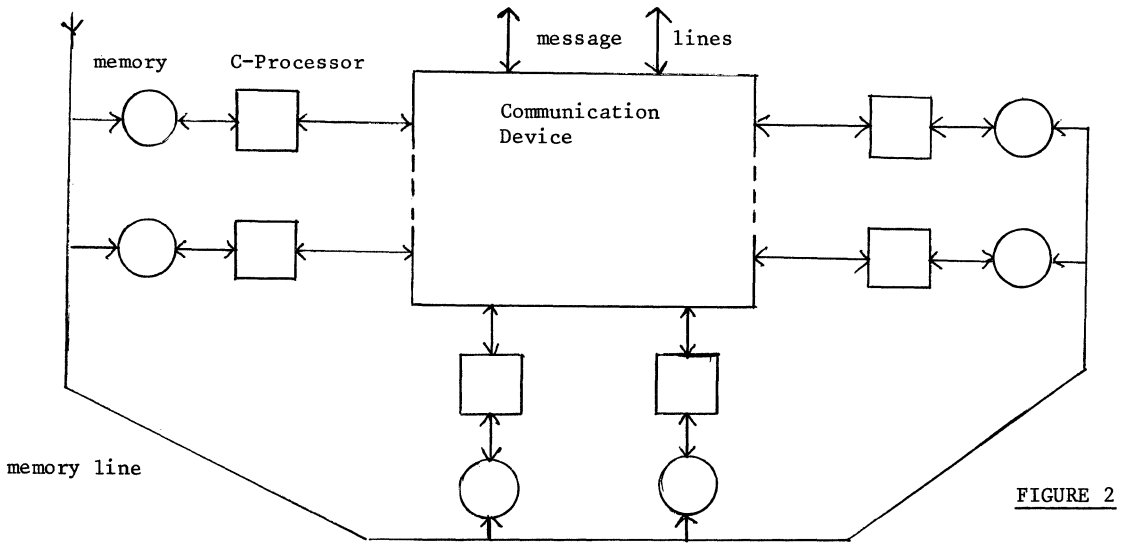


FIGURE 2

A LANGUAGE FOR SPECIFICATION AND PROGRAMMING OF RECONFIGURABLE PARALLEL COMPUTATION STRUCTURES

J. C. Browne, A Tripathi, S. Fedak, A. Adiga, R. Kapur
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

Abstract

The Computation Structures Language (CSL) defined and described herein is a vehicle for specification and programming of multi-type, multi-phase parallel computation architectures. The design principles for CSL include: (i) separation of parallel structuring from sequential computation, (ii) use of higher level language modules as the primitive execution elements, (iii) provision for multiple modes of data sharing and interprocess communication and (iv) capabilities for the replication of instantiations of program units and communication channels. The formal computation model for CSL is an extended form of colored Petri nets. The motivation for development of this programming system is the availability of reconfigurable network architectured computer systems which can implement multi-type/multi-phase parallel computer architectures. Specification of parallel architectures and programming of parallel architecture are separately discussed. The concepts are illustrated by examples.

Specification and Programming of Parallel Architectures

Micro-electronic technology allows computer architectures implementing high degrees of programmable parallelism of several types and even architectures capable of dynamically reconfiguring to different types and degrees of parallelism and communication geometry [KAR77, VIC79, SIE78, BRA79]. The Texas Reconfigurable Array Computer (TRAC) [SEJ80, PRE80, KAP80, JEN81] is a practical example of such an architecture. The Computation Structures Language (CSL) defined and described in this paper is a vehicle for exploiting such architectures. CSL implements both specification of parallel computation structures and programming of these parallel architectures. CSL supports dynamic structuring of computations through multiple phases each of which may display different types and degrees of concurrency and differencing requirements for sharing of data and interprocess communication. The model of computation implemented by CSL is that of an extended form [KAP82] of colored Petri nets [PET80]. A modeling system for analysis of the execution behavior of CSL programs has been developed utilizing this correspondence to colored Petri nets.

CSL is being developed in the context of the Texas Reconfigurable Array Computer Project (TRAC). Its implementation will utilize the unique capabilities of the TRAC architecture for representation of a wide spectrum of parallel computation architectures for dynamically reconfiguring itself between these parallel architectures. TRAC can implement any MIMD configuration of SISD or SIMD tasks within the span of its resource set. (See [SEJ80] for a brief description of TRAC.) The concept base of CSL is, however, in large measure independent of any particular architecture. CSL could be compiled for a conventional serial architecture or for a vector processor. The prototype CSL interpreter is indeed being developed in Pascal with simulation of parallel process executions on a DEC PDP-10.

A Rationale for Multi-type/Multi-phase Parallel Computation

It is well known that applicability of a computer architecture which implements any one fixed type of parallelism has been limited by the difficulty of mapping an extensive set of problems to execute efficiently on any single type of parallel structure. Vector streaming parallelism has been effective on many large scale numerical problems [VOI77,JOR77]. SIMD parallel of fixed degree and fixed interconnections structure has been found to be effective on a limited class of problems [SAM78,KUC77]. The limited effectiveness of these architectures has often been hard-won. There are three reasons for this historically experienced difficulty.

1. Mapping complex computations to a single architecture may lead to significant portions of the computations being based on high operation count algorithms.

2. It is often the case that a computation will pass through several phases each of which need a different parallel structure or different degrees of parallelism for efficient realization.

3. Mapping of the communication requirements of a computation upon a single fixed interconnection geometry may lead to heavy data movement costs [GEN78].

Recent investigations of the interconnection geometries required for efficient execution of such significant tasks as solution of Poisson's Equation [GRO79] as finite element equations [GAN81] have shown that no single type of interconnection network is suitable for these

0190-3918/82/0000/0142$00.75 © 1982 IEEE

problems. Kapur and Browne [KAP81] have decomposed the solution of block tri-diagonal linear systems into its natural computation structures and find three different basic modes of interconnection are required in the absence of a paracomputer architecture. (A paracomputer is a multiprocessor which implements conflict-free access to common memory [SCH80].)

These factors lead naturally to investigation of programming principles for multi-type/multi-phase parallel computation structures.

Design Principles for a Programming System for Multi-type/Multi-Phase Parallel Programming

MULTI-PHASE PARALLEL PROGRAMMING

Parallel programming adds to sequential programming the requirement for definition and programming of protocols to govern the interactions of the concurrently executing processes. A language system for parallel programming must therefore include the following capabilities above those for sequential programming:

*definition and control of concurrently executing processes

*definition of mechanisms for interprocess communication

*definition of mechanisms for correct and efficient sharing of data

Multi-type/multi-phase parallel programming adds the further requirement that the process and communication structures be specifiable at run time and also be reconfigurable as the computation progresses through its phases. It is also often necessary to pass results obtained in one phase to a later phase. We have also found in our attempts to write parallel programs that there is a need for convenient and flexible means of creating multiple instantiations of given program units and communication channels between program units.

CSL implements these requirements in a formulation based upon four design principles.

1. separation of specification and programming of concurrency and interprocess communication from the programming of sequential computation units

2. use of the separably compilable units of a higher level language as the execution units from which the parallel computation structures will be composed.

3. inclusion of both shared address space and address space data transfer modes of communication.

4. recognition that in parallel programming processes and communication channels must have structure and multiple occurrences just as does data in sequential programs.

These four principles are the foundation for a straightforward but flexible language system for efficient parallel programming and effective utilization of dynamic reconfiguration.

CSL deals only with definition and control of computation structures and specification and implementation of interprocess communication. Execution units (tasks or processes) are sequential programs which can be written in any language (Pascal for the current TRAC implementation). CSL provides capabilities for composing execution units into computation structures, for defining the mechanisms and protocols for communication between tasks and for initiating and controlling task executions. CSL provides only that computational power necessary to implement task control. This approach is to be contrasted to that of more conventional languages for parallel programming such as PL/I, ALGOL68, and Concurrent Pascal which have added specific concurrent control features to general purpose programming languages. CSL separates the programming of elementary execution units and the composition and control of computation structures. This separation of conceptually disjoint problem domains leads to a clear specification and programming interface.

Use of separately compilable units of higher level languages as the unit of composition for parallel computations allows flexibility in unit size. The execution units do not know whether they are executing independently or as a part of a concurrent process set. The CSL programmer does require knowledge of which data structures in the tasks will be shared or involved in communication. Indexing of programs is implemented as a means of replication of many program units executing on different data sets. This of course also requires the indexing of communication channels coupling the program occurrences.

CSL is a block structured language which uses macro-definitions as a code compression device. This choice was dictated by the convenience offered for the interpretive implementation planned. Macro declarations may be recursive.

CSL supports both the sharing of variables (overlapping address spaces) between sets of processes and data transfer between disjoint address spaces. TRAC architecture supports efficient implementation of both mechanisms [SEJ80]. Overlapping address spaces are supported by creating network connections linking a given physical memory module to more than one processor. The sharing of access is on a logical basis rather than a physical access basis. A

processor attaches a shared memory through a priviledged instruction. The attach will not be honored by the network hardware until the requested memory module has been released by its current holder (if it is currently attached). A segment of memory is thus switched from address space to address space rather being shared across otherwise disjoint address spaces. The logical concept of data sharing in CSL is, however, specified independently of the implementation model.

CSL can be thought of as representing the logical endpoint of an operating system job control language. A CSL program is a job control program for an environment of great flexibility. It represents a prototype for the job control language for many component and reconfigurable architectures.

The subsequent sections sketch and illustrate some of the capabilities of CSL. The Users Reference Manual [ADI81] gives a full definition of each statement in the language.

Specification of Parallel Architecture

The specifications for a computational architecture are written in terms of logical program elements such as tasks, shared data and logical channels between tasks rather than in terms of device or processor properties. The specifications for a structure of a given type and phase is bound by a CONSTRUCT statement. The architecture bound by a CONSTRUCT statement remains in effect until the execution path encounters another CONSTRUCT statement. CONSTRUCT statements can be nested. TASKS, CONDITIONS and CHANNELS, the elements of a computation structure, are defined within a CONSTRUCT statement. The effect of a CONSTRUCT is for the system resource scheduler to configure an architecture conforming to the specification and to map the TASKS and CHANNELS upon this architecture. A User's Reference Manual which gives examples of each statement as well as a full syntactic and semantic definition is available. The most effective exposition of CSL is, however, by example. Figure 1 is a CONSTRUCT statement for TASKS and shared variables taken from an example program given in full detail in Appendix A.

```
CONSTRUCT
  TASKS
   t2(i)   :  C[s(i),s(i+1),s(i+2)];
   t2(j-1) :  C[s(j-2),s(j-1),s(j)];
   t2(m)   :  C[s(m-1),s(m),s(m+1),s(m+2)]
                RANGE m = i+2 TO j-3 BY 2;
```

Figure 1: Example of a CSL Construct Statement

The first thing to notice is that the TASK declaration, t2(m), is based on indexing. It is often the case that many invocations of identical processing are required. Indexing gives a convenient method for specification of the number of identical process replicas and also for associating data with each invocation. C declares the file upon which the program code is to be found. The [s(i), s(i+1), s(i+2)] associates with t(i) shared data elements s(i), s(i+1) and s(i+2). The actual structure of s is defined within the task code. It might be a column of an array or an entire array. The declaration of s(i) s(i+1) and s(i+2) as associated with t2(i) notifies the system scheduler to establish a memory configuration where t2(i) can access s(i), s(i+1) and s(i+2). The RANGE declaration specifies the number of tasks and shared data elements to be instantiated in this configuration.

A CONDITION may be associated with each TASK. It becomes a variable shared by the TASK and the CSL program. CONDITION's are the only overlap between the address space of tasks and the controlling CSL program.

CHANNEL's implement 1 to N communications between tasks. DATACHANNEL's are declared for the movement of high volume data between task address spaces. MESSAGE CHANNEL's are declared for the movement of control information and low-volume data. Different implementations are used for the two channels constructs. There are a number of extended declaration capabilities such as specification of the number of buffers associated with a CHANNEL. Figure 2 is an example of a DATACHANNEL declaration taken from the program given in Appendix A.

```
CHANNELS
(moveright[i] = DATACHANNEL FROM f-c-p(i) TO
                              f-c-p((i+1) MOD (N+1));
moveleft[i] = DATACHANNEL FROM f-c-p(i) TO
                              f-c-p((i+N) MOD (N+1));)
      RANGE i = 0 TO N;
```

Figure 2: Channel Declarations

CONSTRUCT statements must appear prior to the execution of the tasks or use of the channels they define. The appearance of a CONSTRUCT statement in the execution path of a CSL program voids previous CONSTRUCT statements with release of all resources unless the CONSTRUCT statement encountered is contained in a nested parallel structure (nested COBEGIN, see Section 5). The RETAIN statement provides an exception to release of resources. The memories containing the data elements specified in a RETAIN statement will be (logically) retained in the system for further processing within a subsequently encountered CONSTRUCT statement.

Parallel Programming with CSL

The three additional tasks of parallel programming (over sequential programming) are:

1. establishment and control of concurrent execution of tasks
2. implementation of interprocess communication

3. control of access to shared data

CSL attempts to use as sparse a syntax set as is consistent with ease of programming.

We describe here on those language constructs relevant to parallel programming. Assignment, repetition and sequential control flow statements are minimal in number, Pascal-like in structure and will not be discussed herein.

Execution control is implemented by eight statements. EXECUTE, COBEGIN-COEND, TERMINATE, STOP, CONTINUE, WAIT, SIGNAL, RESET. Let us again refer to an example (Figure 3).

```
WITH s(m), s(m+1) DO EXECUTE t2(m)
   RANGE m = n TO j-1 BY 2;
```

Figure 3: Example of a WITH Statement

EXECUTE is followed by a list of task names. These tasks execute (logically) in parallel. Communication between the list of executing tasks can only be through shared variables. This is the characteristic statement for expressing SIMD executions. The statements contained between a COBEGIN-COEND pair are logically executed in parallel. They will often, however, contain synchronization statements such as WAIT/SIGNAL/RESET or CHANNEL commands such as SEND/RECEIVE. This is the mode for expressing MIMD or pipelined processing.

Access to shared data is governed by the WITH statement. The task(s) named in the execute begin execution whenever they can have exclusive access to the shared variables contained in the WITH statement. The WITH construct in Figure 3 is used for exclusive access to the variables s(m) and s(m+1) for Task t2(m) for even m. The WITH construct also implements in its extended formats read-only access and exclusive access to some data items and read-only to others.

The WHEN statement implements the Dijkstra [DIJ75] guarded command construct. It is normally used on conditions defined in the CSL or on the standard system generated conditions on CHANNELS. WAIT/SIGNAL/RESET complete the synchronization constructs. They are functions defined upon CONDITION variables and have the obvious semantics.

Interprocess communication between independently executing tasks is via CHANNELS. SEND and RECEIVE are illustrated in Figure 4.

```
 (SEND leftarr TO moveleft[i]
SEND rtarr TO moveright[i];
RECEIVE newright FROM moveleft[(i+1) MOD (N+1)];
  RECEIVE newleft FROM moveright[(i+N) MOD (N+1)]; )
    RANGE i = 0 TO N-1;
```

Figure 4: Example of Data Movement through CHANNELS

SEND places a data item on a channel. it is then available for the tasks declared as eligible to RECEIVE. A SEND blocks only if no buffer space is available while a RECEIVE blocks unless there is a data item on the channel specified in the command. A RECEIVE on a data channel "removes" the data from the channel and decrements the count of expected receives. The message is removed only when all expected RECEIVE's have been executed. The action of a SEND/RECEIVE is to transfer values between data structures defined in task address spaces. These data structures must be declared in the outermost block of the Pascal programs defining the tasks. Data appearing in shared data declarations cannot be sent via channels.

The CSL Computation Model

The existence of a formal computation model for a programming language offers a number of significant advantages. It allows an assessment of the power and applicability of the language. It guides the development and verification of correct programs. It provides the foundation for performance analysis of programs written in the language.

The formal computation model for CSL is an extended form of colored Petri nets [PET80]. The principal logical extension is to partition places into HOLD and ENABLE regions. This is illustrated in Figure 5.

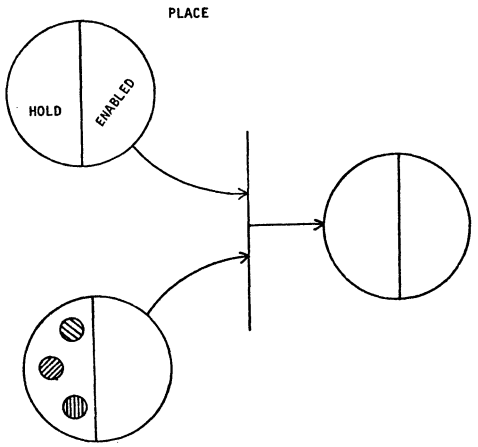


Figure 5: Extended Colored Petri Net Segment

The HOLD region simply holds tokens until the state of the transition and input places enables some one token in each place to participate in a firing. The algebraic representation of such a system is given by defining Set Operation Systems [KAP82]. Set operation systems generalize Vector Replacement Systems [KEL78] by allowing sets of "tokens" to be present at a place. The concept correspondence between CSL and colored Petri nets is given in Table 1. The actual modeling system also incorporates interval clocks in order to support performance evaluation of the execution of CSL programs.

CSL Constructs	Colored Petri Net Constructs
shared data elements, data elements sent through CHANNELS	colored tokens
TASKS	transitions
WITH statements, CHANNEL buffers Conditions	places

Table 1 - Correspondence between CSL constructs and colored Petri nets

Implementation Structure

A full feasibility demonstration for implementation of the CSL based programming system for parallel programming was executed before the design given here was adopted. Figures 6 and 7 schematically show the structure of the system and the relationship of the several components. The responsibility of the several components is as follows.

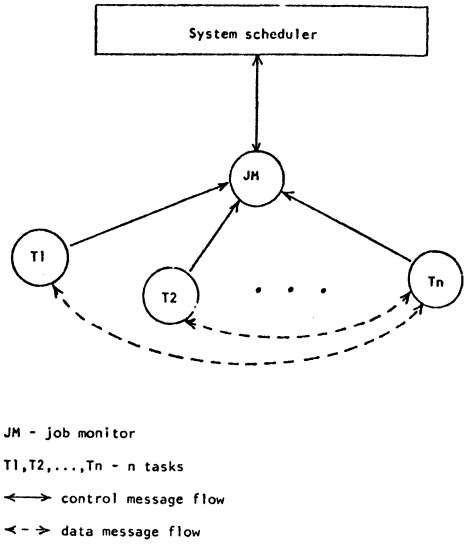


Figure 6

Each job, which consists of a reconfigurable set of tasks, is driven by a job monitor. The job monitor consists of four components. The CSL program specifies the configurations for the computational structure and the parallel process execution. The CSL run-time system is an interpreter for the declaration and executable statements of the CSL program. The configuration analyzer loads the CSL program and scans it for CONSTRUCT statements. The configuration analyzer

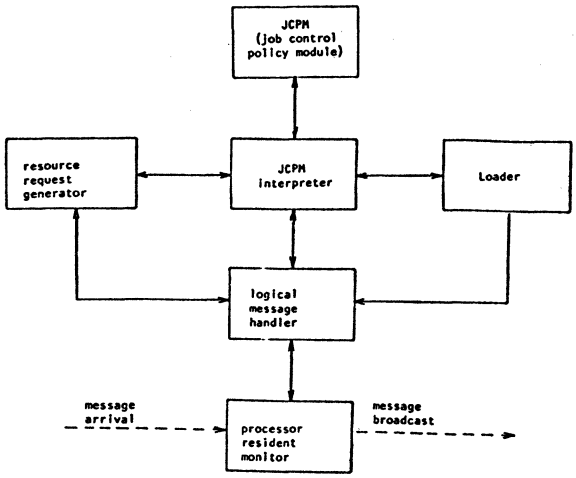


Figure 7

then negotiates with the system scheduler for the establishment of resources and parallel architectured structures to conform to the computational architecture specified in the CONSTRUCT statements. Whenever a configuration has been established, control is turned over to the appropriate executable portion of the CSL program. This program is then interpreted by the CSL run-time system. Control of the tasks of a CSL program is attained through the sending of messages to and from the CSL program running the job monitor and the tasks executing in the several task processors. Task executions are controlled through the sending of messages via packets in the TRAC network. SEND's and RECEIVE's on data channels are executed through the switchable memory concept of TRAC. SEND's and RECEIVE's on message channels are implemented by packet movement.

It is necessary for the processor resident monitor of each task to understand the data structures which it is to send and receive. Accordingly, the configuration analyzer initializes the processor resident monitor with the locations and characteristics of the data structures which it is to send and to receive. The initiation of transfers of data via the shared switchable memory mechanism is also initiated via packets being sent to the appropriate processor resident monitors. The details of this communication are given in a report, "Processor Resident Monitor of TRAC", [CAN81].

The implementation of the CSL interpreter has been completed on the DEC-10 in Pascal. It is serving as a simulator where the task code is replaced by dummy stubs. A design for the job configuration analyzer has been made down to the level of Pascal data structures and flow of control through functions and procedures. The

processor resident monitor has been resolved down to the streams of flow of control through the modules and functions for all major tasks including loading of task modules, initialization procedures, acquire and release of shared memories, handling of page faults and handling of packet arrivals.

Summary

This paper has described a programming system designed to exploit the capabilities of reconfigurable multi-processor architectures. The feasibility of this programming system has been established and a number of non-trivial programs coded in this system to demonstrate its applicability.

References

[ADI81] Adiga, A., Fedak, S., Tripathi, A. and Browne, J.C., "A Computation Structures Language: Revised", Preliminary Technical Report TRAC-27, Dept. of E.E., The University of Texas at Austin.

[BRA79] Brandjwan, A., Hernandez, J.A., Jaly, R. and Kruchten, P., "Overview of the ARCADE System", Proc. 6th Symp. on Comp. Arch. 7, pp. 42-49, 1979.

[CAN81] Canas, D., "The Kernel Monitor for TRAC", Preliminary TRAC Report, Oct. 1981.

[DIJ75] Dijkstra, E.W., "Guarded Commands, Nondeterminancy and Formal Derivation of Programs", CACM, Vol. 18, No. 8, Aug. 1975.

[GAN81] Gannon, D., "On Mapping Non-Uniform PDE Structures and Algorithms onto Uniform Array Architectures", Proc. Xth Int. Conf. on Parallel Processing, Aug. 1981, pp. 100-105.

[GEN78] Gentleman, W.M., "Some Complexity Results for Matrix Computations on Parallel Processors", Journal of the ACM, Vol. 25, 1978, pp. 112-115.

[GRO79] Grosch, C.E., "Performance Analysis of Poisson Solvers on Array Computers" in Supercomputers: Vol. 2, Ed. by C.R. Jesshope and R.W. Hockney, (Infotech, London, 1979) pp. 147-181.

[JEN81] Jenevein, R., DeGroot, D. and Lipovski, G.J., "A Hardware Support Mechanism for Scheduling Resources in a Parallel Machine Environment", Proc. 8th int. Symp. on Comp. Arch., pp. 57-66, 1981.

[JOR77] Jordan, T.L. and Fong, K., "Some Linear Algebraic Algorithms and their Performance on CRAY-1" in High Speed Computers and Algorithm Organization, Ed. by D.J. Kuck, D.H. Lawrie and A.H. Sameh (Academic Press, New York, 1977) pp. 313-316.

[KAP80] Kapur, R.N., Premkumar, U.V. and Lipovski, G.J., "Organization of the TRAC Processor-Memory Subsystem", AFIPS Conf. Proc., pp. 632-629, May 1980.

[KAP81] Kapur, R.N. and Browne, J.C., "Block Tridiagonal Linear Systems on a Reconfigurable Array Computer", Proc. 1981 Parallel Processing Conference.

[KAP82] Kapur, R.N., "On the Synthesis and Analysis of Reconfigurable Computer Programs", Ph.D. Dissertation, Department of Electrical Engineering, The University of Texas at Austin, May 1982.

[KAR77] Kartashev, S.I. and Kartashev, S.P, "A Multicomputer System with Software Reconfiguration of the Architecture", Proc. 8th Int. Conf. Computer Perf., pp. 271-286, 1977.

[KUC77] Kuck, D.J., "A Survey of Parallel Machine Organization and Programming", Computing Surveys 9, pp. 29-60, 1977.

[PET80] Peterson, J.L, "A Note on Colored Petri Nets", Inf. Proc. Lett., Vol. 11, No. 1, Aug. 29, 1980, pp. 40-43.

[PRE80] Premkumar, U.V., Kapur, R.N. and Lipovski, G.J., "Organization of the TRAC Processor-Memory Subsystem", AFIPS Conf. Proc., pp. 623-629, May 1980.

[SAM78] Sameh, A.H., "Parallel Numerical Algorithms" in High Speed Computer and Algorithm Organization, Ed. by D.J. Kuck, D.H. Lawrie and A.H. Sameh (Academic Press, New York, 1977) pp. 205-228.

[SCH80] Schwartz, J., "ULTRACOMPUTERS", ACM Topics 1, pp. 484-521, 1980.

[SEJ80] Sejnowski, M.C., Upchurch, E.T., Kapur, R.N., Charlu, D.P.S. and Lipovski, G.J., "An Overview of the Texas Reconfigurable Array Computer", AFIPS Conf. Proc., pp. 631-641, May 1980.

[SIE78] Siegal, H.J., Mueller, P.T. and Smalley, H.E., "Control of a Partitionable Multimicroprocessor System", Proc. Int. Conf. on Parallel Processing, pp. 9-17, 1978.

[VIC79] Vick, C.R., "A Dynamically Reconfigurable Distributed Computing System", Ph.D. Dissertation, Auburn University, 1979.

[VOI77] Voigt, R.G., "The Influence of Vector Computer Architecture on Numerical Algorithms" in High Speed Computer and Algorithm Organization, Ed. by D.J. Kuck, D.H. Lawrie and A.H. Sameh (Academic Press, New York, 1977) pp. 229-244.

Appendix A - An Example

This CSL program is a complete parallel formulation for a particle-in-cell code. The details of the problem formulation can be found in TRAC project technical report [BRO81]. The

major portion of the code (MACRO's oerr and oerb) is a parallel structuring of Odd-Even Reduction [KAP82]. The MACRO poisson combines oerr and oerb to solve Poisson's equation. The main program alternates calls to the Poisson equation solver and the routine f-c-p which computes fields and move charges. Each copy of f-c-p computes the field and moves the charges in a column of "cells". The copies of f-c-p must communicate with their nearest neighbors in order to compute fields and hand particles to the columns where they move in a given time step. The major execution steps of the program are contained in the last 10 lines of the program.

```
Program Pictst;

VAR
  i,j,N,col,p : INTEGER;

MACRO oerr(s,i,j,k);
{definition for reduction step of Odd even}

VAR   m : INTEGER;

BEGIN
   IF k>1 THEN
   {reduction step number}
   BEGIN
   {phase 1: diagonal block solution}
    CONSTRUCT
      TASKS  t1(m) : B[s(m)] RANGE m = i TO j;
    END;

    EXECUTE t1(m) RANGE m = i TO j;
   END;

   BEGIN
   {phase 2: merging neighboring rows }
   {using 2-pole/3-position switch     }
    CONSTRUCT
     TASKS
      t2(i)   : C[s(i),s(i+1),s(i+2)];
      t2(j-1) : C[s(j-2),s(j-1),s(j)];
      t2(m)   : C[s(m-1),s(m),s(m+1),s(m+2)]
                  RANGE m = i+2 TO j-3 BY 2;
    END;

    COBEGIN
     //WITH s(i) DO EXECUTE t2(i);
     //WITH s(m-1),s(m) DO EXECUTE t2(m)
         RANGE m = i+2 TO j-1 BY 2;
    COEND;

    FOR n = i TO i+1 DO
      WITH s(m),s(m+1) DO EXECUTE t2(m)
        RANGE m = n TO j-1 BY 2;

    COBEGIN
     //WITH s(m+1),s(m+2) DO EXECUTE t2(m)
          RANGE m = i TO j-3 by 2;
     //WITH s(j) DO EXECUTE t2(j-1);
    COEND;

   BEGIN
   {inverse perfect shuffle}
     CONSTRUCT
      TASKS
        t3(2m-1):D[s(2m-1),s(m)]
                  RANGE m = i TO j/2;
        t3(2m):D[s(2m),s(m+(j-i)/2)
                  RANGE m=i+1 TO j/2 -1;
     END; {construct}

     WITH s(m) DO EXECUTE t3(m) RANGE m = i TO j-1;

     COBEGIN
      //WITH s(m) DO EXECUTE t3(2m-1)
                  RANGE m = i TO j/2;
      //WITH s(m+(j-i)/2) DO EXECUTE t3(2m)
            RANGE m = i TO (j-1)/2;
     COEND;
   END;

   k := k - 1;
   RELEASE;
   oerr (s,(j+i)/2,j,k);
   {invoke next pass of reduction}
   CONSTRUCT
     TASKS solve : E;
   END;  {construct}
   EXECUTE solve;
   {solution of single block returned by reduction}

ENDMACRO;  {oerr}

MACRO oerb(s,j);  {back substitution for oer}

{j:block dimensionality of original matrix....}
{.........power of 2 minus 1                   }
{k:step number, initially, k=1                 }

VAR  m : INTEGER;

BEGIN
  FOR k = 1 TO (log(j+1)-1) DO
   BEGIN
     CONSTRUCT
       TASKS
         b(m) : F[s(m-(j+1)/2**(k+1)), s(m),
                  s(m+(j+1)/2**(k+1))]
                RANGE m =(j+1/2**(k+1)
                      TO j+1-(j+1)/2**(k+1)
                      BY (j+1)/2**(k+1);
        END;  {of construct}

    ( WITH s(m),s(m+(j+1)/2**(k+1)) DO EXECUTE b(m)
      WITH s(m-(j+1)/2**(k+1)),s(m) DO EXECUTE b(m))
        RANGE m=(j+1)/2**(k+1) TO j+1-(j+1)/2**(k+1)
                BY (j+1)/2**(k+1);
   END;  {for}
ENDMACRO;  {oerb}

MACRO poisson (s,j);

BEGIN
  oerr(s,1,j,log(j));
  oerb(s,j);
ENDMACRO; {poisson}

BEGIN {main program}
    { initialize
       p : the number of processes
       N : total number of columns in the grid
       col : no. of columns assigned to each process
       M : number of iterations required
      NOTE: 1) p = N/col
            2) each process or task also
               needs the two columns
               adjacent to those assigned to it.}
 CONSTRUCT
   TASKS
    f-c-p(0) (init,move,charge) : Cfile[qv(N),qv(j)]
                  RANGE j = 1 TO col+1;
    f-c-p(i) (init,move,charge) : Cfile[qv(i*col+j)]
              RANGE ((j=0 TO col+1),(i=1 TO N-2));
    f-c-p(p) (init,move,charge)
                  : Cfile[qv(1),qv((N-1)*col +j)]
              RANGE j = 0 TO col;
 END;  {of construct}
 (EXECUTE f-c-p(i).init;  {initialization}
  EXECUTE f-c-p(i).charge;
{maps charges from particle positions
 to mesh points  }
```

```
 FOR j = 1 TO M DO
  BEGIN
   poisson (qv,i);  {solves poisson equation}
  COBEGIN
  CONSTRUCT
  TASKS
   f-c-p(0) (init,move,charge) : Cfile[qv(N),qv(j)]
                  RANGE j = 1 TO col+1;
   f-c-p(i) (init,move,charge) : Cfile[qv(i*col+j)]
              RANGE ((j=0 TO col+1),(i=1 TO N-2));
   f-c-p(p) (init,move,charge)
                  : Cfile[qv(1),qv((N-1)*col +j)]
              RANGE j = 0 TO col;
   CHANNELS
   (moveright[i] = DATACHANNEL FROM f-c-p(i) TO
                          f-c-p((i+1) mod (N+1));
    moveleft[i] = DATACHANNEL FROM f-c-p(i) TO
                          f-c-p((i+N) MOD (N+1));)
          RANGE i = 0 TO N;
END;  {of construct}

 //( EXECUTE f-c-p(i).move;
     {each task moves its particles}
     SEND leftarr TO moveleft[i];
     SEND rtarr TO moveright[i];
     RECEIVE newright FROM
         moveleft[(i+1) MOD (N+1)];
     RECEIVE newleft FROM
         moveright[(i+N) MOD (N+1)];
    {information about particles that
     crossed partitions is sent to
     adjacent tasks }
     EXECUTE f-c-p(i).charge;)
     {calculates new charge distribution}
        RANGE i = 0 TO N-1;
COEND;
END;
END.
```

Appendix Reference

[BRO81] Browne, J.C., Kapur, R.N. and Adiga, A., "Particle-in-Cell Code Analysis", Technical Report TRAC-33, Department of E.E., The University of Texas at Austin, Fall 1981.

This work was supported by the National Science Foundation under Grant Number MCS77-20698 and by the Department of Energy under Grant Number DOE-AS05-81ER10987.

The second author is currently associated with the Corporate Computer Science Center, Honeywell, Incorporated.

ALGEBRA OF EVENTS : A MODEL FOR PARALLEL AND REAL TIME SYSTEMS

P. CASPI, N. HALBWACHS
IMAG Laboratory
Grenoble, FRANCE

Abstract: The model presented here differs from the usual models of parallel processing by two aspects: On one hand, it takes fully into account the metric notion of time, thus allowing the description of hard real time systems. On the other hand, it is a pure behavioural model, in the sense that it does not use any abstract machine notion. From a formalization of the notion of event, we show that the behaviour of a logical system may be described, by means of few operators, in a precise and concise way. The algebraic properties of the model are then studied, in order to define some methods for analysing or transforming systems described in this formalism.

INTRODUCTION

Two different notions of time are used in system modeling. In sequential systems, as far as time performance is not considered, the time concept may be reduced to the ordering of actions, or more generally of events occurring during the system life, that is a perfectly known total ordering relationship. In parallel systems, the ordering of events depends on the execution time of the actions. So a precise description of such a system needs the usual metric notion of time. However, since the execution times are generally uncknown, the correctness of parallel systems is commonly required to hold independently of any assumption about the speeds of the involved processors. So, many authors were led to consider the ordering of events in a parallel system as a partial ordering, and to assimilate parallel systems with undeterministic sequential ones. This approach allows to get rid of any metric notion of time, and has led to most of the parallel programs proof techniques. However it does not apply as soon as real time systems are considered. In such systems, the metric notion of time is used not only to compare the performances of several implementations, but also to decide of the adequacy of a system to its specifications.

Another characteristic of many approaches to parallel behaviour modeling (for instance [1],[7]) is the use of an abstract machine model, more or less derived from finite state automata. A behaviour is defined as an equivalence class upon the set of machines, and thus the proof of a system reduces to the proof of the equivalence between the abstract machines representing the specification and the implementation of the system. The drawbacks of such an operational approach for the initial specification process have been pointed out in [3]. In short, the specification language is generally far from being natural, and may lead to overspecification.

In this paper, we present a purely behavioural model for logical, parallel or real time systems, which takes fully into account the real time dependencies between internal and external events of a system. Our notion of time may be viewed as a simple ordering time, as far as purely parallel systems are considered, or as a metric time, assumed to be the global time of an external observer to the system.

In section 1, the basic notions of time and event are defined. An event is represented by an increasing staircase function from time to non negative integers, which counts the number of occurrences of the event during the time. An ordering relationship and a set of operators are provided in section 2, that structure the set of events as an ordered semiring. To illustrate the descriptive power of this algebra, we show (section 3) that finite state machines and Petri net models may be specified by systems of linear equations and inequalities over events. In order to define an effective calculus on such specifications, the algebra is extended in section 4 to become a ring, the elements of which are called pseudoevents. The use of this calculus to real time systems design problems is illustrated in section 5. Section 6 describes a systematic method to get approximate results about descriptions in our model, by means of discrete transforms of pseudoevents. Some nice properties of the algebra, when the time may be considered as discrete, are given in section 7. In conclusion, the extension of the model towards numerical systems is discussed, and open problems are set, the solution of which would greatly increase the capabilities of our calculus. Most proofs have been omitted, but may be found in an extended version of this paper [5].

1. TIME AND EVENTS

1.1 Time

Our notion of time refers to an absolute one, such as perceived by an external observer to the system. At the description level, the problem of the relative times measured by several subsystems'clocks in a distributed system, such as studied in [6], does not arise. We shall generally model the set $\mathbb{T}$ of times by the set $\mathbb{R}$ or $\mathbb{Z}$ of real or integer numbers. Elements of $\mathbb{T}$ are called times or instants when $\mathbb{T}$ is considered as an affine

This work was supported by C.N.R.S. under grant ATP-"Parallélisme,communication,synchronisation".

0190-3918/82/0000/0150$00.75 © 1982 IEEE

space, and time intervals, delays or durations when the vectorial structure of $\mathbb{T}$ is considered.

1.2 Events

We consider as events the transitions between states that may appear either in a system or in its environment, such as setting a switch, or assigning a new value to a variable. Moreover, an event may occur several times during the period of observation of the system, but, as we deal with discrete systems, the set of occurrences of an event is assumed to be enumerable. At a suitable level of abstraction, we can decide that an occurrence of an event has no duration, and can be viewed as a cut in the time line, that separates the times before and after the event occurs. Thus we define an event e to be a finite or infinite increasing sequence of instants, where e(n) denotes the instant of the n-th occurrence of e. We shall furthermore impose that, if the sequence is infinite, it converges towards $+\infty$ in $\mathbb{T}$ with n. This restriction is motivated by algebraic reasons and may be intuitively justified because, in discrete systems, an event may not occur infinitely often in a finite amount of time.

The number of occurrences of an event e will be noted #e. For convenience, we do not prevent an event from having several simultaneous occurrences. The set of events will be noted E($\mathbb{T}$), or simply E when the choice of $\mathbb{T}$ is irrelevant.

Of course, this definition of events copes with real time behaviour modeling. However, it is also convenient to describe sequential or purely parallel systems: For instance, if L is a language on a vocabulary V, we can associate with each symbol a in V and with each string c in L, the event â which is the increasing sequence of the ranks of the symbol a in c.

This representation by means of sequences allows us to equally handle the present, the past and the future of the system. This is close to the point of view adopted, for instance, in the applicative language LUCID [2].

1.3 Counters

An alternative way for handling events consists of using counters. Such counters have appeared useful in describing or programming synchronization between processes [10],[11]. With each event e, we shall associate a counter μ_e, which is an application from $\mathbb{T}$ to $\mathbb{N}$, defined as follows:

$$\forall t \in \mathbb{T},\quad \mu_e(t) = \max\{\, n \mid 1 \leq n \leq \#e \ \&\ e(n) < t \,\},$$

Thus $\mu_e(t)$ measures the number of occurrences of e that have happened strictly before t. μ_e is an increasing, left continuous staircase function on $\mathbb{T}$. Figure 1 pictures the counter of the event e=(1,3,4,6).

Let an event counter be an increasing, left continuous total function from $\mathbb{T}$ to $\mathbb{N}$, which

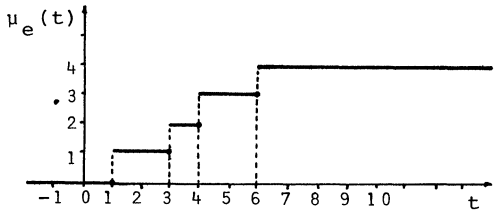


Figure 1

value is zero on some interval $]-\infty,x_o]$, then (using the Church's lambda notation), $\mu = \lambda e.\mu_e$ is obviously a bijection between the set of events and the set of event counters, since:

$$\forall n=1..\#e,\ e(n) = \max\{\, t \in \mathbb{T} \mid \mu_e(t) < n \}$$

2. THE ALGEBRA OF EVENTS

A logical system behaviour will be considered as a vector of interrelated events, and a system as a set of such behaviours. In this section, we shall see how to specify such a system by means of few operators over events.

2.1 Primary Events

If $k \in \mathbb{N}^*$, the primary event $\underline{k}$ is, by definition, the event which has exactly k occurrences, simultaneously happening at the instant zero:

$$\forall n=1..k,\ \underline{k}(n)=0$$

$$\mu_{\underline{k}}(t) = \text{if } t<0 \text{ then } 0 \text{ else } k$$

Since the instant zero will generally represent the initial instant in a system life, primary events will be often used to model initial states.

2.2 Ordering over E

For every e,f in E, let

$$e \leq f \iff \#e \leq \#f \ \&\ \forall n=1..\#e,\ e(n) \geq f(n)$$
$$\iff \forall t \in \mathbb{T},\ \mu_e(t) \leq \mu_f(t)$$

So the (partial) ordering over events coincides with the pointwise ordering over counters. This ordering will be useful, in particular, to represent causality relationships over events.

$(E,\leq)$ is a lattice, and we can define the inf and sup operators as follows:

$\forall e,f \in E$,

$\mu_{inf(e,f)} = \lambda t.\ \min(\mu_e(t),\mu_f(t))$

$\mu_{sup(e,f)} = \lambda t.\ \max(\mu_e(t),\mu_f(t))$

E has a minimum element $\underline{0}$, which is the event which has no occurrences ($\#\underline{0}=0$).

2.3 Sum and Difference of events

The sum of two events e and f is defined to be the event which occurs each time e or f occurs. More precisely, the sequence of occurrences of the event e+f is built by interleaving the sequences

of e and f, according to their temporal ordering. This notion can be easily formalized by means of counters, justifying the additive notation:

$$\mu_{e+f} = \lambda t.\ \mu_e(t) + \mu_f(t)$$

The + operation, being obviously commutative and associative, may be generalized to an arbitrary finite number of operands:

$$f = \Sigma_{i=0}^{k} e_i \iff \mu_f = \lambda t.\Sigma_{i=0}^{k} \mu_{e_i}(t)$$

The product of an event e by a natural integer k is the k times iterated sum of e:

$$ke = \Sigma_{i=1}^{k} e$$

The difference over events is only a partial operation, the definition of which results from the definition of the sum:

$$d = e-f \iff e = d+f$$

Note that the difference e-f is defined only if f is a subsequence of e.

2.4 Delay Operators

Let Δ be a delay, then the delay operator D^{Δ} performs a translation of every occurrence of its operand according to Δ:

$$\mu_{D^{\Delta}e} = \lambda t.\mu_e(t-\Delta)$$

The exponential notation is justified by the obvious properties that D^{0} is the identity operator on E, and that $D^{\Delta}D^{\delta} = D^{\Delta+\delta}$ for every delay Δ, δ . The operator D^{1} will be noted D.

3. APPLICATION TO BEHAVIOURAL DESCRIPTION

Let us show here that the preceding concepts are well suited to the description of parallel and real time systems, and lead to very concise descriptions of such systems.

3.1 Periodic Events

Let us express that an event e occurs at times 0, Δ, $2\Delta, \ldots, n\Delta, \ldots$. Clearly, e satisfies the following recursive definition:

$$e = D^{\Delta} e + \underline{1}$$

Similarly, the weaker assumption that e occurs at positive instants, and that two successive occurrences of e are separated by a delay smaller than Δ may be expressed as follows:

$$e \leqslant D^{\Delta} e + \underline{1}$$

3.2 Response Times

Let e be an input event to a system, and s be the output response to e, that is requested to occur within the time interval Δ following each occurrence of e. This can be expressed by:

$$D^{\Delta}e \leqslant s \leqslant e$$

These examples point out the usefulness of linear equations and inequalities over events. Evidence for such a fact will also be provided by the following application of our model to the behavioural description of finite state machines, Petri nets, and timed Petri nets.

3.3 Finite State Machine

Let $M = (V, Q, \sigma, q_0)$ a finite state machine, where:

- V is a finite vocabulary
- Q is a finite set of states
- σ is a mapping from $Q \times Q$ to V
- $q_0 \in Q$ is the initial state

A behaviour of M is a string $c = a_1 a_2 \ldots a_n \ldots$ of V^* such that there exists a sequence $q_0 q_1 \ldots q_n \ldots$ of states, such that, for every n smaller than the length of c, $\sigma(q_n, q_{n+1})$ exists and is equal to a_{n+1}. In our model, a behaviour of M will be a vector $(\hat{a} \mid a \in V)$ of events, such that $\hat{a}$ is the sequence of the ranks of the symbol a in a string like c.

First, we may describe, for every couple (q,q') of $Q \times Q$, the event "the transition $q \to q'$ is performed". Let $e_{qq'}$ be this event. For notational convenience, let $q^{\cdot}$ (resp. $^{\cdot}q$) be the set of states q' such that $\sigma(q,q')$ (resp. $\sigma(q',q)$) is defined.

Then, by observing that a state q is left at "instant" n if and only if it was reached at "instant" n-1 and it has some successor state, we get:

- For every state q, such that $q^{\cdot} \neq \emptyset$,

$$\Sigma_{q' \in q^{\cdot}} e_{qq'} = D\Sigma_{q'' \in {}^{\cdot}q} e_{q''q} + u(q)$$

where $u(q) =$ if $q = q_0$ then $\underline{1}$ else $\underline{0}$

Now, for every a in V, the event $\hat{a}$ happens each "time" a transition $q \to q'$ is performed, where $\sigma(q,q') = a$. So:

- For every a in V, $\hat{a} = \Sigma_{\sigma(q,q')=a} e_{qq'}$

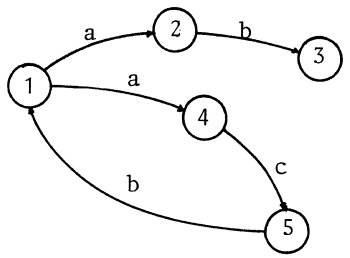


Figure 2

Example: Let us consider the state graph of figure 2. We get:

$$e_{12} + e_{14} = De_{51} + \underline{1} \quad , \quad e_{23} = De_{12}$$

$$e_{45} = De_{14} \quad , \quad e_{51} = De_{45}$$

and:

$$\hat{a} = e_{12} + e_{14} \quad , \quad \hat{b} = e_{23} + e_{51} \quad , \quad \hat{c} = e_{45}$$

from which it follows that:

$$\hat{a} = D^2\hat{c} + \underline{1} \quad \text{and} \quad \hat{b} = D^3\hat{c} + D\hat{c} - \hat{c} + \underline{D}$$

We shall see in section 7 a necessary and sufficient condition for the difference in the last equation to be defined. With this additional condition, the above equations exactly characterize the machine behaviours. Of course, the characterization by means of regular expressions is much simpler, but the same process applies to more complex machines, like communicating systems of [7],[8].

Now, let us see how the model applies to a parallel asynchronous language.

3.4. Petri Nets

Like state machines, Petri nets [9] only use an ordering notion of time. So we shall choose $\mathbb{T} = \mathbb{Z}$ and describe, for each transition of the net, the event "the transition is fired".

Notations: Let P be the set of places, T be the set of transitions. For each place p and each transition a, let us denote:
. $p^\bullet$ (resp. $^\bullet p$), the set of output (resp. input) transitions of p.
. $a^\bullet$ (resp. $^\bullet a$), the set of output (resp. input) places of a.
Let m(p,0) be the initial marking of p, and $\hat{a}$ be the event which happens each time the transition a is fired.

The transitions are fired one at a time, so the marking m(p,n) of the place p at the instant n is:

$$m(p,n) = m(p,0) + \Sigma_{b\in{}^\bullet p}\, \mu_{\hat{b}}(n-1) - \Sigma_{a\in p^\bullet}\, \mu_{\hat{a}}(n)$$

Writting that this marking may not become negative, we get:

$$\forall p \in P, \quad \Sigma_{a\in p^\bullet}\, \hat{a} \leqslant \Sigma_{b\in{}^\bullet p} D\hat{b} + \underline{m(p,0)} \qquad (1)$$

Now, we can write that at most one transition may be fired at each instant:

$$\Sigma_{a\in T}\, \hat{a} \leqslant \Sigma_{a\in T} D\hat{a} + \underline{1} \qquad (2)$$

(1) and (2) constitute a system of linear inequalities which characterize the set of correct behaviours of the net.

3.5 Timed Petri Nets

Of course, the preceding characterization of Petri nets may be extended to synchronous real time models such as timed Petri nets [13]. In such nets, a delay $\Delta(p)$ is associated with each place p. The two following rules differenciate timed Petri nets from ordinary ones:
. If a token reaches a place p at the instant t, it becomes unavailable until the instant $t+\Delta(p)$. A transition is enabled if and only if each of its input places contains an available token.
. A transition may not remain enabled during a non null interval of time: It must be either fired or disabled as soon as it is enabled.

The inequality (2) of ordinary Petri nets does not hold for timed nets, since several transitions may be simultaneously fired. Taking the first rule into account, the system (1) becomes:

$$\forall p \in P, \quad \Sigma_{a\in p^\bullet}\, \hat{a} \leqslant D^{\Delta(p)} \Sigma_{b\in{}^\bullet p}\, \hat{b} + \underline{m(p,0)}$$

The second rule forces every event to be as large as possible, so the above system must become:
$\forall a \in T$,

$$\hat{a} = \inf_{p\in{}^\bullet a}\left(D^{\Delta(p)} \Sigma_{b\in{}^\bullet p}\, \hat{b} + \underline{m(p,0)} - \Sigma_{c\in p^\bullet - \{a\}}\, \hat{c}\right.$$

This system of equations characterize the set of correct behaviours of the net only if it does not contain so called "no duration loop", i.e. if it is impossible for a token to participate in the firing of a transition and to come back simultaneously enabling this transition. Otherwise, the set of correct behaviours is only a subset of the solutions of the system of equations: For instance, if the delays associated with both places of the net of figure 3 are zeros, the only equation we get is $\hat{a} = \hat{b}$, though the true behaviour is $\hat{a} = \hat{b} = \underline{0}$, because of the null initial marking.

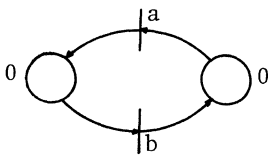


Figure 3

4. PSEUDO EVENTS

In the previous section, we have illustrated the descriptive power of the model. Let us now look for transformation and proof techniques for such descriptions. Starting with an equation such as

$$e = D^\Delta e + \underline{1}$$

the approach taken here consists of giving a sense to the expressions:

$$(1 - D^\Delta)e = \underline{1}$$

and

$$e = \frac{1}{1 - D^\Delta} = \Sigma_{n=0}^{\infty} D^{n\Delta}$$

This is achieved by extending the set of events so as to make the difference a total opera-

tor, and by defining an internal product.

Let us first note that, from the definitions of the sum and delay operators over events, the following identity holds for every event e:

$$e = \Sigma_{n=1}^{\#e} D^{e(n)} \underline{1}$$

Our extension of the set of events staightforwardly results from this identity.

4.1 Definition

A pseudo event is a formal series

$$x = \Sigma_{n=1}^{\#x} \bar{x}_n D^{x(n)}$$

where:
. $(\bar{x}_n)$ is a sequence of non null **relative** integers;
. (x_n) is a **strictly** increasing sequence of instants;
. both sequences have the same length #x, which can be finite or infinite, but in the latter case, the sequence (x_n) converges towards infinity.

The pseudo event 0 is such that #0=0. With each pseudo event x can be associated in a one to one way its counter μ_x defined by:

$$\mu_x = \lambda t.\ \text{if } x=0 \text{ then } 0 \text{ else } \Sigma_{x_n<t} \bar{x}_n$$

The set R of pseudo events is provided with the usual sum and product operators over formal series. $(R,+,\times)$ is an integral, commutative ring with neutral elements 0 and $1=D^0$.

A partial order is defined over pseudo events as follows:

$$a \leq b \ = \ \forall t \in \mathbb{T},\ \mu_a(t) \leq \mu_b(t)$$

$(R,\leq)$ is a lattice, and the sup and inf operators are the corresponding operators on counters.

An event is either 0 or a pseudo event with positive coefficients $\bar{x}_n$. Thus its counter is an increasing function of t. One can see that these definitions are consistent with the previous ones given in sections 1 and 2, with the following loosened notations:

Since, for every pseudo event a and every k in $\mathbb{N}$, ka=ka, we shall omit henceforth to subline the primary pseudo events. Since 1 is the neutral element of the product,it will be omitted in products. So D^Δ will denote the event $D^\Delta 1$. Notice that, with these notations, the expression $D^\Delta a$ may be viewed either as the D^Δ operator applied to a, or as the product of D^Δ by a. More generally, every pseudo event may be viewed as an operator on R.

The product operation, and the above notations justify the first step of the process of formal resolution of the equation $e=D^\Delta e+1$. The second step will be justified by the study of invertibility in R (a pseudo event a has an inverse if and only if there exists a' such that aa'=1).

4.2 Euclidean Division

4.2.1 Proposition: A necessary and sufficient condition for a pseudo event x to have an inverse is that $\bar{x}_1 = \pm 1$. Moreover,

$$\frac{1}{x} = \bar{x}_1 D^{-x(1)} \Sigma_{n>0} y^n ,$$

where

$$y = \text{sign}(-x_1) \Sigma_{n=2}^{\#x} \bar{x}_n D^{x(n)-x(1)} ,$$

and y^n denotes the n times iterated product of y by itself.

4.2.2 Corollaries:. Let a=1-e, where e is an event such that $e(1)>0$, then the inverse of a is an event, since $1/a = \Sigma_{n>0} e^n$.

. A necessary and sufficient condition for the inverse of an event e to be an event is that $e=D^\Delta$ for some Δ. Obviously $1/D^\Delta = D^{-\Delta}$. So $\{D^\Delta \mid \Delta \in \mathbb{T}\}$ is the set of unity elements of the semiring $(E,+,\times)$.

4.2.3 Ring norm: Let us recall that an application ν from a ring R to $\mathbb{N}$ is called a ring norm if and only if:
. $\nu(x) = 0 \iff x = 0$
. $\nu(xy) = \nu(x)\nu(y)$
. x has an inverse if and only if $\nu(x)=1$

So the application ν, which associates with each pseudo event $a\neq 0$ the integer $|\bar{a}_1|$, and such that $\nu(0)=0$ is a ring norm on R.

4.2.4 Proposition: R is an Euclidean ring, i.e. for every a,b in R, $(b\neq 0)$, there exist q,r in R such that a=bq+r and $\nu(r) < \nu(b)$.

Let us give the division algorithm, which is very close to the polynomials division according to increasing variables powers:

. Step 0: Let $r^{(0)} = a$ and $q^{(0)} = 0$;

. Step k+1: If $|\bar{r}_1^{(k)}| < |\bar{b}_1|$ then stop. Else ,let $x^{(k)} = \bar{r}_1^{(k)} / \bar{b}_1$. If $x^{(k)} \notin \mathbb{Z}$ then go to step α, else let:

$$p^{(k)} = x^{(k)} D^{r_{(1)}^{(k)} - b(1)} , \quad q^{(k+1)} = q^{(k)} + p^{(k)} ,$$

$$r^{(k+1)} = r^{(k)} - p^{(k)} b$$

. Step α: Let x be the smallest integer greater than $x^{(k)}$ if $x^{(k)} > 0$, the greatest integer smaller than $x^{(k)}$ otherwise. Let:

$$p = x D^{r_{(1)}^{(k)} - b(1)} , \quad q = q^{(k)} + p , \quad r = r^{(k)} - pb$$

4.3 Linear Inequalities of Pseudo Events

Our formal calculus is now powerful enough to solve any linear equation. However, behavioural specification in our model makes a very general use of linear inequalities, which are more diffi-

cult to handle because of the partial nature of the ordering on R. So, let us examine some properties of this ordering in relation with algebraic operators.

4.3.1 Inequalities and sum: For every a,b,c in R, $a \geqslant b \Rightarrow a+c \geqslant b+c$. In other words, the sum and difference operators are order preserving.

4.3.2 Inequalities and product: A great deal of works concerning ordered algebraic structures (see for instance [14]) make the hypothesis that positive product is order preserving, that is to say, that for every a,b,c:

$$a \geqslant b \;\&\; c>0 \Rightarrow ac \geqslant bc$$

This hypothesis is obviously false in R: For instance 1-D is positive, but $(1-D)^2 = 1-2D+D^2$ is not. So let us consider the set **Mon**(R) of order preserving pseudo events:

$$\mathbf{Mon}(R) = \{\, x \in R \mid a \in R \;\&\; a \geqslant 0 \Rightarrow ax \geqslant 0 \,\}$$

It can be easily shown that **Mon**(R) = E.

Example: Let us consider the two inequalities :

$$x(1-D^{\Delta}) \leqslant 1 \qquad (1)$$

$$x \leqslant \frac{1}{1-D^{\Delta}} \qquad (2)$$

(1) means that x cannot have two occurrences separated by a delay smaller than Δ (cf.3.1). Since $1/(1-D^{\Delta})$ is an event, we may multiply by it the two members of (1), so (1) implies (2). But the converse is false, because $1-D^{\Delta}$ is not an event: Figure 4 pictures an event satisfying (2) but not (1), with Δ=4.

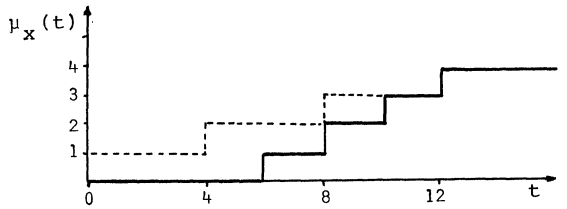


Figure 4

5. APPLICATION TO DESIGN PROBLEMS

In this section, we shall illustrate the use of the calculus on pseudo events on two simple problems.

5.1 First Example

A system receives two strictly periodic sequences of input requests. The former sequence starts from the instant 0, with a 2 seconds period, and the later one starts from the instant 1, with a 4 seconds period. The system is made of n identical processors, each of which takes 7 seconds for processing a request belonging to the former sequence, and 5 seconds for processing a request from the later one. This system may be represented by the timed Petri net of Figure 5.

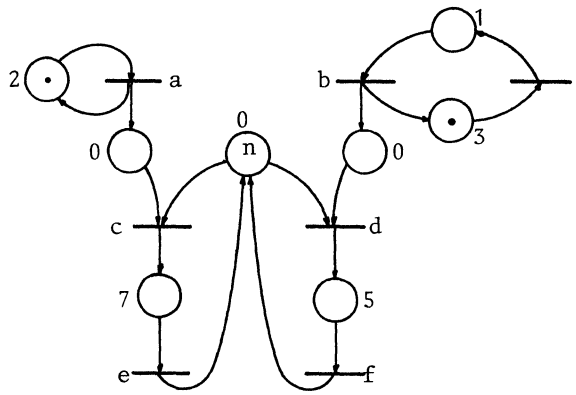


Figure 5

The question is: What is the minimum number of processors needed so as to take into account every request as soon as it happens.

In our model, this problem may be stated as follows: Let $\hat{a}$, $\hat{b}$ be the events respectively associated with input arrivals from each sequence. Let $\hat{c}$, $\hat{d}$ respectively represent the event "an input from the former (resp. later) sequence is taken into account by some processor", and $\hat{e}$, $\hat{f}$ respectively represent the event "a processor ends processing an input from the former (resp.later) sequence". Then:

. The specification of input sequences may be written:

$$\hat{a} = D^2\hat{a} + 1 \text{ and } \hat{b} = D^4\hat{b} + D \qquad (1)$$

. Since a request cannot be taken into account before its arrival, we have:

$$\hat{c} \leqslant \hat{a} \text{ and } \hat{d} \leqslant \hat{b} \qquad (2)$$

. The processing times of requests are specified as follows:

$$\hat{e} = D^7\hat{c} \text{ and } \hat{f} = D^5\hat{d} \qquad (3)$$

. As a request may only be taken into account when there exists an idle processor, we get:

$$\hat{c} + \hat{d} \leqslant \hat{e} + \hat{f} + n \qquad (4)$$

. Finally, the immediate handling requirement provides:

$$\hat{c} = \hat{a} \text{ and } \hat{d} = \hat{b} \qquad (5)$$

Now, (1) reduces to

$$\hat{a} = \frac{1}{1-D^2} \quad \text{and} \quad \hat{b} = \frac{D}{1-D^4}$$

So getting rid of any event variable, the problem

may be restated as follows:

"Find the least integer n, such that

$$\frac{1 - D^7}{1 - D^2} + \frac{D - D^6}{1 - D^4} \leq n \text{ "}$$

or "what is the maximum value of the counter of the pseudo event

$$x = \frac{1 + D + D^2 - D^6 - D^7 - D^9}{1 - D^4} \text{ "}$$

Now, we can perform the division in x, until getting:

$$x = 1 + D + D^2 + D^4 + D^5 - D^7 \frac{1 - D}{1 - D^4}$$

$-D^7(1-D)/(1-D^4)$ is a periodic pseudo event, the counter of which can easily be shown to have the maximum value 0. Thus, the maximum value of the counter of x is the one of $1+D+D^2+D^4+D^5$, which is 5 (see figure 6).So n=5 is the solution.

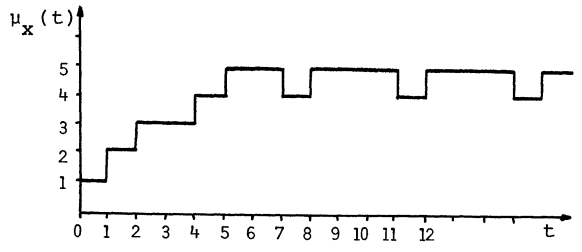


Figure 6

5.2 Second Example

Let us consider two processes p_1 and p_2, sharing an exclusive resource. Each process p_i cyclically asks for the resource, uses it during a delay δ_i , then releases the resource and works during a delay Δ_i, $(\delta_i, \Delta_i > 0)$, after what it comes back asking for the resource. This system is represented by the net of Figure 7.

Now assume that the resource is very expensive and is required to be permanently used. The problem is: What condition must satisfy the delays δ_1, Δ_1, δ_2, Δ_2, to achieve this requirement?

With the notations of the net, the problem may be stated as follows: Find a necessary and sufficient condition on the delays δ_i, Δ_i so that

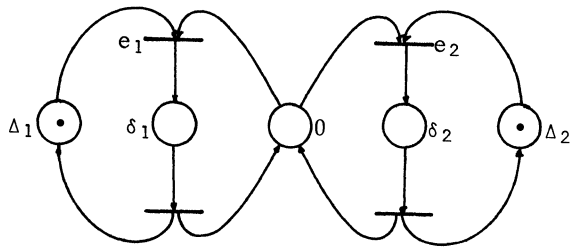


Figure 7

the following system S admits a solution (e_1,e_2) in E×E:

$$S \quad \begin{cases} e_i(1 - D^{\delta i+\Delta i}) \leq 1 \quad , \; i=1,2 \\ e_1(1 - D^{\delta 1}) + e_2(1 - D^{\delta 2}) = 1 \end{cases}$$

Now, since $e_1(1-D^{\delta 1}) + e_2(1-D^{\delta 2}) = 1$ which is an event, then $D^{\delta 1}e_1 + D^{\delta 2}e_2$ must be a subsequence of $e_1 + e_2$. On the other hand, since $\Delta_1 > 0$ and $e_1(1-D^{\delta 1+\Delta 1}) \leq 1$, $D^{\delta 1}e_1$ has no simultaneous occurrences with e_1, for one can easily show that for every integer n:

$$e_1(n+1) > (D^{\delta 1}e_1)(n) > e_1(n)$$

So $D^{\delta 1}e_1$ (respectively $D^{\delta 2}e_2$) must be a subsequence of e_2 (resp. e_1).

$e_1-D^{\delta 2}e_2$ and $e_2-D^{\delta 1}e_1$ are events and their sum equals 1, so one of them must be equal to 1 and the other to 0. Therefore:

$$S = S_{12} \text{ or } S_{21}, \text{ where}$$

$$S_{ij} = \{ e_i = \frac{D^{\delta j}}{1-D^{\delta 1+\delta 2}} \;\&\; e_j = \frac{1}{1-D^{\delta 1+\delta 2}} \;\&\; \frac{D^{\delta j}(1-D^{\delta i+\Delta i})}{1-D^{\delta 1+\delta 2}} \leq 1 \;\&\; \frac{1-D^{\delta j+\Delta j}}{1-D^{\delta 1+\delta 2}} \leq 1 \}$$

So, a solution satisfying S exists if and only if

$$\frac{1-D^{\delta 1+\Delta 1}}{1-D^{\delta 1+\delta 2}} \leq 1 \quad \text{and} \quad \frac{1-D^{\delta 2+\delta 2}}{1-D^{\delta 1+\delta 2}} \leq 1$$

which is equivalent to

$$\delta_1+\Delta_1 \leq \delta_1+\delta_2 \text{ and } \delta_2+\Delta_2 \leq \delta_1+\delta_2$$

The final necessary and sufficient condition is

$$\Delta_1 \leq \delta_2 \text{ and } \Delta_2 \leq \delta_1$$

6 APPROXIMATE ANALYSIS USING DISCRETE TRANSFORM

In section 5, we have given some examples of the use of the formal calculus in proving properties about behavioural specifications. Of course the proofs performed there may have appeared rather ad hoc, and are not susceptible of systematization. On the other hand, it has been shown in §4.3, that the non monotonicity of the product over pseudo events may give rise to difficult problems in dealing with linear inequalities. In this section, we shall propose a systematic method providing approximate results, even when such difficulties arise.

Our definition of pseudo events by means of formal series of the delay operator D is very close to discrete transform techniques widely used in the field of finite difference equations. Nevertheless, to our knowledge, those techniques never have been applied to inequalities.

6.1 Definition: For every pseudo event $a = \sum_{n=1}^{\#a} \bar{a}_n D^{a(n)}$, let us define the function ϕ_a from $\mathbb{R}^+$ to $\mathbb{R}$, by:

$$\phi_a = \lambda x. \sum_{n=1}^{\#a} \bar{a}_n x^{a(n)}$$

ϕ_a is generally a partial function, only defined on an interval $[0, r_a[$, where r_a is the convergence radius of the series.

6.2 Theorem: If a is a positive pseudo event, then ϕ_a is positive on the interval $]0, \min(1, r_a)]$. The converse is not true.

6.3 Example of application: Let us come back to example 5.2. We want the system S to have a solution, where

$$S = \begin{cases} e_i(1-D^{\Delta_i+\delta_i}) \leq 1, \ i=1,2 \\ e_1(1-D^{\delta_1}) + e_2(1-D^{\delta_2}) = 1 \end{cases}$$

Eliminating e_2, we get:

$$S = \begin{cases} e_1(1-D^{\Delta_1+\delta_1}) \leq 1 \\ \dfrac{D^{\delta_2}(1-D^{\Delta_2})}{1-D^{\delta_2}} \leq e_1 \dfrac{(1-D^{\delta_1})(1-D^{\Delta_2+\delta_2})}{1-D^{\delta_2}} \end{cases}$$

Now this system admits a solution e_1 only if there exists a real function ϕ ($=\phi_{e_1}$) such that, for every x in $[0,1[$:

$$\begin{cases} \phi(x)\ (1-x^{\Delta_1+\delta_1}) \leq 1 \\ \dfrac{x^{\delta_2}(1-x^{\Delta_2})}{1-x^{\delta_2}} \leq \phi(x)\ \dfrac{(1-x^{\delta_1})(1-x^{\Delta_2+\delta_2})}{1-x^{\delta_2}} \end{cases}$$

which is equivalent to:

$$\forall x \in [0,1[, \quad F(x) = \frac{x^{\delta_2}(1-x^{\Delta_2})(1-x^{\Delta_1+\delta_1})}{(1-x^{\delta_1})(1-x^{\delta_2+\Delta_2})} \leq 1$$

In the neighbourhood of x=1, $F(x) \sim \dfrac{\Delta_2(\delta_1+\Delta_1)}{\delta_1(\delta_2+\Delta_2)}$.

So a necessary condition for the system S to have a solution is:

$$\Delta_1 \Delta_2 \leq \delta_1 \delta_2$$

It is exactly the result provided by the method of [12] to find permanent behaviours of timed Petri nets. Notice that it is only a necessary condition, since the n.s.c. found in 5.2 was:

$$\Delta_1 \leq \delta_2 \text{ and } \Delta_2 \leq \delta_1$$

7 DISCRETE TIME

All the non real time, and most of the real time digital systems make use of a discrete notion of time. This motivates the investigation of particular properties of $R(\mathbb{Z})$ which is done in this section.

7.1 Discrete Derivatives

7.1.1 Definition: If a $R(\mathbb{Z})$, let us call the derivative of a the pseudo event a(1-D).

This denomination is motivated by the following - obvious, but very useful - proposition, which corresponds to the property of real functions, that a function is increasing if and only if its derivative is positive:

7.1.2 Proposition: A necessary and sufficient condition for a pseudo event a in $R(\mathbb{Z})$ to be an event, is that its derivative is a positive pseudo event.

Example: Let us come back to the example given in 3.3. As announced there, we are now able to express the condition on $\hat{c}$ for $D^3\hat{c} + D\hat{c} - \hat{c} + D$ to be an event, which is:

$$D(1-D) \geq \hat{c}(1-D-D^3)(1-D)$$

7.2 Linear Inequalities and Fixed Points

Notations: For each a,b in $R(\mathbb{Z})$, let us define:
- $[\![a) = \{x \in R(\mathbb{Z}) \mid a \leq x\}$
- $(b]\!] = \{x \in R(\mathbb{Z}) \mid x \leq b\}$
- $[\![a,b]\!] = [\![a) \cap (b]\!]$

7.2.1 Proposition: For each a,b in $R(\mathbb{Z})$, $[\![a)$ (respectively $(b]\!]$, $[\![a,b]\!]$) is a complete inf-closed semilattice (resp. sup-closed semilattice, lattice), i.e. every subset of $[\![a)$ (resp. $(b]\!]$, $[\![a,b]\!]$) has a greatest lower bound (resp. a least upper bound, a least upper bound and a greatest lower bound).

Notice that $R(\mathbb{R})$ does not satisfy this property: For instance, the sequence:

$$\left(x_n = D^{\frac{2n-2}{2n-1}} - D^{\frac{2n-1}{2n}}, n \in \mathbb{N}^* \right)$$

is included in $[\![\ 0,1-D\ ]\!]$, but has no least upper bound in $R(\mathbb{R})$.

7.2.2 Proposition: Let us recall that a function f from R to R is said to be latticecontinuous, if and only if, for every subset X of R admitting a least upper bound $\bar{x}$, (resp. a greatest lower bound $\underline{x}$) the set $\{f(x)\ x\ X\}$ admits a least upper bound $\bar{y}$ such that $\bar{y} = f(\bar{x})$ (resp. a greatest lower bound $\underline{y}$ such that $\underline{y} = f(\underline{x})$.

Then, for every Δ in $\mathbb{T}$ and every pair (f,g) of lattice continuous functions, the functions $\lambda x.D^{\Delta}x$, $\lambda x.f(x)+g(x)$, $\lambda x.\inf(f(x),g(x))$, $\lambda x.\sup(f(x),g(x))$ are lattice continuous.

7.2.3 Application: Let us consider a system of linear inequalities in $R(\mathbb{Z})$, of the following form:

$$S = \{\ x(1-e_i) \leq b_i,\ i=1..n\ \}$$

where all the e_i are events such that $e_i(1)>0$.

Then the set P of solutions of S is the set

of pre-fixed points of the function $f_S = \lambda x..\inf_{i=1..n}(b_i + e_i x)$, which is lattice continuous.

On the other hand, from 4.2.2 and 4.3.2, we have:

$$x(1-e_i) \leq b_i \Rightarrow x \leq b_i/(1-e_i)$$

So P is included in $(\beta]\!]$, with $\beta = \inf_{i=1..n}(b_i/(1-e_i))$. Since $(\beta]\!]$ is a sup-closed semilattice, if P is not empty, it admits a least upper bound $\bar{\beta}$. By Tarski's fixed point theorem, $\bar{\beta}$ is the greatest fixedpoint of f_S. Furthermore, the sequence $(f_S^i(\beta) \mid i \in \mathbb{N})$ is included in the complete lattice $[\![\bar{\beta}, \beta]\!]$ and by Kleene's fixed point theorem, it converges towards $\bar{\beta}$. Note that P is generally only included in $(\bar{\beta}]\!]$. The point is that by this process, we can add to S a new inequality, which is implied by S and may be saturated, since $\bar{\beta} \in P$.

Example:
Let us consider the following system of inequalities:

$$\begin{cases} \dfrac{x}{1-D} \leq \dfrac{1}{1-D^3} \\ x \leq 1 - \dfrac{D^4}{1+D} \end{cases}$$

Neither of the two inequalities may be saturated by x without violating the other. But the system reduces to:

$$X = \frac{x}{1-D} \ , \ X \leq f(X)$$

with $f = \lambda X.\ \inf(\dfrac{1}{1-D^3} \ , \ 1 - \dfrac{D^4}{1+D} + DX)$.
Using the above notations, we get:

$$\beta = \inf(\frac{1}{1-D^3}, \frac{1}{1-D} - \frac{D^4}{1-D^2}) = \frac{1}{1-D^3}$$

Let us compute the greatest fixed point $\bar{\beta}$ of f, smaller than β. We get:

$\beta_0 = f^0(\beta) = \beta = 1/(1-D^3)$
$\beta_1 = f^1(\beta)$
$= \inf(1/(1-D^3),\ 1-D^4/(1+D)+D/(1-D^3))$
$= 1 + (D^3+D^7)/(1-D^6)$ (see Figure 8)
$\beta_2 = f^2(\beta)$
$= \inf(1/(1-D^3),\ 1-D^4/(1+D) + D + (D^4+D^8)/(1-D^6))$
$= \beta_1$

So,

$$\bar{\beta} = 1 + \frac{D^3+D^7}{1-D^6}$$

and the initial system implies:

$$\frac{x}{1-D} \leq 1 + \frac{D^3+D^7}{1-D^6}$$

7.3 Application: Interruption Modeling

As a last illustration of the descriptive

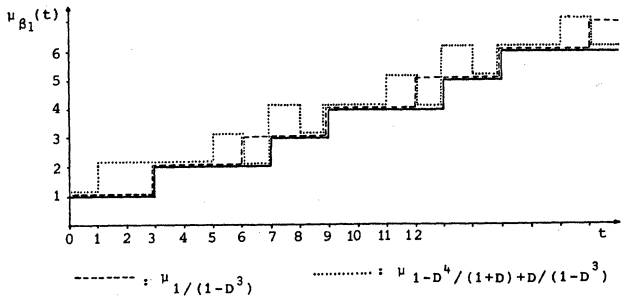


Figure 8

power of our calculus, let us consider the description of a task that needs a delay $\Delta \in \mathbb{N}$, but may be interrupted on every integer instant. The task is assumed to be non reentrent.

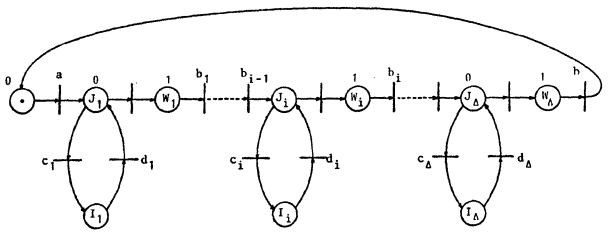


Figure 9

Modeling this task leads to a very complex timed Petri net (Figure 9). In this net, the transition a represents the beginning of the task. When the token reaches a place J_i, the task may be either immediately interrupted by the firing of c_i, then entering the interrupted state I_i until reactivated by the firing of d_i, else continuated for one unit of time in W_i before becoming again interruptible. b ($=b_\Delta$) represents the end of the task.

Proposition: Let $\hat{a},\hat{b},\hat{c},\hat{d}$ be the four events respectively representing the beginning, the end, the interruption and the reactivation of the task. Then, given $\hat{a},\hat{c},\hat{d}$, the event $\hat{b}$ is uniquely determined by the folowing relation:

$$0 \leq \frac{\hat{a} + \hat{d} - \hat{c} - \hat{b}}{1 - D} - \Delta\hat{b} \leq \Delta$$

The proof is rather tedious [5], but completely formal, and the result proved is not trivial and may be used to deal with systems with interruptible tasks in a very simpler way than by means of timed Petri nets.

CONCLUSION

This paper has presented a model for real time and parallel systems, and a set of results allowing, to some extend, the transformation and analysis of the description of these systems in the model. This work must be extended particularly in two directions:

First, the power of the calculus must be increased. We have shown that a great deal of problems involve investigations on systems of linear inequalities. For instance, let us consider two communucating asynchronous processes like in CCS [7]. Assume each process may be described by a system of linear inequalities over its external events. Then the resulting process will be described by the conjunction of the two systems, where the interprocesses communication events have been equalized and eliminated. So we must be able to eliminate a variable from a system of linear inequalities without loosing any information about the remaining variables. Furthermore, many problems, and particularly scheduling problems, may be expressed by linear optimization problems over (pseudo) events. But the partiel nature of the ordering relationship gives raise to a lot of difficult questions in applying linear programming techniques.

Another future extension concerns numerical systems. One way is to combine the results obtained by our calculus with classical techniques of program analysis. Another possibility is to extend the model to deal with variables. This was done in [4] for specification purposes, but the extension of the calculus to such a widened model is far to be obvious.

In spite of these questions, the model presented here seems to us a powerful tool to describe and analyse the behaviour of parallel and real time systems, and a unifying framework for a lot of problems in this field. Of course this approach is not considered as concurrent to the classical state-transition ones, but is expected to lead to complementary results.

REFERENCES

[1] C.André, F.Boeri, "The behaviour equivalence and its applications in Petri nets analysis". Journées d'étude AFCET, schemas de contrôle des systèmes informatiques. Paris (September 1979).

[2] E.A.Ashcroft, W.W.Wadge, "LUCID: A non procedural language with iteration". CACM, vol.20, n°7 (July 1977).

[3] P.Caspi, N.Halbwachs, M.Moalla, "Approche comportementale pour la spécification des systèmes temps réèl". Journée d'étude AFCET, spécification. Toulouse (September 1980).

[4] P.Caspi, N.Halbwachs, An approach to real time systems modeling. R.R.n°253, IMAG Laboratory, Grenoble (June 1981).

[5] P.Caspi, N.Halbwachs, Algebra of events: A model for parallel and real time systems. R.R.n°285, IMAG Laboratory (January 1982).

[6] L.Lamport, "Time clocks, and the ordering of events in a distributed system". CACM, vol.21, n°7 (July 1978).

[7] R.Milner, A calculus of communicating systems. Lecture notes in computer science, n°92. Springer Verlag (1980).

[8] R.Milner, "On relating synchrony and asynchrony", CSR-75-80, Edimburgh University (November 1980).

[9] J.L.Peterson, "Petri nets", ACM Computing Surveys, vol.9, n°3 (September 1977).

[10] D.P.Reed, R.K.Kanodia, "Synchronization with eventcounts and sequencers", CACM, vol.22, n°2 (February 1979).

[11] P.Robert, J.P.Verjus, "Towards autonomous description of synchronization modules". Proc. IFIP Congress, Toronto (1977).

[12] J.Sifakis, "Use of Petri nets for performance evaluation". Measuring,modeling and evaluating computer systems. North Holland Pub. Co. (1977).

[13] J.Sifakis, Le contrôle des systèmes asynchrones: Concepts, propriétés, analyse statique. Thesis, Grenoble University (June 1979).

[14] U.Zimermann, Linear and combinatorial optimization in ordered algebraic structures. Annals of discrete mathematics, n°10, North Holland Pub. Co. (1981).

RESOURCE EXPRESSIONS FOR APPLICATIVE LANGUAGES

Bharadwaj Jayaraman
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27514

Robert M. Keller
Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

Abstract -- A high-level approach to resource management in the framework of an applicative language is presented. A resource is defined as a linguistic construct that may be used either to exercise control over the concurrent evaluation of functions, or to serve as an interface to files, databases, etc. The specification of this control is achieved by *resource expressions*. Resource expressions are closely related to path expressions in their basic approach to specification of constraints, but differ in their semantics and implementation. The semantics of resource expressions is based on the concept of execution graphs and residues, and an implementation has been constructed using a set of queueing primitives for a demand-driven execution model.

1. INTRODUCTION

The results presented here are motivated by a desire to develop resource management primitives which mesh well with an applicative programming language. In retrospect, the language constructs to be described also work well for ordinary languages, but there are other options for those languages which are not attractive for applicative languages.

The major advantage of an applicative language for distributed systems is that no special care need be taken in exploiting available concurrency; the results of a program are guaranteed to be well-defined, independent of system timings. However, the concept of "resource management" for such a language may still be relevant, on two grounds:

1. It may be necessary to control the amount of concurrency which would occur naturally within the execution of a program, as concurrent evaluation of functions require additional resources (e.g. memory) to support.
2. It may be desirable to augment an applicative language with constructs to enable efficient interfacing with files and databases, i.e., structures that may change because of side-effects. Techniques are then needed to encapsulate such side-effects and make them interface cleanly with "pure" applicative code.

After some experimentation with various encapsulation methods for resource control (resembling monitors, serializers, etc. [3, 13, 14]), it was decided that such methods do not fit well within the framework of applicative languages, as they require the introduction of operational notions such as queues, messages, etc., and also do not lend themselves to a convenient denotational semantics. On the other hand, expression-based languages, such as path-expressions and their variants [1, 6, 7, 12] are attractive for three reasons:

1. They may be composed from primitive constructs similar to the way functional expressions are composed, and hence are compatible with the applicative style of programming.
2. They possess "bracketing" qualities similar to function evaluation, i.e., it is not necessary for the programmer to indicate explicitly the start and stop of an action. Rather, these events are contained in the notion of a functional expression being evaluated as a unit.
3. They have been derived from the notion of regular expressions [21] which can be considered as a denotational description of finite automata, suggesting that their extensions to resource control (which require transcending the finite-state languages) might also be amenable to a denotational semantics.

We describe here an expression-based language extension called *resource expressions*. Its uses, semantics, and implementation are the subjects of this paper. Resource expressions are closely related to path expressions in their basic approach to specification of constraints, but differ in their semantics and implementation.

There have been some attempts to introduce the concept of a resource in an applicative framework: Arvind et. al. [2] present dataflow monitors as a means for defining a resource and its scheduling, and Gurd and Catto [11] present some implementation ideas for dataflow monitors. In comparison, resource expressions are a higher level means of specifying resource control, since certain types of scheduling disciplines are expressed more succintly in resource expressions. However, the expressiveness of resource expressions in their current form is more limited compared to dataflow monitors.

Friedman and Wise [10] introduce an indeterminate operator *frons* for constructing a multiset, the order of whose elements is determined only when the multiset is accessed. Although frons may be used to express solutions to a variety of problems requiring the use of indeterminate merging, the issues of resource control are not handled at the level at which frons is used.

Another type of approach, the use of pseudo-functions [16], is attractive, but is less structured than the one presented here. However, pseudo-function constructs are employed in the implementation of our current model.

2. LINKING RESOURCES TO FUNCTIONAL EXPRESSIONS

Programs in applicative languages are presented as expressions denoting the application of functions to their arguments. We refer to these as *functional expressions* to distinguish them from resource expressions, which form the main topic of this paper.

Suppose that we desired to exert greater control over the evaluation of various functions. We could use

This material is based upon work supported by grant MCS 81-06177 from the National Science Foundation.

0190-3918/82/0000/0160$00.75 © 1982 IEEE

some device which explicitly sequences those functions [14, 17]. However, if the functions are evaluated in unpredictable order or are embedded within very large expressions, it is then desirable not to impose a rigid sequencing, but rather to impose a system of *constraints* on functions, e.g. that certain subexpressions do not get evaluated concurrently, etc. This has the effect of giving greater freedom on the order of expression evaluation, in the case where it is difficult to determine a priori orderings giving the right amount of concurrency. These constraints are expressed herein as resource expressions, and we think of them as providing a kind of "synchronizing overlay" on a functional expression.

To indicate the invocation of a function f which is controlled by a resource, we use res('f, args), in place of the usual f(args). Here res is a pseudo-functional object which represents an instance of the resource, and is created by evaluating a pseudo-function specifying the actual resource. Since the function being evaluated is encapsulated inside the resource, the quoted f is used to avoid lexical scoping violations. A variable denoting the quoted f could be used instead. The definition of the actual resource takes the form

```
RESOURCE ...resource name... parameters...
CONSTRAINT
      ...resource expression...
WHERE
   ACCESS ...function definition...
        (with optional IMPORTS)
   ACCESS ...function definition...
        (with optional IMPORTS)
      ...
END
```

where ACCESS is used to identify functions that may be invoked from outside the resource. The current version of the language extensions also allows nested resource definitions, but we will not be concerned with them in this paper.

Once a resource is instantiated, it may then be accessed by so-called *tokens*. A token is a request (or demand) to evaluate some function controlled by the resource, along with the actual parameters, if any, needed for this evaluation. The term *token class* will be used to refer to the function that is controlled by the resource.

We now sketch two examples illustrating different uses of resource expressions: the first illustrates how to interface with a database, and the second, how to control the amount of concurrency arising in concurrent evaluation of functions.

A skeletal example of a resource manager that encapsulates a shared database accessed by "read" and "write" operations is

```
RESOURCE database_manager(database)
CONSTRAINT
   (write*+[read])#
WHERE
   ACCESS write(...) IMPORTS database
   ACCESS read(...) IMPORTS database
END
```

The resource expression here enforces the well-known readers-and-writers constraint [8]. The subexpression write* allows a sequence of arbitrarily many write's and the subexpression [read] allows in parallel arbitrarily many read's. Since "+" denotes nondeterministic selection and "#" denotes non-terminating sequential repetition, it follows that read's and write's always exclude one another. If db1 and db2 are two distinct databases, then identical but independent managers for each could be created by equations

```
LET res1 = database_manager(db1),
    res2 = database_manager(db2)
```

The two databases are synchronized independently using functional expressions such as res1('read), res2('write, val), etc.

As a second example, consider the concurrent computation evoked by the following function definitions:

```
FUNCTION main(x)
RESULT IF x=0 THEN 0 ELSE
        f(x)*g(x)/h(x) + main(x-1)
WHERE FUNCTION f(x) ...
      FUNCTION g(x) ...
      FUNCTION h(x) ...
END
```

Suppose we wished to constrain the evaluation of f and g (but not h) so that only one of them is evaluated at any given time; in other words, f and g must be executed in mutual exclusion of one another. We may express this constraint as follows:

```
FUNCTION main(x)
LET res=mutex()
RESULT IF x=0 THEN 0 ELSE
      res('f, x)*res('g,x)/h(x)
      + main(x-1)
WHERE RESOURCE mutex()
      CONSTRAINT (f+g)#
      WHERE ACCESS f(x) ...
            ACCESS g(x) ...
      END
      FUNCTION h(x) ...
END
```

If we wished to constrain h also, and further wished to allow arbitrarily many h's to follow f or g, we could use the expression $((f+g).h^{\bullet})^{\#}$, where "." denotes sequencing. (The definition of h would now also have to be encapsulated inside mutex.)

If, in addition to the constraints of the preceding example, we were willing to allow arbitrarily many h's to proceed concurrently with themselves, we would use $((f+g).[h])^{\#}$.

To summarize the available constructs, we present the syntax of resource expressions accompanied by a brief informal semantics. Each individual token class is a resource expression. Furthermore, if R and S are resource expressions, then so are

R+S denoting the non-deterministic choice of either R or S as alternatives; the alternative chosen must be "satisfied" by the availability of tokens.

R.S denoting the sequencing of R followed by S only when there are sufficient tokens to satisfy both R and S.

$R^{\bullet}$ denoting a non-deterministic choice of an arbitrary number of sequential repetitions of R; the number of repetitions depends on the number of available tokens.

$R^{\#}$ similar R^{*}, except no non-deterministic choice is involved; # does not terminate.

[R] similar to R^{*}, except that consecutive repetitions may be done in parallel.

{R} similar to $R^{\#}$, except that consecutive repetitions may be done in parallel.

It should be mentioned that the number of repetitions in the above *repetitive* expressions, i.e. R^{*}, R^{*}, {R}, and [R], includes zero. Hence the number of tokens needed to satisfy such expressions is zero. Also, the meaning of our sequencing operator "." is different from the sequencing operator ";" used in path expressions. We elaborate on this distinction in the next section.

3. A BASIS FOR FORMAL SEMANTICS

Our motivation for formalizing the meaning of resource expressions is to provide a precise specification not only for the user, but also the implementor. Most attempts at giving a semantics for expression-based languages have been informal [6, 7], operational [1, 19], or formal-language based [4, 23, 20]. Of these the formal-language semantics is of interest here, since it comes closest to an acceptable denotational definition for resource expressions.

Semantics based on formal languages define the meaning of an expression to be a set of allowable execution sequences of tokens, derived solely from the expression. In general, one considers a set of partial orders on tokens, rather than sequences, to account for concurrent execution. However, for sake of simplicity of presentation, we will use sequences instead of partial orders in the subsequent discussions in this section.

Such a semantics implicitly assumes that one is only interested in "consistent" behaviors, i.e. the sequence of all tokens allowed to execute must be a prefix of some member of the above set. However, the notion of "completeness" is stronger, i.e., any sequence of tokens allowed to execute must be exactly equal to some member of the above set. We will refer to such such a sequence as a *complete* sequence.

The notions of consistency and completeness are expressed by the following two semantic models:

1. Consistency can be realized by an *expedient* approach, which chooses any alternative of the expression which is partially satisfied by the available tokens.
2. Completeness, on the other hand, can only be realized by a *prudent* approach, which chooses an alternative of the expression that is completely satisfied by the available tokens.

As an example, consider the expression (a.b + c.a). Assuming that an "a" and a "c" token are available, an expedient approach may choose the sub-expression "a.b", even though no "b" token had arrived. The "c" token will then not be executed. A prudent approach would prefer the sub-expression "c.a", since it will permit both "a" and "c" to be executed. With the expedient approach, it is possible to get blocked after "a", since a "b" token may never arrive.

Thus, although the expedient approach is more efficient, since it provides faster response to certain tokens, it may fail to execute complete sequences, even when there are sufficient tokens. A prudent approach, on the other hand, generally would take longer to decide what to do with a given collection of tokens, but offers the advantage of always being able to execute complete sequences.

We use a prudent approach for resource expressions consisting of a single sequence, e.g. a.b.a.c would require two a's, one b, and one c token to be present before it is chosen as an alternative. To provide the efficiency of the expedient approach, we have introduced the construct "/" for *commit* which can be used in place of a "." in a sequence. The meaning here is that only enough tokens to enable execution of the prefix up to the "/" are required for committing to the entire sequence. Thus, in a.b/a.c one "a" and one "b" would suffice, and the subsequent "a" and "c" would be processed when they arrive, but would not hold up the first "a" and "b" for their arrival. In this way we give the user the capability of choosing "shades" of expedience and prudence.

Our semantics will be as if there were an implicit commit after the body of each of the repetitive expessions, as well as one at the end of a top-level expression.

It should be noted that a/b is not equivalent to a+a.b, and hence the commit construct can't be simulated using simply sequence and alternation. To see this, compare the behavior of these two expressions on the input tokens {a,b}. The expression a/b will allow both a and b, whereas the expression a+a.b will allow one of two possible outcomes due to the non-determinism of "+": either only a, or both a and b. Thus their behaviors are not equivalent. Alternatively, compare the expressions (a+a.b)* and (a/b)*: the former allows a sequence of only a's, whereas the set of sequences allowed by the latter is exactly the set prefixes of ababab...

Semantics of path expressions define only the consistency requirement, and therefore may be said to use the expedient approach [1, 6, 7, 12]. The commit construct "/" is equivalent to the ";" of path expressions, but the effect of our sequence construct "." is not achievable in path expressions. However, by adding the device of "predicates" [1], it appears that the effect could be achieved. This device could also be used to overcome some limitations of expression-based control described in [5], viz. the inability to specify constraints based on the state of the resource, parameters of tokens, etc. The proper integration of such devices into expression-based control for applicative languages is still a subject of our investigation.

4. FORMAL SEMANTICS

To provide a formal semantics which reflects completeness as well as consistency, we must take into account the collection of input tokens available to the resource expression. Since we wish to define the behavior of repetitive expressions inductively, we must not only define the allowable order of tokens, but also the collection of tokens remaining after each repetition.

We therefore define the behavior of a resource expression for any bag of input tokens T as a set of pairs of the form <g, r>, where g is an *execution graph* and r is a bag of *residues*. These pairs will henceforth be referred to as g-r pairs. We use a bag, rather than a set, since we use to represent inputs which have several tokens belonging to the same token class.

An execution graph is a generalization of an execution sequence, and is defined by the functions SEQ and PAR, which have the following meanings:

SEQ(x,y) : x is executed before y
PAR(x,y) : x is executed concurrently with y

where x and y represent either tokens or execution graphs composed of SEQ and PAR. Both x and y must have completed their execution in order for PAR(x,y) or SEQ(x,y) to complete their execution. For PAR(x,y), x and y need not have started execution at the same time.

The residue r is the bag of tokens T minus the tokens used in defining the execution graph g.

To simplify the definition of its semantics, a resource expression is first converted into an equivalent *normal form*. We then define the semantics of normalized resource expressions inductively by showing how the set of g-r pairs can be constructedme examples illustrating our construction.

The normal form is a set of *alternative prefixed-sequences*, where *prefixed-sequences* are of the form $x_1/\cdots/x_m$ where "/" is the commit construct, and *alternatives* are specified by "+". Each x_i, except for x_1, is a set of *alternative sequences*, where *sequences* are of the form $y_1.\cdots.y_n$. However, x_1 is a sequence of the form $y_1.\cdots.y_n$ (with no alternatives). Finally, each y_i is either an atom or a repetitive expression whose body has been expressed in normal form. Examples of the normal form are shown below:

a.b.a
a.b/(a + b/a)
a* .b.{c} + a/(a + b)
{a/b.c + (a+b)* .b}

Examples of expressions <u>not</u> in the normal form are the following:

(a + b) . (c + d)
(a + b) / c
((a+b)*)*

The normal form is derived by transforming a given resource expression using two sets of equalities. The first set is the following:

P . (Q + R) = (P . Q) + (P . R)
(P + Q) . R = (P . R) + (Q . R)
(P + Q) / R = (P / R) + (Q / R)

where P, Q, and R are assumed to be arbitrary expressions. A notable exception, however, is that P / (Q + R) is not equal to P / Q + P / R. Consider, for example, the meanings of the two expressions a/(b+c) and a/b + a/c: the former expression specifies that the selection of b or c is to be made after a token for a is evaluated, whereas the latter implicitly selects b or c even before a is evaluated. Thus their operational meanings differ.

The second set of equalities relates the repetitive constructs. A partial list is the following:

((R)*)*=(R)*
((R)*)#=(R)#
[[R]] = [R]
{[R]} = {R}
...

In converting a resource expression into the normal form, a subexpression satisfying a form given on the LHS of the above equalities is replaced by the corresponding RHS.

We define the semantics of a normalized resource expression N for a bag of input tokens T by constructing a set L(N,T) inductively. We illustrate the construction for alternatives, prefixed-sequences, and the repetitive expressions only; a complete treatment may be found in [15].

1. For alternatives, the set of g-r pairs is the union of the set for each term. The union is taken to reflect the non-determinism of "+".
2. For prefixed sequences, the g-r pairs for the sequence up to the first commit "/" are first constructed. For each residue r in the above set, the g-r pairs for the subexpression up to the second commit are obtained, etc. The resulting execution graph is obtained by sequencing (using SEQ) the execution graphs of each term; the resulting residue is that of the last term.
3. Finally, for repetitive expressions, there are basically two cases: a) for "*" and "[]", the g-r pairs will also include the pair <e,T>, where e is the null graph and T is the input bag of tokens, whereas for "#" and "{}" this pair will not be included. b) For "*" and "#", sequences are constructed using SEQ, whereas for "[]" and "{}" execution graphs are constructed using PAR. In all cases, the set of g-r pairs is constructed inductively: the residue from the first repetition being used as the bag of input tokens for the second, etc.

We express the semantics of the above three types of expressions more formally as follows:

1. Consider $N = x_1 + x_2 \ldots + x_n$, where x_i's are prefixed sequences. We define
$L(N,T) = L(x_1,T) \cup L(x_2, T) \cup \ldots \cup L(x_n, T)$
to be the set of g-r pairs for N, assuming $L(x_i, T)$ is the set of g-r pairs for each x_i.
2. Consider $N = x_1 / x_2 / \ldots / x_n$, where x_1 is an ordinary sequence, and all other x_i are sets of alternative sequences. Then we define L(N,T) inductively as follows: Let
$L(x_1/\ldots/x_{n-1},T) = \{<g_i,r_i> \mid i=1,k\}$, and
for i=1,k, $L(x_n,r_i) = \{<g_{ij},r_{ij}> \mid j=1,k_i\}$.
Then we define
$L(N,T) = \cup_{i=1,k} \cup_{j=1,k_i}\{<SEQ(g_i,g_{ij}),r_{ij}>\}$.
3. Consider N = [x] where x is any normalized resource expression. Let
$L(x,T) = \{<g_i,r_i> \mid i = 1,k\}$, and
for i = 1,k, let $L(N, r_i) = \{<g_{ij},r_{ij}>, j = 1,r_i\}$.
Then we define
$L(N,T) = \cup_{i=1,k} \cup_{j=1,m_i}\{<PAR(g_i,g_{ij}),r_{ij}>\} \cup <e,T>$,
where e represents the null graph.

We illustrate the set of g-r pairs for some simple resource expressions. We use bag(...) to denote a bag of tokens; bag() is the empty bag.

(1) N = a/(b + c)
T = bag(a,b)
L(N,T) = {<SEQ(a,b), bag()>}

(2) N = a/b + a/c
T = bag(a,b)
L(N,R) = {<SEQ(a,b), bag()>
<a, bag(b)>}

(3) N = c.[a + b].a
T = bag(a,a,b,c)
L(N,R) = {<SEQ(c,a), bag(a,b)>,
<SEQ(c,a,a), bag(b)>,
<SEQ(c,b,a), bag(a)>,
<SEQ(c,PAR(b,a),a), bag()>}

5. IMPLEMENTATION OF RESOURCE EXPRESSIONS

There are two important steps in the implementation of resource expressions:

1. The expressions are represented in an intermediate form which consists of a set of condition-action pairs, similar to guarded commands [9]. (However, we are not relying on an existing implementation of guarded commands in our implementation.)
2. The target language program for a given intermediate form is constructed in a modular form by translating conditions and actions separately, and then combining the resulting programs together. Each repetitive expression is translated as a single recursive procedure, and the top-level expression is translated as a single procedure, if it is not a repetitive expression.

The next three sections describe the intermediate form, the target language primitives, and the translation respectively.

5.1. INTERMEDIATE FORM

The intermediate form is derived using the semantics of the different types of normalized resource expressions, viz. sequences, prefixed-sequences, alternatives, and repetitive expressions. In each case we determine a condition that must be satisfied before the corresponding action is taken. This condition is a conjunction of numeric thresholds for each token class, and indicates the minimum number of tokens of each class that must be present in order to take the corresponding action. The action specifies the actual sequence of tokens to be served. We first briefly explain how the condition-action pairs for a normalized resource expression are derived.

For a sequence, the condition is determined by considering only its atomic terms, i.e. excluding all repetitive expressions in the sequence. Repetitive expressions do not participate in the construction of the threshold condition because they permit zero repetitions of the body to occur, and therefore have a trivially satisfiable threshold condition. However, when a repetitive expression is encountered during the action, the condition corresponding to the body of the repetitive expression will be tested to see if any further repetitions are possible. Thus the actual number of repetitions that occur depend on the number of available tokens at this time.

For a prefixed-sequence, the condition is that determined by the (ordinary) sequence up to the first commit construct in the prefixed-sequence. The condition corresponding to the remainder of the prefixed-sequence is tested only after the action corresponding to the initial prefix has been taken. It should be noted that (COMMIT ...) may occur only as the last term in a sequence of length > 1.

The intermediate form for a set of alternatives such as $w_1+w_2+\cdots+w_n$ is $((c_1a_1)(c_2a_2)\cdots(c_na_n))$, where (c_ia_i) is the intermediate form of w_i.

The intermediate form of repetitive expression r is of the form
(REPEAT ...) where
REPEAT = If r = (x)* then STAR else
If r = (x)# then POUND else
If r = [x] then BRACKET else
If r = {x} then BRACE
and the dots represent the intermediate form x.

For example, the intermediate form of {a.b.a/b + a.[a+c].c} may be derived from the above rules to yield

(BRACE ((c1 a1) (c2 a2))) where
c1 = ((2 a) (1 b))
a1 = (a b a (COMMIT (((1 b)) (b))))
c2 = ((1 a) (1 c))
a2 = (a (BRACKET (((1 a)) (a)) (((1 c)) (c))) c)

5.2. PRIMITIVES FOR SYNCHRONIZATION

We now turn to a brief review of the primitive queueing operators for synchronization described in [14]. The primitive operator queue() creates an empty queue initially. The contents of the queue may be modified, by a side-effect, via the operators enq and deq. enq(q,f) synchronizes the execution of a functional expression f by enqueueing a token for f in the queue q; the actual execution of f can be initiated only after the resource dequeues the token for f from the queue q, using deq(q). The value of enq(q,f) is the value computed by f; the value of deq(q) is delay(f), where the token for f is at the head of q. delay(f) is the unevaluated form of f, and the evaluation may be explicitly *forced* by force(d) where d = delay(f). If we wish to evaluate the token immediately after dequeueing, we may use eval(q), which is equivalent to force(deq(q)). Separating the dequeuing of a request from its evaluation facilitates the execution of several tokens from a single queue in parallel.

When multiple queues are used to synchronize several different classes of tokens, it is often necessary to test for the presence of tokens in the different queues. The operator waitq(q,n) tests and waits until q has at least n tokens in it, and only then returns a value, say true, as its result. In contrast to waitq, the operator nonempty(q,n) returns true if q has at least n tokens in it, and false otherwise; thus no waiting is involved.

The last queueing primitive to be used here is reserveq(q,n) which reserves the first n tokens of q and makes them "invisible" during any subsequent testing of q -- either by waitq or nonempty. The motivation for this operator will be clear when the translation of resource expressions is considered.

We summarize all the queueing operators below:

queue()	creates an empty queue.
enq(q,f)	synchronizes the evaluation of f using q by enqueueing a token for f in q.
deq(q)	returns an unevaluated form, delay(f), where the token for f was at the head of q.
force(d)	evaluates f, where d = delay(f).
evalq(q)	dequeues and evaluates f, where the token for f is at the head of q.
waitq(q,n)	tests and waits until q has at least n tokens.
nonempty(q,n)	returns a boolean value indicating whether or not q has at least n tokens.
reserveq(q,n)	reserves the first n tokens of q.

In order to arbitrate among several queues and exercise control over the order in which tokens from different queues are selected and evaluated, we introduce the following operators:

seq(a_1,...,a_n)	evaluates the expressions a_1,...,a_n in sequence; the result returned is a_n.
spar(a_1,...,a_n)	evaluates the expressions a_1,...,a_n in parallel; the result is a_n, but is returned after all a_1,...,a_n have been evaluated.
arbit(a_1,a_2)	evaluates a_1 and a_2 in parallel; the result is false if a_2 is evaluated before a_1, otherwise true.

5.3. TRANSLATION

The basic approach to the translation is to allocate one queue for each distinct token class. Given an intermediate form ((c_1a_1) (c_2a_2) ... (c_na_n)), we test the conditions c_1,...,c_n in parallel and select the one that is detected to be true earliest. This parallel testing and selection is accomplished by means of a chain of arbit's as follows:

```
LET t1 = arbit(c1,t2)
    t2 = arbit(c2,t3)
       ...
    tn-1 = arbit(cn-1,cn)
RESULT
   IF t1 THEN a1 ELSE
   IF t2 THEN a2 ELSE
      ...
   IF tn-1 THEN an-1 ELSE an
```

where c_i and a_i are to be replaced by their translated programs respectively. Note that c_i is of the form ((n_1 op_1) (n_2 op_2) ... (n_k op_k)). Hence if we allocate q_1 to token class op_1, q_2 to op_2, ..., q_n to op_n , we may translate c_i as spar(waitq(q_1,n_1), waitq(q_2,n_2),..., waitq(q_k,n_k)), which tests and waits until the threshold condition c_i becomes true. Note: The abbreviations $t_1, \ldots, t_{n-1}$ are treated as common subexpressions in FGL, and hence are evaluated only once. Also, the order in which the abbreviations are defined is immaterial.

The general form of an action a_i is (x_1 x_2 ... x_j) where each x_k can only be an atomic term or a repetitive expression; however, the last term, x_j, can also be a prefixed sequence. For sake of uniformity we will assume that the result produced by a_i is in the unevaluated form and must be forced explicitly, similar to that for any atomic term. Thus we have the following general form for the translated program for a_i:

```
LET d1 = trans(x1)
    d2 = trans(x2)
       ...
    dj = trans(xj)
RESULT
   seq(d1,...,dj,
       delay(seq(force(d1),...force(dj)))
      )
```

where trans(x) =

If atom(x) then deq(queue_for_x) else
If x_j = (COMMIT ...) then commit(queues_for_x_j) else
If x = (STAR ...) then star(queues_for_x) else
If x = (BRACE ...) then brace(queues_for_x) else
etc.

where commit, star, and brace are procedures for (COMMIT ...), (STAR ...), and (BRACE ...) respectively.

The difference between # and * (and also between {} and []) from the standpoint of their implementation is that the recursion in the former case has no termination condition, whereas for the latter the recursion terminates when none of the threshold conditions of its body is satisfied by the available tokens. Thus the recursion expands, in the former case, only as much as there are tokens in the input to satisfy some threshold condition of the body.

When an expression occurs as the last term in an action, say x_j, the evaluation of x_j must take place after x_{j-1}, but the threshold condition for x_j may be tested concurrently with evaluation of x_1,x_2,...,x_{j-1}. Assuming that the translated program for x_j is represented by commit(queue_for_x_j), we may express the translated program for ($x_1x_2 \cdots x_{j-1}$ (COMMIT ...)) by modifying the LET and RESULT expression above as follows:

```
LET com = commit(queues_for_xj)
RESULT seq(d1,...,dj-1,
           spar(seq(force(d1),...,force(dj-1),
                    force(com)),
                com))
```

The difference between the translation of * and [] is that the evaluation of successive repetitions will be sequential for * and concurrent for []. In both cases, however, the testing of the threshold condition of the body and the construction of the unevaluated form will be similar, i.e. the threshold condition of successive repetitions of a [] will be tested sequentially. Once a threshold condition among the set of alternatives is selected, it is necessary to reserve as many tokens as indicated in the threshold condition. Such reservations ensure that these tokens are not re-used during the testing of the threshold condition inside the body of the repetitive expression. The number of repetitions in both cases will depend on the number of available tokens, but is in general indeterminate. Furthermore, the actual set of tokens used in constructing the unevaluated form is dequeued before any evaluation is initiated.

We illustrate some of the important steps of the translation using the expression {a.b.a/b + a.[a+c].c}. The intermediate form is

```
(BRACE
 (((2 a) (1 b))
      (a b a (COMMIT (((1 b)) (b))) ))
 (((1 a) (1 c))
      (a (BRACKET (((1 a)) (a)) (((1 c)) (c))) c))
)
```

The translated program is

```
PROCEDURE brace(qa,qb,qc)
LET  t1 = arbit(spar(waitq(qa,2), waitq(qb,1)),
           spar(waitq(qa,1), waitq(qc,1)))
    d1 = deq(qa)      d2 = deq(qb)
    d3 = deq(qa)      d4 = deq(qc)
    com = deq(qb)
    brc = brace(qa,qb,qc)
    brck = bracket(qa,qc)
RESULT seq( IF t1 THEN seq(d1,d2,d3,
              delay(spar(seq(force(d1), force(d2),
                         force(d3), force(com)),
                       com)))
                   ELSE seq(reserveq(qa,1),
                         reserveq(qc,1),
                         seq(d1,brck,d4,
                   delay(seq(force(d1),force(brc),
                         force(d4))))),
            brc)
WHERE
   PROCEDURE bracket(qa,qc)
   LET t1 = or(n1,n2)
       d1 = deq(qa)     n1 = nonempty(qa,1)
       d2 = deq(qc)     n2 = nonempty(qc,1)
       brck = bracket(qa,qc)
   RESULT IF t1 THEN
             IF n1 THEN seq(d1,brck,
                   delay(seq(force(d1),
                         force(brc))))
                ELSE seq(d2,brck,
                   delay(seq(force(d2),
                         force(brc))))
                ELSE nil
```

In order to initiate the evaluation of the entire program, it is necessary to force the top-most expression by force(brace(qa,qb,qc)).

6. SUMMARY AND CONCLUSIONS

Resource expressions are proposed here as a high-level linguistic means of specifying resource control. These expressions are composed of primitive constructs for arbitration, iteration, etc., and are capable of specifying solutions to a variety of problems. Given that it may be necessary to coordinate concurrent computations that arise in functional programs and interface functional programs with structures that have a shared state, e.g. databases, we feel an expression-based language like resource expressions is appropriate for this purpose, since they are notationally compatible with the applicative style, and also a simple denotational definition for them can be constructed.

Resource expressions are very similar to path expressions in their basic approach to specification, but differ in their semantics and implementation. We have formalized the semantics of resource expressions in terms of a set of execution graph-residue pairs, by defining this set for any bag of input tokens. The main difference in our semantics is that we take into account the notions of consistent as well as complete behavior of an expression. In order to provide the user the capability of choosing from different shades of the two approaches, the "/" construct has been included.

We have presented a systematic translation of resource expressions in terms of the queueing primitives of a demand-driven execution model. In comparison to implementations of path expressions, our approach does not require restrictions, such as those barring repeated occurrences of an operation name, etc. [1, 6, 7]. The two main steps in our translation are the following: conversion to an intermediate form (condition-action pairs) based on the semantics of the different constructs, and translation of the intermediate form in terms of the queueing primitives. The latter translation is in turn separated into translating conditions and actions, and then combining the two translated program fragments together. Owing to the modularity in the translation process and its close bearing to the defined semantics, we have been able to construct a correctness proof of the translation for an abstract implementation [15].

An interesting application of demand-driven evaluation in this implementation is in representing infinite execution graphs: e.g. we translate the expression $(a+b)^{\#}$ using what appears to be a nonterminating recursion; however, because of demand-driven evaluation, the recursion will expand out as much as is necessary to accomodate available tokens. Other benefits of demand-driven evaluation for resource control are discussed in [14], e.g. in rendering simple solutions to the problem of busy-waiting, etc.

It is possible to perform several optimizations on the translated program, in order to reduce their space and time requirements, by techniques such as: 1) combining deq and force into evalq, 2) minimizing the number of queues, 3) avoiding unnecessary reservations, etc. A fuller discussion of these optimizations and the conditions under which they are applicable are presented in [15].

REFERENCES

[1] S. Andler, Predicate Path Expressions: A High-level Synchronization Mechanism. Ph.D. thesis, Carnegie-Mellon University, (August, 1979).

[2] Arvind, K.P. Gostelow, and W. Plouffe, "Indeterminacy, monitors, and dataflow". Operating Systems Review 11(5), (November, 1977), pp. 159-169.

[3] R.R. Atkinson and C.E. Hewitt, "Specification and proof techniques for Serializers". IEEE Transactions on Software Engineering SE-5(1), (January, 1979), pp. 10-23.

[4] V. Berzins and D. Kapur, Denotational and axiomatic definitions for Path Expressions. Laboratory for Computer Science, M.I.T., Computation Structures Group Memo 153-1, (November, 1977).

[5] T. Bloom, "Evaluating synchronization mechanisms," In Proc. of the Seventh ACM Symposium on Operating Systems Principles, (1979), pp. 24-32.

[6] R.H. Campbell and A.N. Habermann, "The specification of process synchronization by Path Expressions," In Gelenbe and Kaiser (editors), Operating Systems, Springer, (1974), pp. 89-102.

[7] R.H. Campbell and R.B. Kolstad, A practical implementation of Path Expressions. University of Illinois at Urbana-Champaign, Technical Report TR UIUCDCS-R-80-1008, (June, 1980).

[8] P.J. Courtois, R. Heymans, and D.L. Parnas, "Concurrent control with readers and writers," Communications of the ACM 14(10), (October, 1971), pp. 667-668.

[9] E.W. Dijkstra, "Guarded commands, non-determinacy, and a calculus for the derivation of programs," Communications of the ACM 18(8) (August, 1975), pp. 453-457.

[10] D.P. Friedman and D.S. Wise, "An Indeterminate Constructor for Applicative Programming," In Proc. Seventh Annual Symposium on Principles of Programming Languages, (1980), pp. 245-250.

[11] J.R. Gurd and A.J. Catto, "Resource Management in Dataflow," In Conference on functional programming languages and computer architecture, (1981), pp. 77-84.

[12] A.N. Habermann, Path Expressions. Dept. of Computer Science, Carnegie-Mellon University, Technical Report, (July, 1975).

[13] C.A.R. Hoare, "Monitors: An operating system structuring concept," Communications of the ACM 17(10), (October, 1974), pp. 545-557.

[14] B. Jayaraman and R.M. Keller, "Resource control in a demand-driven data-flow model," In Proc. International Conference on Parallel Processing, (1980), pp. 118-127.

[15] B. Jayaraman, Resource control in a demand-driven data-flow model. Univ. of Utah, Ph.D. thesis, (August, 1981).

[16] R.M. Keller, Denotational models for parallel programs with indeterminate operators. North-Holland, E.J. Neuhold (ed.), Formal Description of Programming Concepts. (1978), pp. 337-366.

[17] R.M. Keller and G. Lindstrom. "Applications of feedback in functional programming," In Conference on functional programming languages and computer architecture. (1981), pp. 123-130.

[18] R.M. Keller, G. Lindstrom, and S.S.Patil, "A loosely-coupled applicative multi-processing system," In Proc. AFIPS, (1979), pp. 613-622.

[19] R.M. Keller and G. Lindstrom, "Hierarchical analysis of a distributed evaluator," In Proc. International Conference on Parallel Processing, (1980), pp. 299-310.

[20] T. Kimura. "An algebraic system for process structuring and inter-process communication," In ACM Eighth Annual Symposium on the Theory of Computing, (May, 1976), pp. 92-100.

[21] S.C. Kleene, Representation of events in nerve nets. Princeton University Press, (1956), pp. 3-40.

[22] P.E. Lauer and R.H. Campbell, "Formal semantics of a class of high-level primitives for coordinating concurrent processes," Acta Informatica 5: (1975), pp. 297-332.

[23] W.E. Riddle. "An approach to software system behavior description," Computer Languages 4: (1979), pp. 29-47.

[24] A.C. Shaw. "Software Specification languages Based on Regular Expressions," In W.E. Riddle, R.E. Fairley (editor), Software Development Tools, (May, 1979), pp. 148-174.

PARALLEL IMPLEMENTATION OF FUNCTIONAL LANGUAGES

J.R. Kennaway and M.R. Sleep
University of East Anglia
Norwich, NR4 7TJ, U.K.

Abstract

Functional programming, and its implementation using parallel architectures, is receiving increasing attention in the literature. Turner [4] has proposed a novel implementation for sequential machines using a variable-free form of code based on logical combinators.

We present one translation of combinatory representations to process nets which allows full exploitation of parallelism. Our notation (LNET) is an exchange-view, behaviour passing variant of Milner's CCS.

Introduction

Programming even a sequential machine in a provably correct and maintainable fashion presents a complex and challenging intellectual task. Managing this complexity in the face of parallel architectures presents an immense challenge, and approaches which reduce this complexity are currently receiving widespread attention in the literature.

Functional languages reduce the complexity of the programming task by prohibiting destructive assignment: a functional program may be viewed as a set of mathematical equations which specify the solution. This is good from the software engineering viewpoint, but makes life hard for the implementer who now has to work out when it is safe to forget values, resorting to garbage collection in extremis.

On the other hand, because there are no side-effects in a functional language, expressions may always be evaluated in parallel, which suggests we may remedy at least some of the perceived inefficiency of functional programming by buying speed from parallel technology.

We present here a language, LNET, for describing parallel processes, and show how functional languages can be translated via combinators into LNET.

We rely heavily on the reader's willingness to read [2], [3], and [4]. The recent ACM conference [1] contains much useful background.

Major characteristics of LNET

LNET stands for Language of Named Experiment Trees, reflecting its origins in CCS [3], with which we assume familiarity. There are two major changes with respect to CCS:

1. CCS ports, which present some difficulties of implementation, are eliminated in favour of process names, and the underlying message-passing medium is no longer assumed to be synchronous. Instead, each process has a name, generated at run-time when the process was created, by which it is known to other processes. A communication between two processes takes the form of an exchange of messages, in which one process (the active partner in the exchange) directs a message to the other process (the passive partner), and then waits for that process to accept the message and send back a reply. The passive process may perform some local processing to compute the reply, but no intervening communications are allowed. The result is that a single exchange can be implemented as a pair of asynchronous communications, while being logically equivalent to one indivisible synchronous event. This allows LNET to be given a clean axiomatic semantics.

2. Process behaviours and process names can be sent as messages from one process to another. This extension is necessary to allow the dynamic rearrangement of patterns of communication which our distributed implementation of graph reduction requires.

Basic LNET constructs

Space precludes a full syntax and semantics for LNET. Here we shall informally describe the principal constructs of the language.

An LNET process is of the form X:p. p is a process behaviour, which specifies the communications the process is capable of. X is a process identifier, which at run time will be bound to an automatically generated process name unique to the process. (Note that process names themselves do not appear in the syntax of LNET). The behaviour p may take any of the following forms:

1. NIL - the behaviour which does nothing.

2. par X1:p1 | ... | Xk:pk in p'
This creates k new processes whose behaviours are p1,...,pk and whose run-time names are bound to the process identifiers X1,...,Xk. They run concurrently with the process X:p'.

3. g.p' g is a guard, which is constructed from serial or parallel combinations of communications. In the notation of context-free grammars we have:

g ::= c / g.g / g|g

The communication c takes one of three forms.

(i) e!X?x This is an active communication,

0190-3918/82/0000/0168$00.75 © 1982 IEEE

directed at the process identified by X. The expression e is evaluated and sent to X; the communication then awaits the reply and binds it to x. The remainder of the behaviour in which this communication appears may use the value of x.

(ii) x?!e This is a passive communication. It accepts from any other process a message which it binds to the variable x. It then evaluates the expression e (which may depend on x) and transmits the result back to the process making the active communication. The value of x is, as with the previous form of communication, available to the remainder of the behaviour.

We extend this by also allowing passive communications to take the forms t?!e and t(x)?!e, where t is any of some countable but otherwise unspecified set of tokens. The communication t?!e will wait for some process to make it an active communication of the form t!X?x' (with the same token t). Similarly the communication t(x)?!e will only accept a corresponding active communication of the form t(e')!X?x'.

(iii) wait This is also a passive communication. It suspends the process in which it occurs until some other process makes an active communication to it, and then proceeds without replying. The process making the active communication is still waiting for a reply; a later passive communication of the form (ii) will succeed and provide the reply.

4. ind(X') This is an indirection behaviour. It accepts any active communication made to it and retransmits it to the process identified by X'. X', if and when it accepts the communication, will send its reply directly to the process that made it, rather than via the indirection process.

5. LNET has a fairly conventional apparatus of let and where declarations, and conditionals.

Translating functional programs into LNET

We assume familiarity with Turner's paper [4] in which he shows how functional programming languages can be implemented by translating their programs into a variable-free form, by introducing a few constant functions called combinators. Apart from the usual basic values and operators, just two combinators, called K and S, are sufficient to express any functional program:

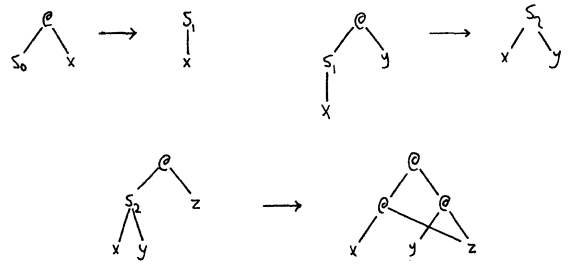
(K x) y = x ((S x) y) z = (x z)(y z)

These definitions of K and S can be read as rewrite rules, allowing any instance of their left hand sides (a redex) to be rewritten as the right hand side. After translation into combinators, a program can be executed by repeatedly applying these rules. A translation into K and S alone is highly inefficient; however, by introducing a few more combinators - six, in fact - a more efficient translation can be obtained.

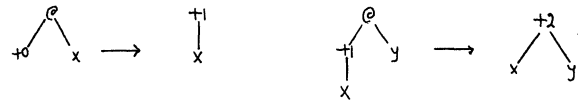
Expressions built up from combinators, basic values, and operators can be represented as trees, or, more generally, as directed graphs, which allow sharing of common subexpressions. We shall now show how these combinator graphs can be translated into LNET process nets. The basic idea is that each node of the graph is modelled by a process, and processes representing adjacent nodes of the graph communicate with each other in such a way that the resulting behaviour of the process net corresponds with the operation of graph reduction. Various regimes of graph reduction (normal order, parallel innermost, etc.) and combinations of these can be modelled by choosing the translation appropriately.

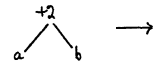
The reduction rule for S, in graphical notation, is as follows. The nodes marked "@" (read "apply") represent function applications.

The circled part of the left-hand graph is the redex reduced by this rule. If the nodes of the graph are processes distributed in some way over an underlying network of processors, a significant amount of non-local computation may be required just to establish the existence of a redex. Our first step is therefore to break down the reduction rules into smaller steps, each of which requires communication only between one process and its immediate neighbours. We subdivide S into three different forms, S0, S1 and S2, with the rules:



We do the same thing for basic operators such as +:





⟶ a node holding the value of a+b
(when a and b are simple integers)

List-handling operators (cons, nil, head, tail) can be handled similarly.

We now define LNET behaviours to model these.

```
S0 = let p = app?!S1 . p in p
S1 = λX. let p = app?!(S2 X) . p in p
S2 = λX.λY. let p = app?!(S3 X Y) . p in p
S3 = λX.λY.λZ. par  V:(APPLY X Z)
                  | W:(APPLY Y Z)
               in  (APPLY V W)
```

Most other combinators may be modelled in the same way. Two, the identity I and the "deleter" K, require the use of indirection behaviours. Their combinator reduction rules are:

$$I\ x \rightarrow x \qquad (K\ x)\ y \rightarrow x$$

Accordingly, we define the "incomplete" behaviours I0, K0 and K1 analogously to S0 and S1, and define I1 and K2 as:

$$I1 = \lambda X.\ \underline{ind}(X) \qquad K2 = \lambda X.\lambda Y.\ \underline{ind}(X)$$

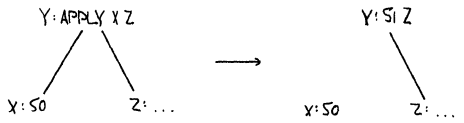
APPLY is the behaviour which models apply nodes:

$$APPLY = \lambda X.\ \lambda Y.\ \underline{app}!X?z\ .\ (z\ Y)$$

This behaviour takes as arguments the process identifiers identifying the processes which model the left and right hand descendants of the apply node in the combinator graph. It sends a token app to the left hand descendant X. The reply it receives (which should be a parametrised behaviour requiring a process identifier) is bound to z. The behaviour then becomes the result of supplying the identifier Y as an argument to z. Comparing this APPLY with the definition of, for example, S0, we see that in the parallel combination

$$X{:}S0 \mid Y{:}(APPLY\ X\ Z) \mid Z{:}\ldots$$

the APPLY process sends the token app to the S0, which replies with the message S1. The APPLY then becomes the behaviour (S1 Z). The transformation of the process net can be pictured thus:



Note that the process X:S0 is still there and is ready to respond to another active communication. This is necessary because of the possibility of sharing. There may be many other APPLY processes sending app tokens to X, and the process X must reply to all of them. When there are no more references to X in the process net the X:S0 process is garbage and may be collected. (Determining when this happens is a non-trivial question and is not addressed here).

Constants such as 17 or true are represented by processes which repeatedly (because of possible sharing) send the constant they hold to any other process which asks for it.

```
CONST = λx. let val?!x . p in p
```

The behaviour (CONST 17) expects a val token, to which it replies with the message "17". This val token will have been sent by a process representing a basic operator such as +:

```
+2 = λX.λY. (val!X?a | val!Y?b) . (CONST(a+b))
```

The + on the right hand side is the "real" one that actually does the addition. There are also a +0 and +1 defined analogously to S0 and S1.

A proof that this translation correctly models combinator graph reduction requires the construction of a formal semantics for LNET and a mathematical statement of the precise correspondence between the reduction of a combinator graph and the behaviour of its LNET translation. It is beyond the scope of this paper. A detailed example of the reduction of a simple graph to normal form is presented in [2].

The translation we have given models the regime of combinator reduction which reduces every redex in the graph concurrently. This maximises parallelism but is dangerous in the case of graphs which, while processing a normal form, also allow nonterminating reduction sequences. However, other reduction methods, such as lazy reduction, can be modelled by choosing other translations of the combinators, operators, and apply node.

Conclusion

The representation of functional programs as variable-free combinator graphs allows them to be modelled as networks of parallel processes which act in concert to perform a distributed evaluation of the whole expression.

Acknowledgements

This work was supported by grants from the U.K. S.E.R.C. Kent Karlsson and Jan Galkowski made detailed comments on an early draft, and Matthew Huntbach and Warren Burton provided a stimulating environment for our work.

References

[1] ACM Conference on Functional Programming Languages and Computer Architecture. New Hampshire (October, 1981).

[2] J.R. Kennaway and M.R. Sleep, "Expressions as Processes", ACM Symposium on LISP and Functional Programming, Pittsburg (August, 1982).

[3] Milner, R., A Calculus of Communicating Systems, Springer-Verlag Lecture Notes in Mathematics, vol. 92, (1980).

[4] Turner, D.A., "A new implementation technique for applicative languages", Software: Practice and Experience, vol. 9, (1979), pp. 31-49.

Parallel Generation Of The Postfix Form*

Eliezer Dekel+ and Sartaj Sahni
University of Minnesota

Abstract

An efficient parallel algorithm to obtain the postfix form of an infix arithmetic expression is developed. The shared memory model of parallel computing is used.

Key Words and Phrases: Arithmetic expressions, postfix, infix, parallel computing, complexity.

1. Introduction

The parallel parsing and evaluation of arithmetic expressions has been the focus of research for many years. [1], [2], [9], [11], and [13] are some of the important papers written on the parallel evaluation of arithmetic expressions. The most significant result here is due to Brent [1]. Brent [1] has shown that arithmetic expressions containing n, $n \geq 1$, operands; operators (+, *, and /); and parentheses can be evaluated in $4\log_2 n+10(n-1)/p$ time when p processors are available. Parallel parsing of arithmetic expressions has been considered by Fisher [5], Krohn [8], Lipkie [12], and Schell [16] (among others). Fisher's work is restricted to vector (or pipelined) computers. While Krohn's work was intended primarily for pipelined computers (specifically for the CDC STAR-100), the ideas contained in [8] can be extended to parallel multiprocessor computers. Krohn, however, does not consider the asymptotic performance that could be obtained from his parallel parsing algorithm. Lipkie [12] and Schell [16] explicitly consider parsing on parallel multiprocessor computers. Lipkie [12] provides some grammar rules for parallel parsing but does not develop a formal algorithm. Schell [16] is a thorough study of parallel techniques for several of the phases normally encountered in compiling (scanning, syntax analysis, parsing, error recovery, etc.). Schell develops a parallel LR parser. The complexity of this parser is, however, quadratic in the input size (under some constraints, he shows that its complexity becomes linear). Schell also discusses the applicability of his techniques to precedence grammars.

In this paper, we develop a parallel algorithm to obtain the postfix form of an arithmetic expression. The reader unfamiliar with the postfix form of an expression is referred to Horowitz and Sahni [6].

The model of parallel computation that we shall use here is commonly referred to as the shared memory model (SMM). Much work has been done on the design of parallel algorithms using the SMM. The reader is referred to [3], [4] and the references contained therein.

While one can talk of obtaining the postfix form for an entire program, we shall limit our disucssion here to simple expressions. These are permitted to contain only operands (constants and simple variables), operators (only the binary operators +, -, *, /, and ↑ are permitted), and parentheses ('(', and ')').

The parallel algorithm that we shall develop here is closely related to the common priority based sequential infix to postfix algorithm. We shall make explicit reference to the version of this algorithm that is presented in [6]. This algorithm utilizes a stack as well as a dual priority system. The *instack priority* (ISP) of an operator or parenthesis is the priority associated with the operator or parenthesis when it is *inside* the stack. The *incoming priority* (ICP) is used when the operator or parenthesis is outside the stack. For the operator and parenthesis set we are limited to,the priority assignment of Figure 1 is adequate.

The algorithm of [6] assumes that the infix expression is in E(1:n) where E(i) is an operator, operand, or parenthesis, $1 \leq i \leq n$ (in practice, E(i) will be a pointer into a symbol table). For example, the expression A+B*C is input as E(1)=A, E(2)=+, E(3)=B, E(4)=*, and E(5)=C. The postfix form is output in P(1:m), $m \leq n$. For our example, we shall have P(1)=A, P(2)=B, P(3)=C, P(4)=*, and P(5)=+. The sequential time complexity of the postfix algorithm of [6] is O(n).

operator/parenthesis	*ISP*	*ICP*
)	-	0
↑,unary+,unary-	3	4
*,/	2	2
binary+, -	1	1
(	0	4
-∞	0	-

Figure 1: Instack and incoming priorities.

In Section 2, we shall see that the algorithm of [6] can be effectively parallelized.

*This research was supported in part by the Office of Naval Research under contract N00014-80-C-0650.

+ Author's present address: Mathematical Sciences Program, University of Texas at Dallas, Richardson, TX75080.

0190-3918/82/0000/0171$00.75 © 1982 IEEE

2. Parallel Generation of the Postfix Form

Let the infix expression be given in E(1:n) as described in Section 1. For every E(i) that is an operator or an operand, we define a value AFTER(i) such that E(i) comes immediately after E(AFTER(i)) in the postfix form of E(1:n). For the first operand in the postfix form, we define the AFTER value to be zero. Note that since parentheses do not appear in the postfix form, an AFTER value for them need not be defined. As an example, consider the expression (A+B)*C. Its postfix form is AB+C*. Since E(1:7) = ('(', A, +, B, ')', *, C), AFTER(1:7) = (-, 0, 4, 2, -, 7, 3).

Our parallel algorithm to obtain the postfix form of E(1:n) will consist of two phases. In the first, the values AFTER(1:n) will be computed. In the second phase, the postfix form will be obtained using these values. In order to determine AFTER(1:n), we need to first compute the *level* L(i) of each token in the expression. Informally, the level of a token gives the depth of nesting of parentheses in which this token is contained. So, if a token is not within any parentheses, its level is 0. More formally, the level, L, is defined by the algorithm of Figure 2.

$$\textit{step 1: } G(i) \leftarrow \begin{cases} 1 & \text{if } E(i)='(' \\ -1 & \text{if } E(i)=')' \\ 0 & \text{otherwise} \end{cases}, \quad 1 \le i \le n$$

$$\textit{step 2: } L(i) \leftarrow \sum_{j=1}^{i} G(j), \quad 1 \le i \le n$$

$$\textit{step 3: } L(i) \leftarrow L(i)+1 \text{ if } E(i)=')', \quad 1 \le i \le n$$

Figure 2 Computation of L.

In Figure 3, we give an example arithmetic expression together with the L() values associated with each token (row 3).

Let us sequence through procedure POSTFIX of [6] as it works on the example expression of Figure 3. The variable i points to the token in E that is currently being examined. When i=1, E(1)='(' and '(' gets put onto the stack. Next, i=2, and E(2)=A is placed into the postfix form. When i=5, the postfix form has P(1:2)=(A,B) and the stack has the form -∞, (, *. During this iteration, * is unstacked (as ISP(*) ≥ ICP(E(5))). We shall say that E(3) gets *unstacked* by E(5). E(5) gets added to the stack and on the next iteration, E(6)=C is placed in the postfix form. When i=18, the stack has the form -∞, +, ↑, (, -, *, ↑, ↑ and P(1:9)=(A,B,*,C,D,E,F,G,H). During this iteration, E(16)=↑, E(14)=↑, E(12)=*, and E(10)=- get unstacked (in that order). I.e., E(16), E(14), E(12), and E(10) get unstacked by E(18). Furthermore, E(10) is the last operator to get unstacked by E(18).

For each i such that E(i) is an operator, we may define U(i) to be the index in E of the operator or parenthesis that causes E(i) to get unstacked. In case E(i) gets unstacked after the entire expression has been seen, then U(i) = n+1. For our example, U(3) = 5, U(10) = U(12) = U(14) = U(16) = 18. Also, for each i such that E(i) is either an operator or a right parenthesis, we may define LU(i) to be the index of the last operator that gets unstacked by E(i). If no operator is unstacked by E(i), then LU(i) is set to 0. For our example, LU(3)=0, LU(5)=3, LU(7)=LU(10)=LU(12) = LU(14)=LU(16)=0, and LU(18)=10.

Continuing with our example, we see that when i=19, P(1:13)=(A,B,*,C,D,E,F,G,H,↑,↑,*,-), and the stack has the form -∞,+,↑. At this time, E(7)=↑ is unstacked and E(19)=* is stacked. So, LU(19)=7 and U(7)=19. Rows 6 and 7 of Figure 4 give the U, and LU values for all the operators and parentheses of our example. Note that U is defined only for operators and LU only for operators and right parentheses.

An examination of procedure POSTFIX [6] and our definition of the level L of a token reveals that if E(i) is an operator, then:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
E	(	A	*	B	+	C	↑	(	D	-	E	*	F	↑	G	↑	H	)	*	I	-	(	(	J	+	K	)	*	L	)	+	M	)
G	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	-1	0	0	0	1	1	0	0	0	-1	0	0	-1	0	0	-1
L	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	1	1	1	2	3	3	3	3	3	2	2	2	1	1	1
ICP	4		2		1		4	4		1		2		4		4		0	2		1	4	4		1		0	2		0	1		0
ISP	0		2		1		3	0		1		2		3		3		▨	2		1	0	0		1		▨	2		▨	1		▨
U			5		21		19			18		18		18		18			21		31				27			30			33		
LU			0		3		0			0		0		0		0		10	7		5				0		25	0		28	21		31
AFTER	▨	0	4	2	19	3	10	▨	6	12	9	14	11	16	13	17	15	▨	20	7	28	▨	▨	5	26	24	▨	29	75	▨	32	21	▨
Position in P	▨	1	3	2	17	4	14	▨	5	13	6	12	7	11	8	10	9	▨	16	15	23	▨	▨	18	20	19	▨	22	21	▨	25	24	▨

Figure 3

U(i) = least j, j > i such that ISP(E(i)) ≥ ICP(E(j)) and L(i)=L(j). If there is no j satisfying this requirement, then U(i)=n+1.

From the definition of U, it follows that if E(i) is an operator or a right parenthesis, then LU(i) is given by:

LU(i) = least j, j < i such that U(j)=i. If there is no j with U(j)=i, then LU(i)=0.

Before proceeding to determine AFTER, it is useful to eliminate extraneous right parentheses. An *extraneous right parenthesis* is formally defined to be one for which the LU value is 0. Extraneous right parentheses together with their matching left parentheses serve no useful function but may be present in E nonetheless. Examples of occurrences of such parentheses are: (A), ((A+B)*C, and (((A+B))) (extraneous right parentheses have been underlined).

The elimination of extraneous right parentheses may be accomplished in the following way. Define C(1:n) as below:

$$C(i) = \begin{cases} 0 & \text{if } E(i) = \text{) and } LU(i) = 0 \\ 1 & \text{otherwise} \end{cases}$$

Let S(i) be the sum $\sum_{j=1}^{i} C(j)$, $1 \le i \le n$. S(i) gives the number of tokens in E(1:i) that are not extraneous right parenthesis. The replacement:

(E(S(i)), U(S(i)), LU(S(i))) ← (E(i), U(i), LU(i))

carried out for all i such that E(i) is not an extraneous right parenthesis results in the elimination of all extraneous right parentheses from E.

As an example, consider the expression:

(((A+B+C))+D)*(((E)))

The extraneous right parentheses are underlined. Following the elimination of these parentheses, the expression E takes the form:

(((A+B+C)+D)*(((E

As we shall see below, following the determination of the levels L(1:n), the left parentheses serve no useful function in our algorithm. Hence, these could be eliminated along with the elimination of the extraneous right parentheses. To accomplish this, we need only define C(1:n) as:

$$C(i) = \begin{cases} 0 & \text{if } E(i) = \text{ (or } (E(i) = \text{) and } LU(i) = 0) \\ 1 & \text{otherwise} \end{cases}$$

and proceed as before.

Once the extraneous right parentheses have been eliminated, AFTER may be computed as described below. In the following discussion of the computation of AFTER, we assume that n has been updated to the value S(n) defined above.

case 1: E(i) is an operand.

In this case, we determine the largest j, j < i such that E(j) is either an operand or LU(j) is defined and greater than 0 (note that as extraneous parenthesis pairs are not permitted, if E(j)=')' then LU(j) > 0). Such a j does not exist iff E(i) is the first operand in the expression. From procedure POSTFIX and our definition of LU, it follows that

$$AFTER(i) = \begin{cases} 0 & \text{if no } j \text{ as above exists} \\ j & \text{if } E(j) \text{ is an operand} \\ LU(j) & \text{otherwise} \end{cases}$$

case 2: E(i) is an operator.

In this case, we see that if there exists a j such that j > i and U(j)=U(i), then AFTER(i) is the smallest j with this property. So, in our example expression, U(10) = U(12) = U(14) = U(16) = 18. Also, in P, E(10) comes immediately after E(12) which comes immediately after E(14). E(14) comes immediately after E(16).

For E(16), however, there is no j, j > 16 and U(j) = U(16). For operators with this property, there are two possibilities: either U(i)-1 is an operand or U(i)-1 is a right parenthesis. If U(i)-1 is an operand, then E(U(i)-1) is the token placed in P just before the unstacking caused by E(i) begins. Hence, AFTER(i) = U(i)-1. If E(U(i)-1) is a right parenthesis, then this right parenthesis would have caused at least one operator to get unstacked (by assumption, extraneous parenthesis pairs are not permitted). Hence, LU(U(i)-1) ≠ 0 and E(LU(U(i)-1)) is the operator that immediately precedes E(i) in P. So, we get:

j ← least j, j > i and U(j)=U(i)

$$AFTER(i) = \begin{cases} U(i)-1 & \text{if } j \text{ is undefined and } E(U(i)-1) \text{ is an operand} \\ LU(U(i)-1) & \text{if } j \text{ is undefined and } E(U(i)-1) = \text{')'} \\ j & \text{if } j \text{ is defined} \end{cases}$$

Row 8 of Figure 3 gives the AFTER values for all the operators and operands in our example expression. The AFTER values link the E(i)s in the order they should appear in the postfix form. This linked list is shown explicitly in Figure 4. From this linked list, we wish to determine the position, POS, of each operator and

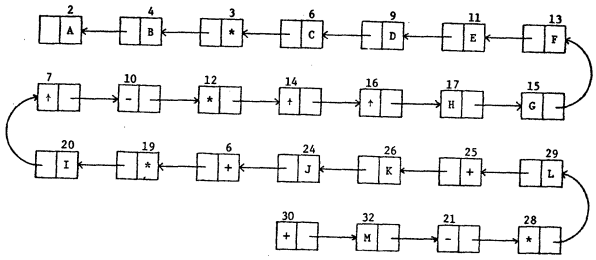


Figure 4

operand in the postfix form. For one of the operands, i.e., the one with AFTER(i)=0, this position is already known (it goes into P(1)). With each E(i), let us associate a one bit field K(i). K(i)=0 iff the position of E(i) in P(i) has not been determined. Initially, K(i)=0 for all but one of the tokens (i.e. the one with AFTER(i)=0).

For any node, i, in the linked list defined by the AFTER values, POS(i), is one more than the number of nodes preceding it in that list (the node with AFTER value 0 is the first node in the list; so the list is linked backwards). The POS values may be obtained by recursively splitting this linked list. The first time the list is split, we get two lists (A and B) consisting of alternating elements from the original list. The POS value of the first element in list A is already known and that for the first element of list B is now known to be 2. Figure 5(a) shows the resulting lists when we start with the lists of Figure 4. The lists A and B are again split. When the list A of Figure 5(a) is split, we get the lists A1 and A2 of Figure 5(b). At this time, the POS value for the first node of list A2 becomes known, i.e., 3. Each time a list is split, we get two lists of about half the length. So, following $\lceil \log n \rceil$ splits, all lists will be of size 1 and all the POS values will be known. The formal algorithm to determine POS is given in Figure 6.

```
step 1 //initialize//
     case
          :AFTER(i) is undefined: K(i) ← undefined
          :AFTER(i)=0 : K(i) ← 1; POS(i) ← 1
          :else: K(i) ← 0
     end case

step 2 //split lists and compute POS//
     for v ← 1 to ⌈log n⌉ do
       if K(i)=0 then j ← AFTER(i)
                      AFTER(i) ← AFTER(j)
                      if K(j)=1 then
                                K(i) ← 1
                                POS(i) ← POS(j)+2^(v-1)
                      endif
          endif
       endfor
```

Figure 6 Algorithm to compute POS.

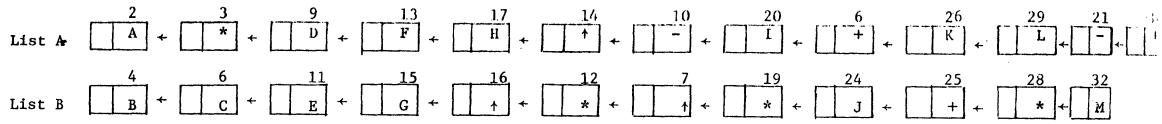
The correctness of the algorithm of Figure 6 can be established formally by providing a proof by induction on the length of the initial linked list. We omit this proof here.

Once the POS values have been computed as described above, the postfix form P is obtained by executing the following instruction:

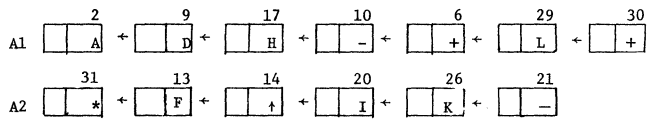
if AFTER(i) is defined *then* P(POS(i)) ← E(i)

Complexity Analyisis

First, let us consider the computation of the levels L (Figure 2). Step 1 can be done in O(1) time using n PEs (each PE is assigned to compute a different G(i)). It can also be done in O(log n) time using n/log n PEs (each PE sequentially computes log n of the G()s). The L(i)s of step 2 may be computed in O(log n) time using



(a) Splitting the list of Figure 4



(b) Splitting the list A of (a)

Figure 5

n/log n PEs and the partial sums algorithm of [4]. Finally, step 3 can be preformed in O(log n) time using n/log n PEs. Hence, the levels L() may be obtained in O(log n) time using n/log n PEs.

Next, consider the computation of U and LU. One possibility is to use mp PEs to first make m copies of each of the p operators and right parentheses in E (m is the number of operators in E). This takes O(log m) time. Note that O(logn) time is needed to avoid read conflicts. Each operator now has a copy of the operators and right parentheses in E for itself. Each operator E(i) is assigned p PEs to work with. These are first used to eliminate operators and right parentheses E(j) with $j \leq i$. Next, the level and ISP of E(i) is transmitted to the remaining operators and right parenthesis. This takes O(log p) time (again having no read conflicts) with p PEs. Operators and right parentheses with a different level number or with ICP > ISP (E(i)) are eliminated. The operators and right parentheses not yet eliminated are candidates for U(i). The one with least j can be determined in O(log p) time using a binary tree comparison scheme and p PEs. If there are no candidates, U(i)=n+1. LU may now be determined in a similar manner. This strategy to compute U and LU takes $O(n^2)$ PEs and O(log n) time. Using the techniques of [4], it can be made to run in O(log n) time using only $O(n^2/\log n)$ PEs.

An alternative stategy is to first collect together all operators and right parentheses that have the same level number. This can be done in $O(\log^2 n)$ time using n PEs as follows. First, each left parentheses determines the position of its matching right parentheses. This is done by simply sorting the left and right parentheses by their level number. If a stable sort is used, each left parenthesis will be adjacent to its matching right parentheses following the sort (Figure 7). The sort can be accomplished in $O(\log^2 n)$ time using n PEs [15]. Now, each left parenthesis can determine the address, M(i), of its matching right parenthesis.

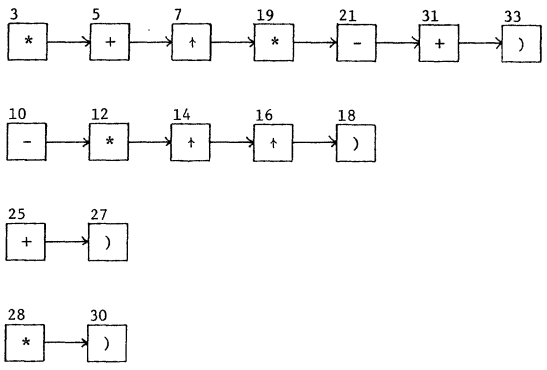


Figure 7

Once M(i) has been determined for each left parenthesis E(i), we can link together all operators and right parentheses with the same level as needed in the computation of U. There are only two possibilities for any operator i. These are:

(a) E(i+1)='(': In this case, E(i) is linked to M(i+1)+1.
(b) E(i+1) ≠ '(': In this case i+2=n+1 or E(i+2) is an operator. Regardless, E(i) is linked to i+2.

Performing this linkage operation on the example of Figure 3 gives the linked lists of Figure 8. Now, each linked list can be treated independently. For operators with the highest ISP (i.e., ↑), the U value is obtained by collapsing together consecutive chains of ↑ so that all ↑ point to the nearest non ↑. The U value equals the link value. So, U(7) = 19, U(14) = U(16) = 18. For operators with the next highest ISP, the U values are obtained by removing all nodes representing the operator ↑. The link values give the U value. Doing this on the lists of Figure 7, yields the lists of Figure 9. So, U(3)=5, U(19)=21, U(12)=18, U(28)=30. Now, by eliminating all nodes that represent * and / and collapsing the lists we can determine the U value for the next ISP class. We obtain U(5)=21, U(21)=32, U(10)=18, and U(25)=27. Each elimination and collapsing operation above can be performed in O(log n) time using n PEs and the strategy used in Figure 6 to compute POS. Since the number of ISP classes is a constant, the time needed to determine U is O(log n).

It should be evident that LU can be computed during the computation of U. Each operator and right parentheses keeps track of the farthest operator it unstacks from each ISP class. In comparing the two strategies to obtain U and LU, we note that the first strategy takes O(logn) time but requires $O(n^2/\log n)$ PEs while the second strategy takes $O(\log^2 n)$ time and requires only n PEs. So, the logn speed-up of the first strategy over the second is obtained through a considerble increase in the number of processors used.

The extraneous right parentheses can be eliminated in O(logn) time using n/logn PEs. The initial values of AFTER() may now be computed. First, each operand determines the nearest (on its left) binary operator, right parenthesis, and operand. These are shown in Figure 9 for our example of Figure 3. Zeroes indicate the absence of a nearest quantity on the left. These three sets of nearest values can be determined in O(log n) time using n PEs. For example, to get the nearest operands, we eliminate all E(i)s that are not an operand. The remaining E(i)s are concentrated to the left. This enables each operand to determine its nearest left operand. Next, the operands are distributed back to their original spots (see [14] for an O(log n) distribution algorithm).

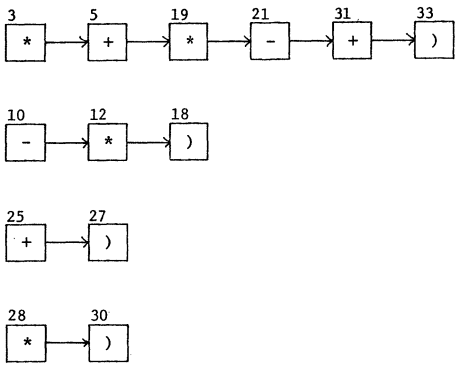


Figure 8

If E(i) is an operand and has no nearest operand on the left, AFTER(i)=0. If the nearest binary operator (on the left) has LU() > 0, then AFTER(i) equals this LU value. If E(i) has a nearest right parenthesis (on the left) then AFTER(i) is the LU value of this parenthesis. Otherwise, AFTER(i) is the location of the nearest operand on the left.

If E(i) is an operator, we can determine the smallest j, j > i such that U(j)=U(i) during the computation of U and LU. So, if such a j exists, AFTER has already been computed. If no such j exists, AFTER(i) is to be set to either U(i)-1 or LU(U(i)-1). Both these quantities are already known. So, the computation of AFTER for operators takes O(1) additional time.

The computation of POS (Figure 6) requires only O(log n) time and n PEs. The formation of P takes O(1) time and n PEs. Hence, using n PEs, the postfix form may be computed in $O(\log^2 n)$ time (the second strategy to compute U and LU must be used as only n PEs are available). The complexity is dominated by the sort step. Another complexity measure worth computing is the EPU (effectiveness of processor utilization). This is the the ratio of the complexity of the fastest known sequential algorithm and the product of the complexity of the parallel algorithm and the number of processors used by this algorithm. For our parallel postfix algorithm, we have:

$$EPU = \Omega\left(\frac{n}{\log^2 n * n}\right)$$

$$= \Omega\left(\frac{1}{\log^2 n}\right)$$

Note also that by using n^2 PEs and the first strategy to compute U and LU, the postfix form can be computed in O(logn) time. The EPU of the resulting algorithm is $\Omega\left(\frac{n}{\log n}\right)$.

3. Conclusions

We have shown that it is possible to effectively parallelize the postfix algorithm given in [6]. Our parallel algorithm runs in $O(\log^2 n)$ time when n PEs are available. If only n/k PEs are available, our algorithm can still be used. The complexity will be $O(k \log^2 n)$.

The results of this paper nicely complement the work reported on the parallel evaluation of expressions (see [1], [2], [9], [11], and [13]).

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
E	(	A	*	B	+	C	↑	(	D	-	E	*	F	↑	G	↑	H	)	*	I	-	(	(	J	+	K	)	*	L	)	+	M	)
nearest binary operator		0		3		5			7		10		12		14		16			19				21		25			28			31	
nearest right parenthesis		0		0		0			0		0		0		0		0			18				18		18			27			30	
nearest operand		0		2		4			6		9		11		13		15			17				20		24			26			29	

Figure 9

References

1. Brent, R., "The parallel evaluation of general arithmetic expressions,". *J. ACM* 21, 2, April 1974 pp. 201-206.
2. Brent, R., Kuck, D.J., and Maruyama, K.M., "The parallel evaluation of arithmetic expressions without divisions," *IEEE Trans. Comput.* C-22, May 1973, pp. 532-534.
3. Dekel, E., and Sahni, S., "Parallel scheduling algorithms," University of Minnesota TR 81-1, to appear in *Operations Research.*
4. Dekel, E., and Sahni, S., "Binary trees and parallel scheduling algorithms," in Lecture Notes in Computer Science, vol 111, CONPAR 81, Springer Verlag, New York, 1981.
5. Fischer, C.N., "On parsing and compiling arithmetic expressions on vector computers." *TOPLAS* Vol. 2, No. 2, April 1980, pp. 203-224.
6. Horowitz, E. and Sahni, S., "Fundamentals of data structures," Computer Science Press, Patomac, MD, 1976.
7. Knuth, D.E., "An empirical study of FORTRAN programs," *Software* 1, April 1971, pp. 105-133.
8. Krohn, H.E., "A parallel approach to code generation for Fortran like compilers,"*SIGPLAN Notices*, March 1975, pp. 146-152.
9. Kuck, D.J., "Evaluating arithmetic expressions of n atoms and k divisions in $\alpha(\log_2 n+2 \log_2 k)+$ c steps, manuscript, March 1973.
10. Kuck, D.J., "Parallelism in ordinary programs," Proc. Symposium on Complexity of Sequential and Parallel Numerical Algorithms, Carnegie-Mellon, Pittsburgh, PA, May 1973. Academic Press, New York.
11. Kuck, D.J., and Maruyama, K.M., "The parallel evaluation of arithmetic expressions of special forms," Rep. RC4276, IBM Res. Center, Yorktown Heights, NY, March 1973.
12. Lipkie, D.E., "A compiler design for multiple independent processor computers," PhD dissertation, University of Washington, Seattle, 1979.
13. Maruyama, K.M., "On the parallel evaluation of polynomials," *IEEE Trans. Comput.*,C-22, Jan. 1973, pp. 2-5.
14. Nassimi, D. and Sahni, S., "Data broadcasting in SIMD computers," *IEEE TRANS. on Computers.* C-30, no. 2., Feb 1981, pp 101-107.
15. Preparata, F.P., "New parallel-sorting schemes," *IEEE Trans. on Computers*, C-27, No. 7,July 1978, pp. 669-673.
16. Schell, R.M., Jr., "Methods for constructing parallel compilers for use in a multiprocessor environment," PhD dissertation, University of Illinois, Urbana, 1979.

A Parallel Matching Algorithm for Convex Bipartite Graphs*

Eliezer Dekel+ and Sartaj Sahni
University of Minnesota

Abstract

An efficient parallel algorithm to obtain maximum matchings in convex bipartite graphs is obtained.

Key Words and Phrases: Parallel algorithm, convex bipartite graph, scheduling , complexity.

1. Introduction

A convex bipartite graph G is a triple (A,B,E). A = $\{a_1,a_2,...,a_n\}$ and B = $\{b_1,b_2,...b_m\}$ are disjoint sets of vertices. E is the edge set. E satisfies the following properties:

(1) If (i,j) is an edge of E, then either i∈A and j∈B or i ∈B and j∈A; i.e., no edge joins two vertices in A or two in B.

(2) If $(a_i,b_j)\in E$ and $(a_i,b_{j+k})\in E$, then $(a_i,b_{j+q})\in E$, $1\le q<k$.

Property (1) above is the bipartite property while property (2) is the convexity property. An example convex bipartite graph is shown in Figure 1.1.

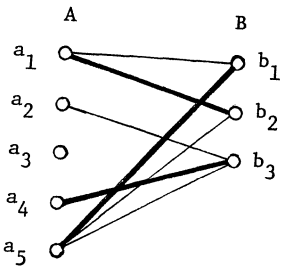


Figure 1.1 A convex bipartite graph.

F ⊆ E is a *matching* in the convex bipartite graph G=(A,B,E) iff no two edges in F have a common endpoint. F1={(a_1, b_2), (a_4,b_3), (a_5,b_1)}is a matching in the graph of Figure 1.1 while F2={$(a_1,b_1),(a_1,b_2),(a_2,b_3)$} is not. F is a *maximum cardinality matching* (or simply a maximum matching) in G iff (a) F is a matching and (b) G contains no matching H such that |H|>|F| (|H|=number of edges in H). The matching depicted by solid lines in Figure 1.1 is a maximum matching in that graph.

In what follows, we shall find it convenient to have an alternate representation of convex bipartite graphs. It is clear that every convex bipartite graph G=(A,B,E), A={$a_1,...,a_n$}, B={$b_1,...,b_m$} is uniquely represented by the set of triples:

$$T = \{(i,s_i,h_i) \mid 1\le i\le n\}$$
$$s_i = \min\{j \mid (a_i,b_j)\in E\}$$
$$h_i = \max\{j \mid (a_i,b_j)\in E\}$$

In the *triple representation*, we explicitly record the smallest (s_i) and highest (h_i) index vertices to which each a_i is connected. For the example of Figure 1.1, we have T = {(1,1,2), (2,3,3), (3,0,0), (4,3,3), (5,1,3)}.

As an example of the use of matchings in convex bipartite graphs, consider the problem of scheduling n unit time jobs to minimize the number of tardy jobs. In this problem, we are given a set, of n jobs. Job i has a release time r_i and a due time d_i. It requires one unit of processing. We assume that r_i and d_i are natural numbers. A subset F of J is a *feasible subset* iff every job in F can be scheduled on one machine in such a way that no job is scheduled before its release time or after its due time. A feasible subset F is a *maximum feasible subset* iff there is no feasible subset MAXM of J such that |Q|>|J|.

A maximum feasible subset F can be found by transforming the problem into a maximum matching problem on a convex bipartite graph. Without loss of generality, we may assume that min{r_i}=0; $r_i<d_i$, $1\le i\le n$; and max {d_i}≤n. The convex bipartite graph corresponding to J is given by the triple set T = {(i, s_i, h_i) | $s_i=r_i, h_i=d_i-1$}. Figure 1.2 shows an example job set and the corresponding convex bipartite graph G. Vertex i of A represents job i while vertex i in B simply represents the time slot [i,i+1]. There is an edge from job i to time slot [j,j+1] iff $r_i\le j<d_i$. Hence, every matching in G represents a feasible subset of J. Also, corresponding to every feasible subset of J there is a matching in G. Clearly, a maximum cardinality feasible subset of J can be easily obtained from a maximum matching of G. In addition, a maximum matching also provides the time slots in which the jobs should be scheduled.

Glover [5] has obtained a rather simple algorithm to find a maximum matching in a convex bipartite graph G=(A,B,E). Let $h_i=\max\{j \mid (a_i,b_j)\in E\}$, $1\le i\le|A|$. Glover's algorithm considers the vertices in B one by one starting at b_1. We first determine the set R of remaining vertices in A to which the vertex b_j currently being considered is connected. Let q be such that $a_q\in R$ and $h_q = \min_{a_p\in R}\{h_p\}$.

*This research was supported in part by the Office of Naval Research under contract N00014-80-C-0650.

+Current address: Mathematical Sciences Program, University of Texas at Dallas, Richardson, Texas 75080.

0190-3918/82/0000/0178$00.75 © 1982 IEEE

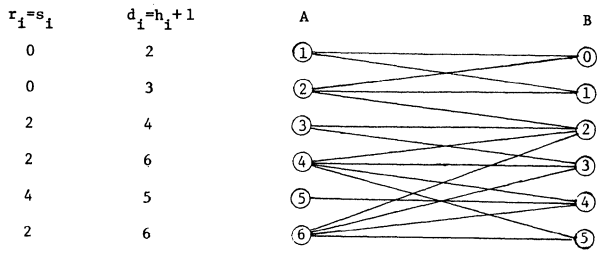


Figure 1.2

Vertex b_j is matched to a_q and a_q deleted from the graph. The next vertex in B is now considered. Observe that Glover's algorithm is essentially the same as that suggested by Jackson [6].

A straightforward implementation of Glover,s algorithm has complexity O(mn). When m is Ω(log log n), a more efficient implementation results from the use of the fast priority queues of van Emde Boas ([4] and [8]). The resulting implementation has complexity O(m+nloglogn). The fastest sequential algorithm known for the matching problem is due to Lipski and Preparata [8]. It differs from Glover's algorithm in that it examines the vertices of A one by one rather than those of B. This algorithm has complexity O(n+mA(m)) where A(.) is the inverse of the Ackermann's function and is a very slowly growing function.

In Section 2, we obtain a parallel algorithm for maximum matchings in convex bipartite graphs. Our analysis of this algorithm will assume the availability of as many PEs as needed. This is in keeping with much of the research done on parallel algorithms. In practice, of course, only k processors (for some fixed k) will be available. Our analyses are easily modified for this case. It will be apparent that if our algorithm has time complexity t(n) using g(n) PEs, then with k PEs (k<g(n)), its complexity will be t(n)g(n)/k.

The parallel computer model used is the shared memory model (SMM). This is an example of a single instruction stream multiple data stream (SIMD) computer. In a SMM computer, there are p processing elements (PEs). Each PE is capable of performing the standard arithmetic and logical operations. The PEs are indexed 0,1,...,p-1 and an individual PE may be referenced as in PE(i). Each PE knows its index and has some local memory. In addition, there is a global memory to which every PE has access. The PEs are synchronized and operate under the control of a single instruction stream. An enable/disable mask may be used to select a subset of the PEs that are to perform an instruction. Only the enabled PEs will perform the instruction. Disabled PEs remain idle. All enabled PEs execute the same instruction. The set of enabled PEs may change from instruction to instruction.

If two PEs attempt to simultaneously read the same word of the shared memory, a ***read conflict*** occurs. If two PEs attempt to simultaneously write into the same word of the shared memory, a ***write conflict*** occurs. Throughout this paper, we shall assume that read and write conflicts are prohibited.

The reader is referred to [2] for a list of references dealing with graph algorithms, matrix algorithms, sorting, scheduling, etc. on a SMM computer.

2. Parallel Matching In Convex Bipartite Graphs

In Section 1, we showed that every instance of the problem of scheduling jobs to minimize the number of tardy jobs could be transformed into an equivalent instance of the maximum matching in a convex bipartite graph problem. It should be evident that the reverse is also true. Hence, the two problems are isomorphic. A parallel algorithm for a special case of the job scheduling formulation was obtained by us in [1]. In this special case, it was assumed that all jobs have the same release time. This corresponds to the case when the convex bipartite graphs are of the form $T=\{(i,s_i,h_i)|1\le i\le n\}$ and $s_i=c, 1\le i\le n$ for some c.

We shall now proceed to show how the solution for the special case described above can be used to solve the general case when all the r_is are not the same. This will be done using the binary tree method described by Dekel and Sahni [2]. Rather than specify the new algorithm formally, we shall describe how it works by means of an example.

A convex bipartite graph is shown in Figure 2.1. For this graph, $|A|=14$ and $|B|=13$. The s_i and h_i values associated with each vertex of A are given in the first two columns of this figure. The first step in our parallel algorithm for maximum matching is to sort the vertices in A in nondecreasing order of s_i. Vertices with the same s_i are sorted into nondecreasing order of h_i. For our example, the result of this reordering is shown in Figure 2.2.

Following the sort, we identify the distinct s_i values. Let these be $R_1,R_2,...,R_k$. Assume that $R_1<R_2<...<R_k$. Let $R_{k+1}=\max\{h_i\}+1$. For our example, k=4 and R(1:k+1) = (1, 4, 9, 12, 14).

We are now ready to use the binary tree method of [2]. The underlying computation tree we shall use is the unique complete binary tree with k leaf nodes. Figure 2.3 shows the complete binary trees with 4, 5, and 6 leaf nodes. For our example, k=4 and we use the tree of Figure 2.3(a). With each node, P, in the computation tree, we associate a contiguous subset {u,u+1,u+2,...,v} of the vertices in B. This subset is denoted [u,v].P or simply [u,v].

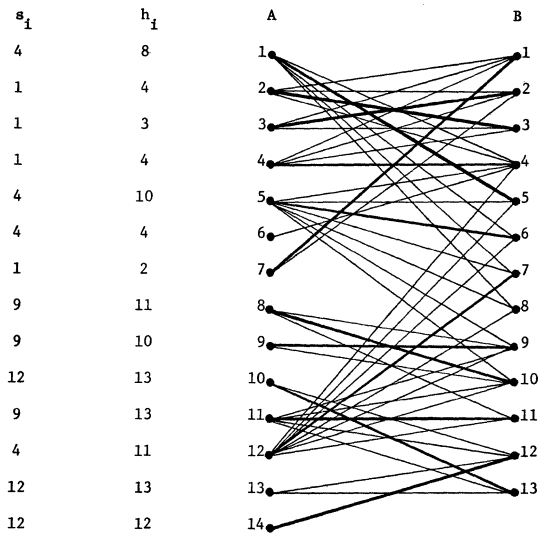


Figure 2.1

i	7	3	2	4	6	1	5	12	9	8	11	14	10	13
s_i	1	1	1	1	4	4	4	4	9	9	9	12	12	12
h_i	2	3	4	4	4	8	10	11	10	11	13	12	13	13

Figure 2.2

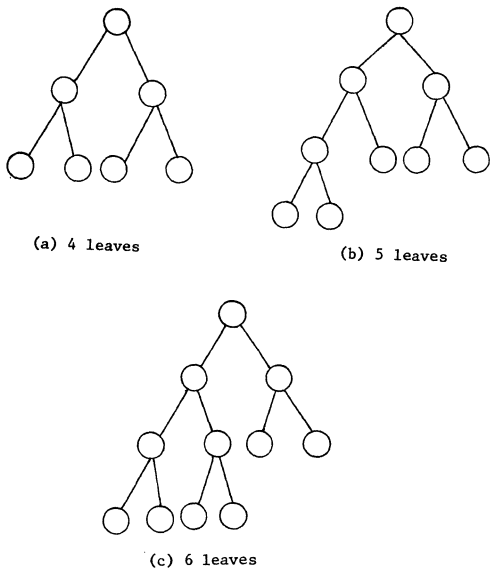


Figure 2.3: Complete binary trees.

Let the leaf nodes of the computation tree be numbered 1 through k, left to right. If P is the ith leaf node, then [u,v].P is $[R_i,R_{i+1}-1]$ (i.e., $u=R_i$ and $v=R_{i+1}-1$). If P is not a leaf node, then the subset of B associated with P is [u,v].LC(P) U [u,v].RC(P) where LC(P) and RC(P) are, respectively, the left and right children of P. The subsets of B associated with each of the vertices in the computation tree for our example are shown in Figure 2.4. The number in each node of this tree is its index.

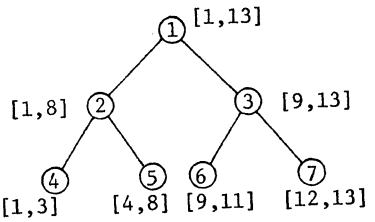


Figure 2.4

Let P be any vertex of the computation tree. Let [u,v] be the subset of B associated with P. The subset of A *available for matching* at node P is denoted M(P) and is defined to be:

$$M(P)=\{i \mid u \le s_i \le v\}$$

For,example,

M(1)={1,2,...,14};
M(2)={1,2,3,4,5,6,7,12};
M(4)={2,3,4,7};
etc.

The subset M(P) of A vertices available for matching at P may be partitioned into three subsets MAXM(P), I(P), and T(P). MAXM(P) is a maximum cardinality subset of M(P) that may be matched with vertices in [u,v].P by algorithm MATCH; this subset is called the *matched set*. I(P) denotes the *infeasible set*. It consists of all vertices $i \in M(P)$-MAXM(P) such that $h_i \le v$. The *transferred set* T(P) consists of all vertices $i \in M(P)$-MAXM(P) such that $h_i > v$.

Consider node 2 of Figure 2.4. The matching problem defined at this node is given in Figure 2.5. Note that $h'_i = \min\{v, h_i\}$. A' is the set M(2) and B' is [u,v].2. If Glover's algorithm is used on this graph, then {1,2,3,4,5,7,12} defines a subset of A' that can be matched with vertices in B'. Further, this gives a maximum matching. Hence, MAXM(2)={1,2,3,4,5,7,12}; I(2)=6; and T(2)=ϕ. Observe that |MAXM(1)| is the size of a maximum matching in the original convex bipartite graph. Also, T(1)=ϕ and I(1)=A-MAXM(1).

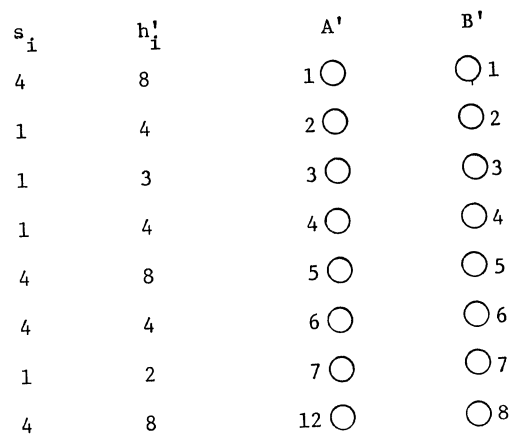


Figure 2.5

We shall make two passes over the computation tree. The first pass begins at the leaves and moves towards the root. During this pass, the MAXM, I, and T sets for each node are computed. The second pass starts at the root and progresses towards the leaves. In this pass, the MAXM set for each node is updated so as to correspond to the set of A vertices matched by Glover's algorithm to the B vertices associated with that node.

Pass 1

In this pass, we make extensive use of the parallel algorithm developed in [1] for the case when all the s_is are the same. For our purposes here, it is sufficient to know the sequential algorithm (FEAS of [1]) that this parallel algorithm is based on. This sequential algorithm is given in Figure 2.6. For convenience, this has been translated into the graph language used here. The parallel version of this algorithm has complexity O(logn) and uses n/logn PEs [1].

```
line procedure FEAS(n,u,v)
          //Find a maximum matching of vertices in
          A onto the B set [u,v]. For every vertex i∈A,
          s_i=u.//
 1  global MAT(1:n); set A; integer n,u,v,i,j
 2  sort A into nondecreasing order of h_i
 3  MAT(1:n) ← 0//initialize//
 4  j ← u
 5  for i ← 1 to n do
 6    case
 7    :j>v: return(j) //all vertices in B matched//
 8    :j≤h_i: //select i// j ← j+1, MAT(i) ← 1
 9    end case
10  end for
11  return(j)
12  end FEAS
```

Figure 2.6

An examination of Glover's algorithm reveals that it performs exactly as does procedure FEAS when the restrictions and simplifications applicable to FEAS are incorporated into it.

Hence, for a leaf node of the computation tree, the MAXM set may be obtained by a direct application of procedure FEAS (or its parallel equivalent). For example, for node 4 of Figure 2.4, we have A = M(4) = {2,3,4,7}; h_2=4; h_3=3; h_4=4; h_7=2; u=1; and v=3. Using FEAS (observe that this algorithm yields the same results regardless of whether the h_i values or the modified values h'_i of Figure 2.5 are used), we obtain MAXM(4)={7,3,2}. Note that MAXM consists of exactly those vertices i with MAT(i)=1. I() consists of exactly those vertices i with MAT(i)=ϕ and $h_i \le$v. The remaining vertices form T(). The matched set MAXM, transferred set T, and infeasible set I for each of the leaves in our example are shown in Figure 2.7. Null sets are not explicitly shown. So, for node 4, I(4)=ϕ; T(4)={4}; and MAXM(4)={7,3,2}. The sets are ordered by h_i.

For a nonleaf node P, the MAXM, I, and T sets may be obtained by using the MAXM, I, and T sets of the children of P. Let L and R, respectively, be the left and right children of P. To determine MAXM(P), we first use procedure FEAS with u=u_R and v=v_R ($[u_R,v_R]$ is the subset of B associated with the right child R of P). The A set consists of T(L) U MAXM(R). Since both T(L) and MAXM(R) are already sorted by h_i, the sort of line 2 of FEAS can be replaced by a merge. Let S be the subset of T(L) U MAXM(R) that has MAT()=1 following the execution of FEAS. The following theorem establishes that MAXM(L) U S is a maximum cardinality subset of M(P) that may be matched with vertices in [u,v].P. Hence, MAXM(P)=MAXM(L) U S. Following the determination of S, MAXM(L) and S are merged to obtain MAXM(P) in nondecreasing order of h_i.

Theorem 2.1: MAXM(L) U S as defined above is a maximum cardinality subset of M(P) that may be matched with vertices in [u,v].P using algorithm MATCH.

Proof: The proof is by induction on the height of the subtree of which P is the root (A tree consisting of only a root has height 0). If this height is 1, then MAXM(L) and MAXM(R) are maximum cardinality subsets of M(L) and M(R) that can, respectively, be matched by Glover's algorithm with vertices in [u,v].L and [u,v].R. If this distance is greater than 1, then MAXM(L) and MAXM(R) satisfy this maximum cardinality matching requirement by induction.

As far as node P is itself concerned, we see that only vertices in M(L) are candidates for matching with vertices in $[u_L,v_L]$ (recall that for vertices in M(R), the s_i value exceeds v_L). Furthermore, when Glover's algorithm is used with the A set being M(P) and the B set being $[u_L,v_R]$ = [u,v].P, vertices in B are considered in the order u_L, u_L+1,...,v_L, u_R,...,v_R. Hence, MAXM(L) is precisely the subset of M(P) that gets matched with

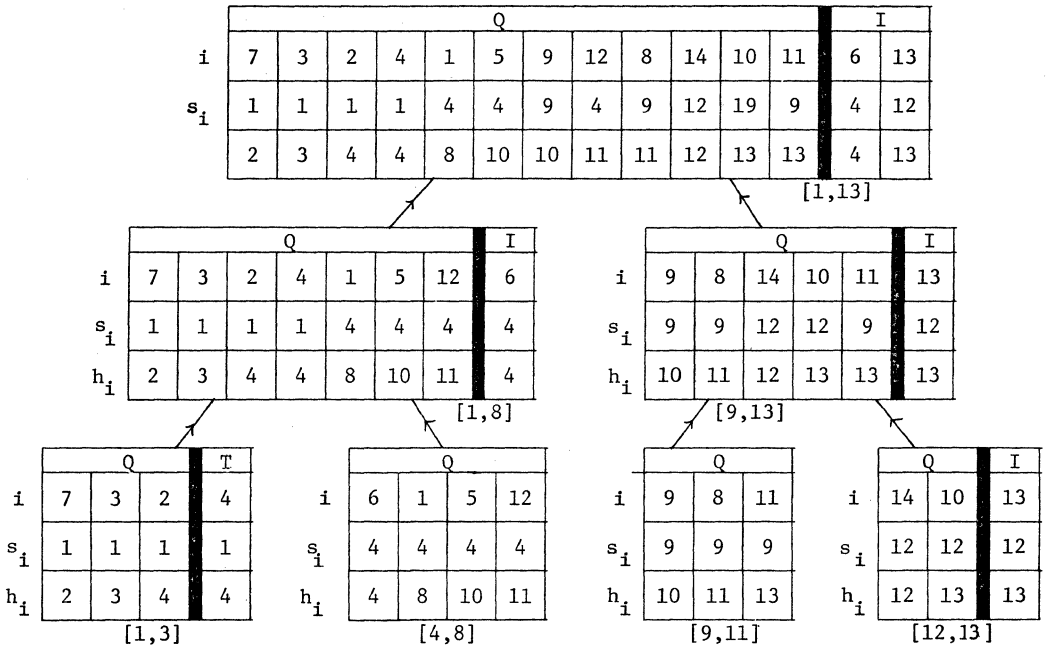


Figure 2.7. Results of first pass.

vertices in $[u_L,v_L]$.

The candidates for the remaining vertices in B, i.e., $[u_R,v_R]$ are clearly T(L) U M(R). From the way Glover's algorithm works, it is also clear that the vertices of T(L) U M(R) that will get matched to $[u_R,v_R]$ are a subset of T(L) U MAXM(R). Let this subset be S'. We wish to show that S is a legitimate choice for S'. First, we show that S represents a feasible matching. Then, we shall show that S is in fact selectable by Glover's algorithm.

We know that MAXM(R) can be matched into $[u_R,v_R]$. Let Z be any such matching. Since S is selected by FEAS, we know that every vertex in S can be paired with a distinct vertex in $[u_R,v_R]$ in a such a way that no vertex j in S is paired with a vertex with index greater than h_j. Consider a pairing W that meets this condition. Now suppose that some vertex j in S is paired with a vertex q in $[u_R,v_R]$ with index less than s_j. Clearly, j must be a member of MAXM(R) (as all vertices in T(L) have an s value less than u_R). Suppose that j is matched to j_1 in Z. So, $q<j_1$. If j_1, is free in W, then the pairing of j in W may be changed from q to j_1. If j_1 is not free, then suppose it is matched to j_2. From the restriction on W, it follows that $q<j_1\le h_{j_2}$. If $j_2\in$T(L), then j_2 may be paired with q and j with j_1 (since $q<j_1$, $s_{j_2}<q<h_{j_2}$). If $j_2\in$MAXM(R), then suppose that j_2 is matched to j_3 in Z. It is easy to see that $j_3\neq j_1$. If $q=j_3$ or j_3 is free in W, then we may pair j with j_1 and j_2 with j_3. If q is in the interval $[s_{j_2}, h_{j_2}]$, then we may pair j with j_1 and j_2 with q. If q is not in this interval, then since $q<h_{j_2}$, $q<s_{j_2}\le j_3$. Note that the condition $q<j_{2i+1}$ is preserved. This is needed in case $j_{2i+2}\in$T(L). Now, let j_4 be the vertex paired with j_3 in W. It should be clear that we can continue in this way and modify W so that j is paired with j_1, j_2 with j_3, j_4 with j_5, etc. In the new pairing, there is one fewer vertex of S that is paired with a vertex with smaller s value.

Repeating the above construction several times, W can be transformed into a matching such that every vertex $j\in$S is matched to a vertex q in $[u_R, u_L]$ such that $s_j\le q\le h_j$. Hence, S represents a feasible matching.

Let S' (as defined earlier) be the subset of MAXM(R) U T(L) matched by Glover's algorithm to the vertex set $[u_R,v_R]$. We shall now proceed to show that S is a valid choice for S'. Let Z be any matching of MAXM(R) into $[u_R,v_R]$. Let Y be a matching of S' in which all vertices in MAXM(R) n S' are matched to the same vertex in $[u_R,v_R]$ as in the matching Z. Let W be a corresponding matching for S. The existence of the matchings Y and W is a consequence of the construction used to show the feasibility of S.

From the definition of S', it follows that S'⊂/S. Also, from the working of FEAS, it follows that S⊂/S'. Let $j\in$S be a vertex with least h_j such that $j\notin$ S'. If no such j exists, then S=S'. Assume that j is matched to q in W. If q is free in Y, then S' cannot be of maximum cardinality. So, let $p\in$S' be matched to q in Y. By definition of Y and W, $p\notin$ S. Also, from the order in which FEAS considers vertices, $h_p\ge h_j$ (as otherwise, FEAS would consider p before j and select p for S). Hence, S'U{j}-{p} is also a subset selectable by Glover's algorithm (Since $h_p\ge h_j$, by

ensuring j<p, Glover's algorithm will be forced to match j before p.). S' U {j}-{p} agrees with S in one place more than does S'.

By repeating this interchange process, S' may be transformed into S with the result that S is also a maximum cardinality subset of MAXM(R) U T(L) that is selectable by Glover's algorithm for matching in $[u_R, v_R]$.

Hence, MAXM(L) U S is a maximum cardinality subset of M(P) selectable by Glover's algorithm for matching in [u,v].P. ▪

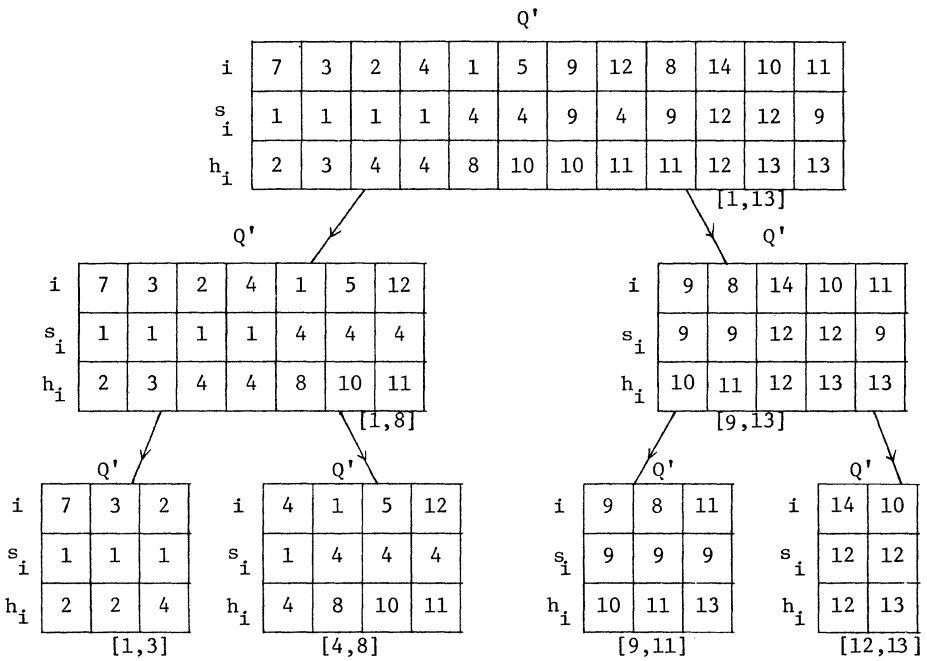
Once MAXM(P) is known, T(P) and I(P) are easily computed. Actually, as I(P) is never used, we may omit its computation. Figure 2.7 shows the MAXM, I, and T sets (except when empty) for all nodes in our example.

Pass 2

In the second pass, for each vertex P of the computation tree, we compute a set MAXM'(P) which represents the set of A vertices matched by Glover's algorithm with the set [u,v].P. With respect to the matching shown by solid lines in Figure 2.1, we see that if P is the root, then MAXM'(P) = {1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 14}; if P is node 3 of the computation tree, then MAXM'(P) = {8, 9, 10, 11, 14}.

If P is the root node, then MAXM'(P)=MAXM(P), by definition of MAXM(P). Let P be any nonleaf node for which MAXM'(P) has been computed. Let L and R be the left and right children, respectively, of P. Let [u,v].L = $[u_L, v_L]$ and [u,v].R = $[u_R, v_R]$. Let V = {j | j∈MAXM'(P) and $s_j < u_L$}. Let W be the ordered set obtained by merging together V and MAXM(L) (note that both V and MAXM(L) can be maintained so that they are in nondecreasing order of h_i and that W is also in nondecreasing order of h_i). MAXM'(L) consists of the first min {|W|, $v_L - u_L + 1$} vertices in W. The correctness of this statement may be established by induction on the level of P. MAXM'(R) is readily seen to be MAXM'(P)-MAXM'(L). Figure 2.8, shows the MAXM'( ) sets for all the vertices in the computation tree of our example.

From the MAXM'( ) sets of the leaves, the matching is easily obtained. If P is a leaf, and [u,v].P=[a,b], then the first vertex in MAXM'(P) is matched with a, the second with a+1, etc. (note that MAXM'(P) is in nondecreasing order of h_i). The matching for our example is also given in Figure 2.8.



matching={(7,1),(3,2),(2,3),(4,4),(1,5),(5,6),(12,7),(9,9),(8,10)
(11,11),(14,12),(10,13)}

Figure 2.8 Second pass

Complexity Analysis

The initial ordering of A by s_i and within s_i by h_i can be done in $O(\log^2 n)$ time using n/2 PEs([11] and [12]). During the first pass, the computation of MAXM() requires the use of FEAS and a merge. The use of FEAS (without the sort) takes O(logn) time and requires $O(|M(P)|/\log|M(P)|)$ PEs [1]. The merge at node P takes O(logn) time with $|M(P)|/2$ PEs. Since, MAXM can be computed in parallel for all nodes on the same level of the computation tree, O(logn) time is needed per level. The total time for the first pass is $O(\log^2 n)$ and n/2 PEs are needed. Pass 2 requires only some merging per node. The total cost of this pass is also $O(\log^2 n)$ and n/2 PEs suffice.

Hence, the overall complexity of our parallel algorithm for maximum matching in convex bipartite graphs is $O(\log^2 n)$. The PE requirement is O(n).

Another complexity measure often computed for parallel algorithms is the effectiveness of processor utilization (EPU) (see [1], [2], and [14]). For any problem P and parallel algorithm A, this is the ratio of the complexity of the fastest known algorithm for P and the product of the complexity of A and the number of PEs used by A.

For our algorithm, we have an EPU that is $\Omega((n+mA(m))/(\log^2 n * n))$ (recall that $m=|B|$).

3. Conclusions

This paper has further enhanced the utility of the binary tree method of Dekel and Sahni [2] for the design of parallel algorithms. It should also be pointed out that while all of our complexity analyses have assumed the availability of as many PEs as needed, our algorithms can be used when fewer PEs are available. The complexity of each algorithm will increase by no more than the shortfall in PEs. So if only half the number of PEs is available, then the time needed will at most double (except for a possible constant increase in overhead).

The parallel matching algorithm developed here can be used to obtain efficient parallel algorithms for several scheduling algorithms. These algorithms are developed in [15].

4. References

1. Dekel, E. and Sahni, S., "Parallel scheduling algorithms," *International Conference on Parallel Processing*, pp. 350-351, 1981. To appear in *Op. Res.*
2. Dekel, E., and Sahni, S., "Binary trees and parallel scheduling algorithms," In Lecture Notes In Computer Science, vol 111, CONPAR81, Springer Verlag, 1981, pp. 480-492.
3. Dekel, E., Nassimi, D., and Sahni, S., "Parallel matrix and graph algorithms", *SICOMP*, vol. 10, no, 4, Nov 1981, pp. 657-675.
4. Emde Boas, P. van, "Preserving order in a forest in less than logarithmic time and linear space," *Info. Proc. Let.*, 6, pp, 80-82, 1977.
5. Glover, F., "Maximum matching in a convex bipartite graph," *Naval Res. Logist. Quart.*, 14 (1967), pp. 313-316.
6. Jackson, J.R., "Scheduling a production line to minimize maximum tardiness," Research Report 43, Management Science Research Project, University of California, Los Angeles, 1955.
7. Lageweg, B.J. and Lawler, E.L., Private communication, cited in "Sequencing by enumerative methods," by J.K. Lenstra, p.22, Mathematisch Centrum, Amsterdam, 1976.
8. Lipski, W. Jr. and Preparata, F. P., "Efficient algorithms for finding maximum matchings in convex bipartite graphs and related problems," *ACTA Informatica*, 15, pp. 329-346, 1981.
9. McNaughton, R., "Sequencing with deadlines and loss functions," *Manag. Sci.*, **6, pp.1-12, 1959.**
10. **Muntz, R. R., and Coffman, E. G., Jr., "Optimal preemtive scheduling on two-processor systems,"** *IEEE Trans on Computer*, **c-18, (1969) pp. 1014-1020.**
11. **Nassimi, D., and Sahni S., "Bitonic sort on a mesh connected parallel computer,"** *IEEE Trans on Computers*, **c-28, no. 1, January 1979, pp.2-7.**
12. **Preparata, F. P., "New parallel-sorting schemes,"** *IEEE Trans. on Computers*, c-27, no.7, July 1978, pp.669-673.
13. Rinnooy Kan, A. H. G., "Machine scheduling problems, classification complexity, and computation". Nighoff, The Hague, 1976.
14. Savage, C., "Parallel algorithms for graph theoretic problems," Ph.D. Thesis, University of Illinois, Urbana, August 1978.
15. Dekel, E., and Sahni, S., "A parallel matching algorithm for convex bipartite graphs and applications to scheduling", University of Minnesota, Technical Report TR 81-3.

SIGNIFICANCE OF PROBLEM SOLVING PARAMETERS ON THE PERFORMANCE OF COMBINATORIAL ALGORITHMS ON MULTI-COMPUTER PARALLEL ARCHITECTURES

F. Gail Gray, W. M. McCormack, Robert M. Haralick

Dept. of Computer Science and Dept. of Electrical Engineering
Virginia Polytechnic Institute and State University
Blacksburg, Virginia

ABSTRACT (1)

This experiment has determined an optimum problem solving strategy for the consistent labeling problem. One combination of factors, depth first search strategy-transmit large problems-transmit 50% of a processor's work, was found to be statistically best, especially for large problem sizes or for architectures with restricted communications paths. Future work involves experimentation to understand the architecture related factors. The results in this paper indicate that the performance of the system, even using the optimum problem solving strategy, will vary considerably with architecture.

I. INTRODUCTION

Combinatorial problem solving underlies numerous important problems in areas such as operations research, non-parametric statistics, graph theory, computer science, and artificial intelligence. Examples of specific combinatorial problems include, but are not limited to, various resource allocation problems, the travelling salesman problem, the relation homomorphism problem, the graph clique problem, the graph vertex cover problem, the graph independent set problem, the consistent labeling problem, and propositional logic problems [12-15]. These problems have the common feature that all known algorithms to solve them take, in the worst case, exponential time as problem size increases. They belong to the problem class NP.

This paper describes the interaction between specific algorithm parameters and the parallel computer architecture. The classes of architectures we consider are those which have inherent distributed control and whose connection structure is regular.

---

(1) This work was supported in part by the Office of Naval Research Grant N00014-80-C-0689.

Combinatorial problems require solutions which do searching. To help in describing the parallel combinatorial search, we associate with the space yet to be searched the term "the current problem." A representation mechanism which can partition the space yet to be searched can divide the current problem into mutually exclusive subproblems.

Now suppose that one processor in a parallel computer is given a combinatorial problem. In order to get other processors involved, the processor divides the problem into mutually exclusive subproblems and gives one subproblem to each of the neighboring processors, keeping one subproblem itself. At any moment in time each of the processors in the parallel computer network may be busy solving a subproblem or may be idle after having finished the subproblem on which it was working. At suitable occasions in the processing, a busy processor may notice that one of its neighbors is idle. On such an occasion the busy processor divides its current problem into two subproblems, hands one off to the idle neighbor and keeps one itself.

The key points of this description are

1. the capability of problem division
2. the ability of every processor to solve the entire problem alone, if it had to.
3. the ability of a busy processor to transfer a subproblem to an idle neighbor.

The parallel computer architecture research issue is: to determine that way of problem subdivision which maximizes computation efficiency for each way of arranging a given number of processors and their bus communication links.

To define this research issue precisely requires

1. that we have a systematic parametric way of describing processor/bus arrangements and

0190-3918/82/0000/0185$00.75 © 1982 IEEE

2. that we have alternative problem subdivision techniques.

This paper addresses the interaction between the processor/bus graph and problem size subdivision transfer mechanism. Once these relationships are determined and expressed mathematically, the parallel computer architecture design problem becomes less of an art and more of a mathematical optimization.

Our ultimate goal is to allow computer engineers to begin with the combinatorial problems of interest and determine via a mathematical optimization, the optimal parallel computer architecture to solve the problems assuming that the associated combinatorial algorithms are given.

II. PROCESSOR-BUS MODEL

In this section we discuss a processor-bus model which can be used to model all known regular parallel architectures [1,3,4,7,8,10,21-26]. The model does not currently include the general interconnection and shuffle type networks.

The graphical basis for the model is a connected regular bipartite graph. A graph is bipartite if its nodes can be partitioned into two disjointed subsets in such a way that all edges connect a node in one subset with a node in the second subset. A graph is connected if there is a path between every pair of nodes in the graph. A bipartite graph is regular if every node in the first set has the same degree and every node in the second set has the same degree. One subset of nodes represents the processor nodes and one subset represents the communication nodes in the parallel processing system. Every edge in the graph then connects a processing node to a communication node.

Any regular bipartite graph can be used to design a parallel computer structure by assigning the nodes in one set to be processors and the nodes in the other set to be communication links (or buses). Notice that theoretically either set of the bipartite graph could be the processor set. Therefore, each unlabeled bipartite graph represents two distinctly different computer architectures depending upon which set is considered to be processors and which set is considered to be the buses.

The notation $B(n_p,d_p,n_c,d_c)$ will be used to denote a regular bipartite graph which represents an architecture with n_p processors (each connected to d_p communication nodes) and n_c communication nodes (each servicing d_c processors). The Boolean 3-cube will then be represented by a graph B(8,3,12,2). In general, the Boolean n-cube will be represented by a graph $B(2^n, n, n2^{n-1}, 2)$. Reversing the assignment of nodes to processors and buses produces the B(12,2,8,3) graph which is called the p-cube by some investigators.

Other common architectures also have representations as bipartite graphs. For example, a planar array of size x^2 connected in the Von Neumann manner is modeled as a $B(x^2,4,2x^2,2)$ graph, the Moore connection results in a $B(x^2,8,4x^2,2)$ graph, the common bus architecture (or star) with x processors is a B(x,1,1,x) graph, and the common ring architecture is a B(x,2,x,2) graph. All existing architecures with regular local neighborhood interconnections can be modeled as a $B(n_p,d_p,n_c,d_c)$ graph.

III. PROBLEM SOLVING FACTORS

Introduction to Tree Searching

In order to make effective use of a multiple asynchronous processor for any problem, a major concern is how to distribute the work among the processors with a minimum of interprocessor communication. Kung [14] defines module granularity as the maximal amount of computational time a module can process without having to communicate. Large module granularity is better because it reduces the contention for the buses and reduces the amount of time a processor is either idle or sending or receiving work. Also, large granularity is usually better because of the typically fixed overhead associated with the synchronization of the multiple processors.

In the combinatorial tree search problems we are considering, module granularity as defined by Kung is not as meaningful because each processor could in fact solve the entire problem by itself without communicating to anybody. For our problem a more appropriate definition of module granularity might be the expected amount of processing time or the minimum amount of processing time before a processor splits its problem into two subproblems, one of which is given to an idle neighboring processor and one of which is kept itself.

When a processor has finished searching that portion of the tree required to solve its subproblem, it must wait for new work to be transferred from another processor. The amount of time a proces-

sor must wait before transmission begins and until transmission is completed is time wasted in the parallel environment that would not be lost in a single processor system. Thus, one must expect improvement in the time to completion to solve a problem in the multiple processor environment to be less than proportional to the number of processors. The factors that can affect the performance by either reducing the average transmission time or reducing the required number of transmissions include choice of algorithm, choice of search strategy, and choice of subproblems that busy processors transfer to idle processors.

Choice of Algorithm

In the single processor case, various algorithms have been proposed and studied to efficiently solve problems requiring tree searches. These usually involve investing an additional amount of computation at one node in the tree in order to prune the tree early and avoid needless backtracking. In work on constraint satisfaction [11], the forward checking pruning algorithm was found to perform the best of the six tested and backtracking the worst.

For the same reasons, it seems clear that pruning the tree early should be carried over to a multiple processor system to reduce the amount of computation necessary to solve the problem. There are other reasons as well. Failure to prune the tree early may later result in transfers to idle processors of problems which will be very quickly completed. Since a transfer ties up, to some extent, both the sending and receiving processor, time is lost doing the communication and the processor receiving the problem would shortly become idle.

We would, therefore, expect that in the multiple processor environment the forward checking pruning algorithm for constraint satisfaction would work much better than backtracking. However, in the uniprocessor environment Haralick and Elliott also showed that too much look ahead computation at a node in the search could actually increase the problem completion time. It is not clear that this would be true in the multiple processor case. It may be best to do more testing early reducing future transfers, communication overhead, and delay in contrast to the single processor case where only some extra testing has been found to be worthwhile.

A second consideration in the selection of a search algorithm is the amount of information that must be transferred to an idle processor to specify a subproblem and any associated lookahead information already obtained pertinent to the subproblem. In most cases this is proportional (or inversely proportional) to the complexity of the problem remaining to be solved. Thus the transmission time will be a function of the problem complexity. Backtracking requires very little information to be passed while, for forward checking, a table of labels yet to be eliminated must be sent.

Search Strategy

Search strategy is a second factor of importance to the multiple processor environment. When a problem involves finding all solutions, like the consistent labeling problem, the entire tree must be searched. Thus, in a uniprocessor system the particular order in which the search is conducted, i.e., depth first or breadth first, has no effect. In a multiple processor system, however, this is a critical factor because it directly affects the complexity of the problems remaining in the tree to be solved and available to be sent to idle processors from busy processors.

A depth first search will leave high complexity problems to be solved later (that is, problems near the root of the tree.) This would seem to be desirable in the multiple processor environment because passing such a problem to an idle processor would increase the length of time the processor could work before going idle and thereby reduce the need for communication. On the other hand, a breadth first search would tend to produce problems of approximately the same size. Since the problem is not completed until all processors are finished, the breadth first strategy might be preferable if it results in all processors finishing at about the same time. It might be that the best approach could be some combination of the two; for example, one might follow a depth first strategy for a certain number of levels, then go breadth first to a certain depth, and then continue depth first again.

Problem Passing Strategy

A factor closely related to the search strategy occurs when a processor has a number of problems of various complexities to send to an idle processor. The optimization question is how many should be sent and of what complexity(ies). Further complicating this is a situation where the processor is aware of more than one idle processor. In such a situation, how should the available work be divided and still leave a significant amount for the sending processor?

Further complicating this question is the fact that the overhead involved in synchronizing the various processors and transmitting problems to idle ones will eventually reach a point where it will be more than the amount of work left to be done. An analogous situation exists in sorting; fast versions of QUICKSORT eventually resort to a simple sort when the amount remaining to be sorted is small [13].

In this case, it would appear that a point will eventually be reached where it is more effective for a processor simply to complete the problem itself rather than transmit parts of it to others. Determination of this point will depend on the depth in the tree of the problem to be solved and the amount of information that must be passed (which depends on the lookahead algorithm being used.)

Processor Intercommunication

One decision that has to be made is how the need to transfer work is recognized. Specifically, does a processor which has no further work interrupt a busy processor, or does a processor with extra work poll its neighboring processors to see if they are idle.

The advantage of interrupts is that as soon as a processor needs work, it can notify another processor instead of waiting to be polled. This assumes, however, that a processor would service the interrupt immediately instead of waiting until it had finished its current work. A disadvantage is that when a processor goes idle, it cannot know which of its neighbors to interrupt. Using polling, an idle processor can be sent work by any available neighboring processor instead of being forced to choose and interrupt one. In addition, although an interrupted processor may be working or transmitting (a logical and necessary condition) when interrupted, it may not have a problem to pass when it is time to pass work to the interrupting processor. In fact, the interrupted processor could itself go idle. For these reasons the simulation we discuss in section IV uses polling. Whenever a processor completes a node in the tree, and as long as it has work it could transfer, it checks each neighboring CPU and the connecting bus. If both are idle, a transfer is made.

IV. SIMULATION EXPERIMENTS

In order to better understand the behavior of the tightly coupled asynchronous parallel computer, we have designed a series of simulation experiments using the consistent labeling constraint satisfaction problem. The simulation used to perform these experiments was written in SIMULA [Birtwistle,, Myhrhaug & Nygaard, 1973]. Let U and L be finite sets Let $R \subseteq (U \times L)^2$. We use the simulated parallel computer to find all functions $f: U \rightarrow L$ satisfying that for all $(u,v) \in U \times U, (u,f(u),v, f(v)) \in R$. The goal of the experiments is to determine which architectural and which problem related factors are significant enough to warrant further investigation. This paper presents the results for problem related factors.

In this experiment each problem factor was tested at two levels. The factors and levels tested are given in Table 1. Based on previous experiments [16], it was very clear that forward-checking was significantly better than backtracking so all experiments used the forward-checking algorithm [11]. In order that the results be applicable for different architectures and problem sizes, two problem sizes (small and medium) and two very different architectures (in terms of the number of communication paths) were used. The architectures chosen were symmetric to eliminate the need for assumptions about the architecture related factors discussed earlier. The ring architecture, B(64, 2, 64, 2), due to the limited interconnection structure, will have difficulty passing work from the initial processor to distant processors. The Boolean 6-cube B(64, 6, 192, 2), should be able to effectively utilize most of the 64 processors. Finally, one replication was run of each combination. This involves running the simulation with different random number seeds to create statistically equivalent combinatorial problems. An analysis of variance was used to determine the significance of the problem related parameters and to determine interactions of the parameters [20]. The measure of performance used was the time until the problem was solved.

Results

The analysis of variance was done using the SAS (Statistical Analysis System) package. The analysis showed statistically significant differences in the means (at a level of 0.0001), and second and third order interactions for the search strategy, size passed, and number passed. The means for the two cutoff point levels were not statistically dif-

ferent. Because the three way interaction among strategy, size, and number was significant, the combinations of these three factors were treated as eight levels of one combined factor for further analysis.

Duncan's multiple range test was performed [20] to divide the levels into groups with similar performance. The results, based on the average time to completion for the different experimental conditions, are shown in Table 2.

The key result is that one combination is clearly superior, depth-large-50%, and should be used in further experiments. (This combination also produced the lowest mean for each of the four architecture-problem size pairs.)

There is a logical explanation for the groupings. For each factor one value can be classified as positive (i.e., it should contribute to improved performance regardless of other factors), and the other negative (i.e., it should result in poorer performance). The positive factors are indicated as level 1 in Table 1. For example, passing more than one sub-problem or passing large sub-problems should be preferable as the idle processor should stay busy longer. Since in a depth first search a processor works on small problems, this should leave larger problems to pass. As a result communication time is reduced.

Using this idea of a positive level for each factor, only one combination has all 3 levels positive, three have two positive, three have one positive, and one no positive levels. The grouping produced by Duncan's test confirms this analysis and, in fact, produces a finer partition. Thus, the interaction between these factors agrees with the analysis. The analysis of variance also indicated significant interactions between the combined factor and the experimental conditions of problem size and architecture. To best understand these interactions, the values were plotted as suggested by Cox [6]. (Figures 1,2,3). If there were no interaction, then the curves in each figure would be parallel.

Figure 1 shows a clear interaction between problem size and architecture. For a small problem, a small number of processors is sufficient; thus, the inability of the ring to spread sub-problems to idle processors is not a severe handicap. However, for a larger problem, the performance of the ring is much worse than that of the 6-cube which is able to involve many more of the processors. In each case the time to completion was approximately 3 times longer in the ring architecture. Since the degree of each processor node in the ring is 1/3 of the degree of each processor node in the Boolean 6-cube, it appears that performance may be proportional to the degree of the processor nodes. This has intuitive appeal because more communication paths should improve the ability of processors to keep busy. Later experiments will confirm or deny this conjecture. It is also possible that diminishing returns may set in for extremely large numbers of communication nodes. This plot indicates that the use of an optimum architecture becomes more crucial for large problems.

Figure 2 shows the interaction of the combined problem solving factor with problem size. Clearly, the need to determine the best combinations of problem solving factors becomes more critical as the size of the problem increases because a bad choice has a greater detrimental effect on the larger problem.

Figure 3 shows the interactions of the combined problem solving factor with architecture type. This plot shows that an optimum choice of problem-solving factors tends to reduce the effects of a bad choice of architecture. However, the difference in performance between the architectures using the optimum problem solving strategy is still a factor of 3, so that further experiments to determine an optimum architecture seem justifiable.

REFERENCES

(1) Anderson, G. A., and E. D. Jensen, "Computer Interconnection Structures: Taxonomy, Characteristics, and Examples", Computing Surveys, Vol. 7, Dec. 1975, pp. 197-213.

(2) Armstrong, J. R. and F. G. Gray, "Some Fault Tolerant Properties of the Boolean n-Cube", Proceedings of the 1980 Conference on Information Sciences and Systems, Princeton, NJ, March 26-28, 1980, pp. 541-544.

(3) Benes, V. E., "Optimal Rearrangeable Multistage Connecting Networks", Bell System Technical Journal, July 1964, pp. 1641-1656.

(4) Batcher, K. E., "Sorting Networks and Their Applications", Spring Joint Computer Conference, 1968, pp. 307-314.

(5) Birtwistle, G. M., Dahl, O. J., B. Myhrhaug, and K. Nygaard, SIMULA Begin, Auerbach Publishers Inc., Philadelphia, PA, 1973.

(6) Cox, D. R., Planning of Experiments, John Wiley & Sons, Inc., New York, 1958.

(7) Despain, A. M. and D. A. Patterson, "X-Tree: A Tree Structured Multi-processor Computer Architecture", 5th Annual Symposium on Computer Architectura, architecture, 1978, pp. 144-151.

(8) Finkel, R. A. and M. A. Solomon, "Processor Interconnection Strategies", IEEE Transactions on Computers, Vol. C-29, May 1980, pp. 360-370.

(9) Foster, M. J. and H. T. Kung, "The Design of Special Purpose VLSI Chips", Computer, Jan. 1980.

(10) Goke, R. L. and B. S. Lipovski, "Banyon Networks for Partitioning Multiprocessor Systems", Proceedings of First Conference on Computer Architecture, 1974, pp. 21-28.

(11) Haralick, Robert M. and G. Elliott, "Increasing Tree Search Efficiency for Constraint Satisfaction Problems", Artificial Intelligence, Vol. 14, 1980, pp. 263-313.

(12) Hillier, F. S. and G. S. Lieberman, Operations Research, Holden Day, Inc., San Francisco, 1979.

(13) Knuth, D. E., The Art of Computer Programming, Sorting and Searching, Addison-Wesley Publishing, Reading, MA, 1973.

(14) Kung, H. T., "The Structure of Parallel Algorithms", in Advances in Computers, Vol. 19, edited by M. D. Yovits, Academic Press, 1980.

(15) Lee, R. B., "Empirical Results on the Speed, Redundancy and Quality of Parallel Computations", Proceedings of 1980 International Conference on Parallel Processing, 1980.

(16) McCormack, W. H., F. G. Gray, J. G. Tront, R. M. Haralick and G. S. Fowler, "Multi-Computer Parallel Architectures for Solving Combinatorial Problems", Multi-Computer Architectures and Image Processings: Algorithms and Programs, Academic Press, New York, 1982.

(17) Mead, C. A. and M. Rem, "Cost and Performance of VLSI Computing Structures", IEEE J. Solid State Circuits, sc-14(2), pp. 455-462, 1979.

(18) Mead, C. A. and L. A. Conway, Introduction to VLSI Systems, Addison-Wesley, Reading, MA, 1980.

(19) Mirza, J. H., "Performance Evaluation of Pipeline Architectures", Proceedings of 1980 International Conference on Parallel Processing, 1980.

(20) Ott, Lyman, An Introduction to Statistical Methods and Data Analysis, Duxbury Press, North Scituate, MA, 1977.

Table 1 - Experiment Summary

FACTORS TESTED

FACTOR	LEVEL 1	LEVEL 2
search strategy	depth-first	breadth-first
size of sub-problem passed	largest	smallest
number of sub-problems passed	50% of expected total work	1 sub-problem
cutoff point	none	4 units to be tested

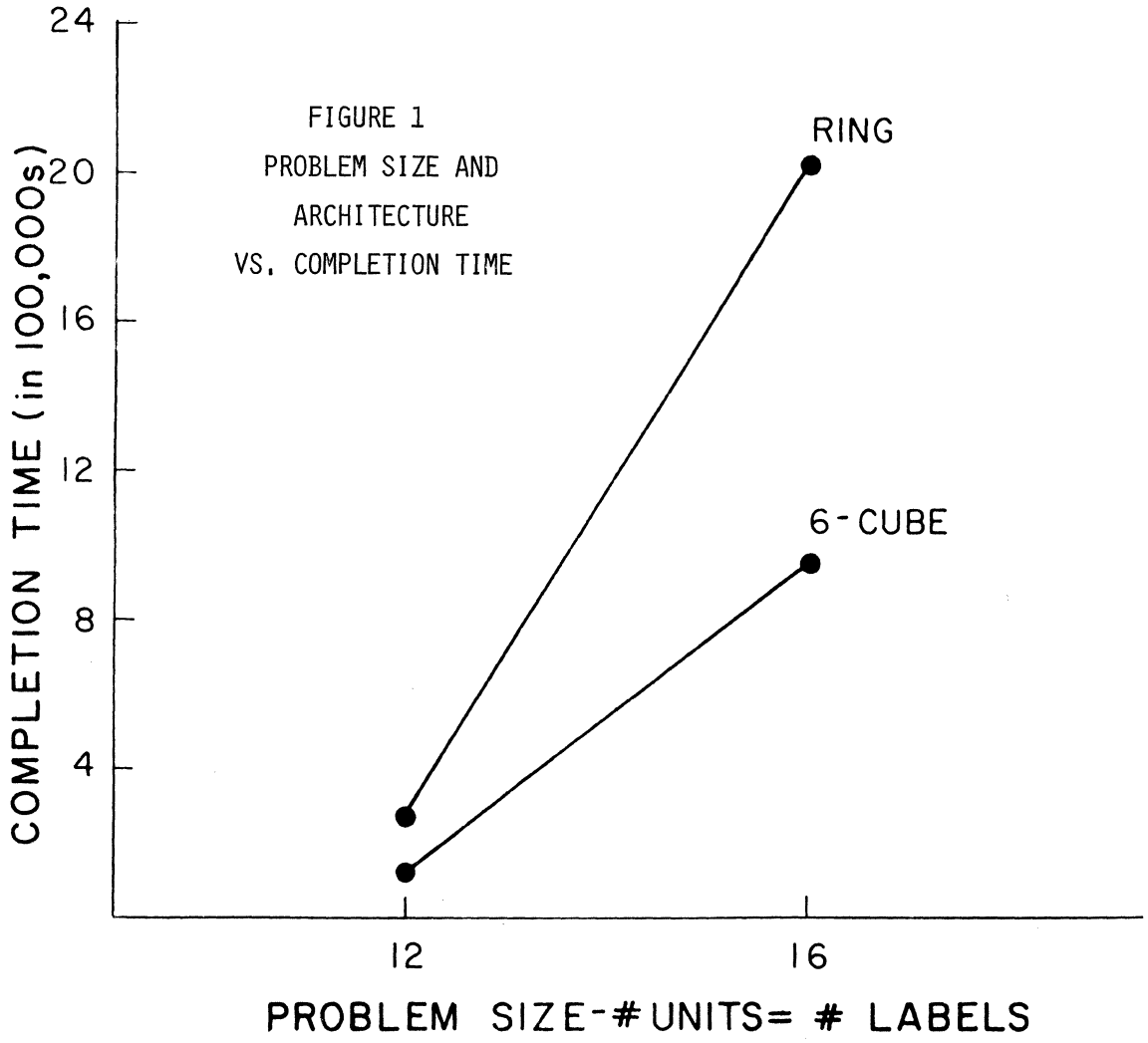
EXPERIMENTAL CONDITIONS

Architecture	Ring	6-cube
number of processors	64	64
number of buses	64	192
Size of combinatorial problem	small - 12 units & labels	medium - 16 units & labels
One replication	random	random

Table 2 - Duncan's Multiple Range Test

GROUPING*	MEAN COMPLETION TIME	ID NUMBER	FACTOR COMBINATION SEARCH	SIZE	NUMBER
A	2,705,274	8	breadth	small	one
B	1,874,887	7	breadth	large	one
C	689,372	4	depth	small	one
D	451,133	6	breadth	small	50%
E	335,267	5	breadth	large	50%
F E	301,774	3	depth	large	one
F	247,667	2	depth	small	50%
G	147,181	1	depth	large	50%

*means with the same grouping are not significantly different
significance level = 0.05



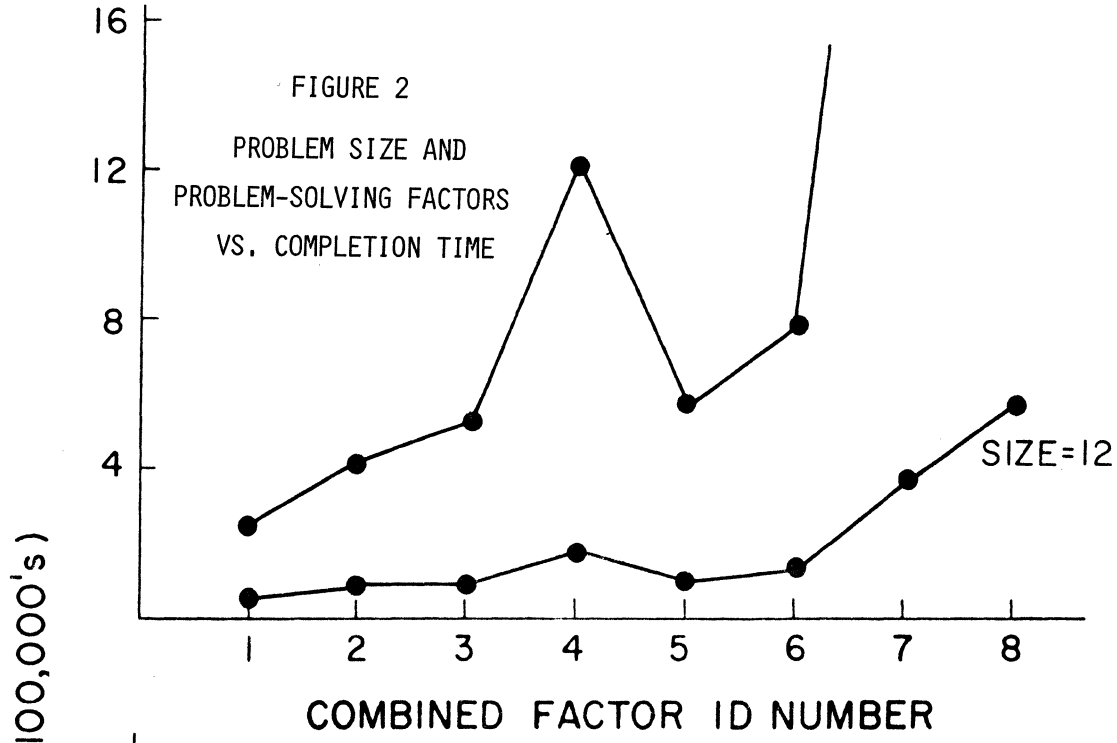


FIGURE 2

PROBLEM SIZE AND
PROBLEM-SOLVING FACTORS
VS. COMPLETION TIME

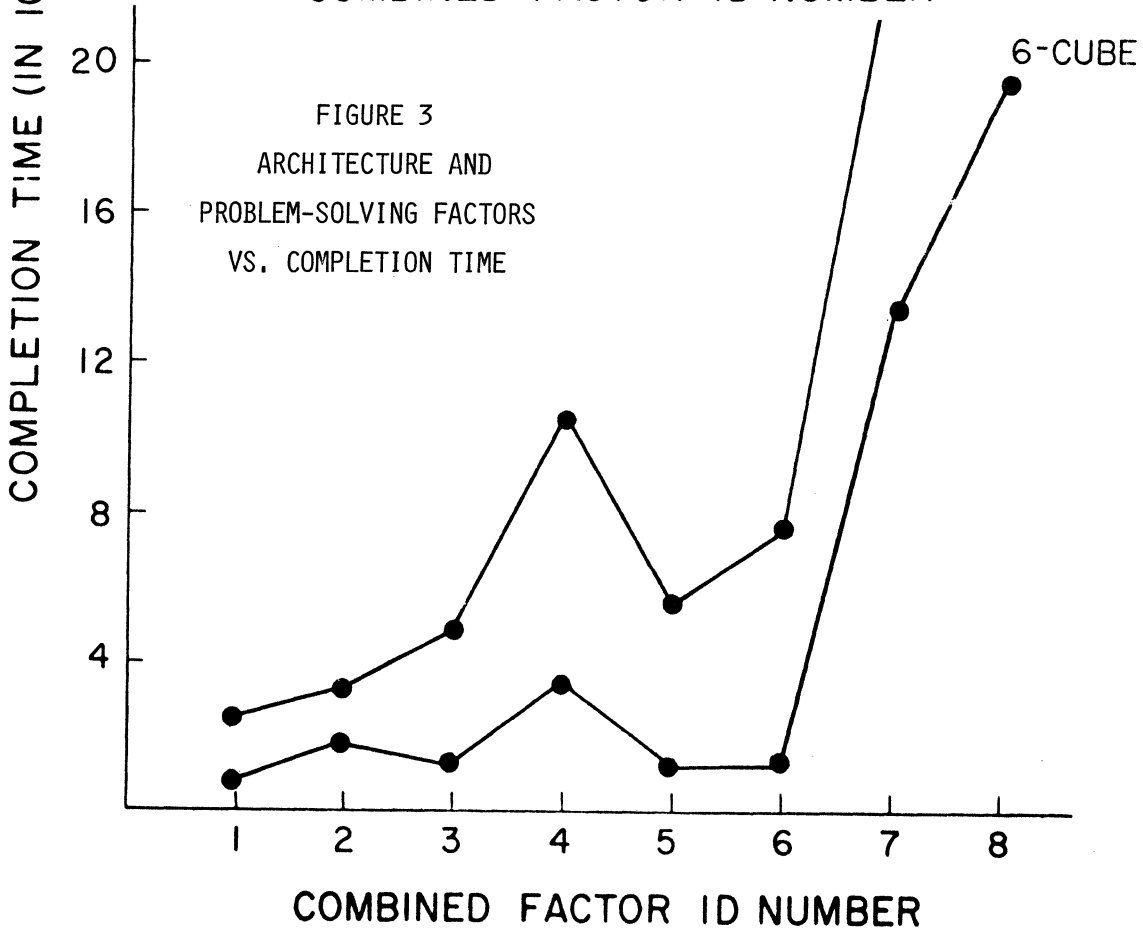


FIGURE 3

ARCHITECTURE AND
PROBLEM-SOLVING FACTORS
VS. COMPLETION TIME

NOVAC - A NON-TREE VARIABLE TREE FOR COMBINATORIAL COMPUTING

B.C. Desai C. Lam J.W. Atwood
J. Opatrny P. Grogono S. Cabilio

Computer Science Department
Concordia University
Montreal, Quebec, Canada

Abstract - - In exact computation, a number of problems exist, the solution to which demands an exhaustive search and hence a great deal of computing time. The algorithms used are simple but the computation involved is so great that it cannot be done economically on a large scale time-shared general purpose computer. The present multiprocessor project at Concordia consists of a dynamically variable virtual tree structured system for solving a class of combinatorial problems. The proposed multiprocessor structure consists of loosely coupled processors with no shared memory. Each processor in the system can be a master or a slave or both, and under certain conditions a master processor can become a slave of its own slave processor. A master assigns tasks to the slaves and subsequently obtains results from them. The nature of the problem being solved and the high bandwidth of the interprocessor communication bus is expected to cause inappreciable degradation due to contention. The user expresses the problem being solved in a high level language called Pascal-C; this is conventional Pascal with a number of additional constructs including synchronization statements. Another part of the project involves designing the extensions to an existing operating system to support this dynamically variable structure and the runtime system of Pascal-C.

Introduction

There are many problems in exact computation requiring a great amount of computing time to solve them. The computations involved are simple however the amount of computation is so great that it cannot be done economically on a large scale general purpose computer. Parallel processing of subproblems derived from a large class of problems on multi-microprocessors is becoming increasingley feasible economically. Interprocess communication of these processors is implemented by an interconnection network. A number of surveys of interconnection networks have appeared in relevant literature, eg. [7]. Numerous systems have been proposed to exploit the parallelism in such problems. A computer structure in the form of a tree has been proposed in [1]; a system to solve problems that may be expressed with recursive algorithms is presented in [4]. In [5], a microprocessor based system has been described for the 0-1 programming problem. A number of adaptive computer architecture schemes have been proposed recently; [9,12] are examples of such systems. However, many of these systems are in the developmental or experimental stage and/or are too expensive for general availability.

The NOVAC project at Concordia consists of a loosely coupled, non-tree structured multiprocessor system with the potential of being dynamically structured into a virtual variable tree to solve a class of combinatorical problems. This system is to be built with off-the-shelf mini and micro computers, and interconnected using an inexpensive asynchronous bus. The logical structure, consisting of a hierarchy of masters, each with a number of slaves is natural for the set of problems which can be split-up into a number of identical sub-problems.

Novac Hardware

The hardware, Figure 1, consists of a number of PDP/11 based processing systems; each processor in the system has its own private memory and is under control of its own operating system. Each processor executes independently and communication between processors is via the common Novacbus. Each processor thus, has the same physical status as any other processor. The initial system consists of a PDP-11/34 and several LSI-11/23 processors. The PDP-11/34 is equipped with conventional peripherals (terminal, printer, and disk drives) an an UNI (Unibus to Novacbus) interface to the Novacbus, but the LSI-11/23 has only a UNI interface, and as such can only communicate with the other processors in the system.

The proposed system has the following features (i) the problem presented to it can be solved by the same program code (a copy of the code is resident in the memory of each processor); (ii) the logical tree structure with a master-slave relationship of the processors: the master assigns the subtasks to the slaves; the slaves can act as masters and divide their tasks and assign them to their slaves; (iii) there is a main master at the "root" of the tree and the user communicates his problem via this master; (iv) there is no shared memory and communication is limited between the master and slave; (v) no communication exists amongst the slaves; (vi) the amount of communication between the master and slave is not extensive; (vii) the processors are interconnected via the UNI interface to the Novacbus.

Since the amount of communication between the processors is limited, a single asynchronous bus of high bandwidth to serve the maximum number of processors is proposed; the high bandwidth keeps the contention for use of this bus low. The proposed channel will support a hierarchy of communication protocols from high level virtual communication between programs, to low-level physical communication between hardware units.

0190-3918/82/0000/0193$00.75 © 1982 IEEE

This system which has only one interconnection per processor has the following drawbacks. In applications where interprocessor communication is very high the system will degrade considerably. However, in compute bound situations where the ratio of local processing to interprocess communication requirements is high (ie. where for each word of interprocess communication, the number of instructions executed is of the order of 10^3 to 10^6) this interconnection will allow a large number of processors to be interconnected. Each processor in this system has its own copy of the program code which makes inefficient use of memory. In addition, data must be transmitted from the master processor to the slave processor instead of pointers. However, in the applications considered for NOVAC, where the amount of communication is limited, the transmission time for interprocessor communication is expected to be low.

The proposed interconnection is simple and inexpensive while providing for modularity. The communication protocol is simple to set up and control.

Pascal-C

The design of the language for the multiprocessor system is based on the following objectives and assumptions:

1. Programs for the system can be developed in a familiar high-level language which has been augmented with only a few new constructs;

2. There should be specific language constructs to allow efficient and simple utilization of the processors in the system;

3. Synchronization of processes is simple or even unnecessary in most computationally bound combinatorial problems.

Since none of the well known languages for concurrent programming eg., Concurrent Pascal [2], Modula [11], Ada [8], Edison [3] satisfied our specific requirements, we decided to use Pascal-C which is Pascal, augmented by these three additional constructs: down procedures, critical procedures, and synchronization statements. The syntax and the usage of these constructs are given below, however the details of these extensions are given in [10].

The procedure and function declaration part of a block in Pascal-C may include declarations of critical procedures and down procedures.

<critical procedure declaration>::=critical
<procedure declaration>

<down procedure declaration>::=down
<procedure heading>
<copy section>
<block>

<copy section>::=copy<identifier>{,<identifier>}
| <empty>

Thus, in the declaration of a critical procedure the keyword critical precedes the keyword procedure. In the declaration of a down procedure the keyword down precedes the keyword procedure. Furthermore, the heading of a down procedure can be followed by the keyword copy and a list of identifiers containing global variables, procedures and functions which can be used in the statements of the down procedure.

The following synchronizing statements are available in Pascal-C:

<wait statement>::=wait (<identifier>
{,<identifier>})

<terminate statement<::=terminate (<identifier>
{,<identifier>})

where <identifier> must be a name of a down procedure.

In addition to the scope level as usually defined in Pascal, we will also define for each element of the language, (e.g. variable, function, procedure, down procedure) its process level. This process level indicates the nesting level with respect to down procedures. The critical and down procedures cannot be recursive; in addition the critical procedure cannot be nested or call another critical procedure, and all its parameters must be value parameters.

An invocation of a down procedure creates an independent concurrent process in a slave processor. A critical procedure call in a slave processor creates a new process in the master processor. Critical procedures are used by a slave to pass its results back to the master. Synchronization statements allow the process in a master to wait for the results of its slave(s), or to terminate processes in its slaves. Down procedure statements and critical procedure statements are used to invoke down procedures and critical procedures, respectively. The syntax of down procedure statements and critical procedure statements is identical to the syntax of ordinary PASCAL procedure statements. The actual parameter of a down procedure corresponding to a variable parameter must be a variable whose scope includes the down procedure.

The present design of Pascal-C does not allow for the use of pointers as variable parameters to down procedures or in copy sections.

Pascal-C has been used successfully to program and dry run several different kinds of combinatorial problems.

Implementation of Novac

Here are the three major areas in which implementation effort is underway:

1. Assembly of the NOVAC-Tree hardware units. This will be from off-the-shelf systems with very little additional hardware to provide interprocessor communication.

2. Construction of an operating system (OS) that provides the multiprocessing capabilities required for the NOVAC-Tree system and the Runtime system (RTS) of Pascal-C. The OS is based on RT-11 which allows the processors connected to the NOVAC bus to appear as ordinary peripherals. The RT-11 OS will require additional features; for example, it must support the mapping between virtual and real slave processors; it must allow atomic execution of critical procedures.

3. Construction of a compiler and runtime system for Pascal-C. The proposed language closely resembles standard Pascal, and it should be possible to adapt an existing Pascal compiler to the requirements of the project. Most of the support for non-standard features is provided by the RTS, and a substantial proportion of the language implementation effort will be directed towards the RTS.

The progress of this project and experience gained from it will be revealed in a future paper.

References

[1] J.L. Bentley, H.T. Kung, "Two Papers on a Tree-Structured Parallel Computer", CMU CS-79-142, Carnegie Mellon University, Pittsburgh, Pa.

[2] P. Brinch Hansen, "The Programming Language Concurrent Pascal", IEEE Transactions on Software Engineering, 5(2), 1975, p.199-207.

[3] P. Brinch Hansen, "Edison-a Multiprocessor Language", Software-Practice and Experience, 11(4) 1981, p.325-360.

[4] B. Buchberger, J. Fegerl and F. Lichtenberger, "Computer Trees: A Multicomputer Concept for Special Purpose Parallel Processing", Microprocessors and Microsystems, 3(6) July/August, 1979.

[5] B.C. Desai, "A Parallel Processing System to Solve 0-1 Programming Problem", Ph.D. Thesis, McGill University, January 1977.

[6] D.J. Farber, "A Ring Network", Datamation, February 1975.

[7] Tse-yun Feng, "A Survey of Interconnection Networks", Computer, December 1981, p.12-27.

[8] J. Ichbiah et al, "Reference Manual for the Ada Programming Language", U.S. Department of Defence Publication, 1980.

[9] S.I. Kartashev and S.P. Kartashev, "Multicomputer System with Dynamic Architecture", IEEE Transactions on Computers, 28(10), October 1979, p.704-721.

[10] C. Lam, J.W. Atwood, S. Cabilio, B.C. Desai, P. Grogono, J. Opatrny, "A Multiprocessor Project for Combinational Computing", CIPS-82, Saskatoon, Sask. Canada.

[11] N. Wirth, Modula: "A Language for Modular Multiprogramming", Software-Practice and Experience, 7(1), 1977, p.3-35.

[12] L.D. Wittie, MICROS, "A Distributed Operating System for MICRONET, A Reconfigurable Network Computer", IEEE Transactions on Computers, C29(12), December 1980, p.564-572.

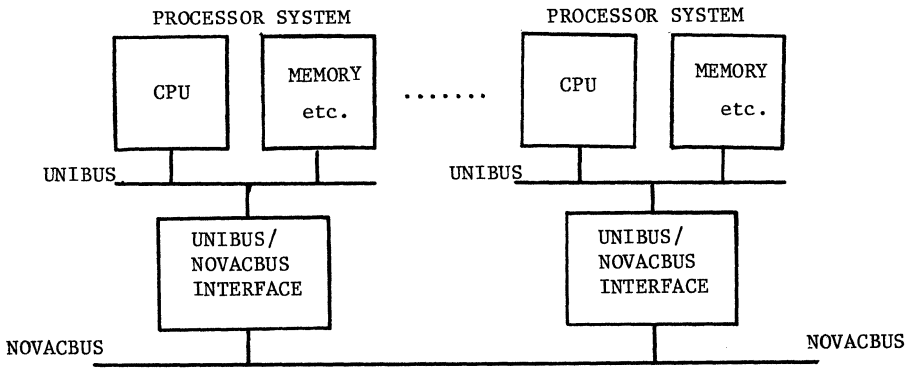


Figure 1. Proposed NOVAC System

RESULTS IN PARALLEL SEARCHING, MERGING, AND SORTING
(Summary)

Clyde P. Kruskal

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

Introduction

Comparison problems such as merging and sorting are of fundamental importance in computer science, and much effort has been devoted to finding efficient algorithms for solving these problems on sequential processors. Recently a similar effort has been devoted to to solving these problems on parallel processors [1] - [7]. In this paper we present parallel algorithms for searching, merging, and sorting which have good worst-case performances.

Initially, our algorithms are presented under Valiant's model of parallel computation [7] which captures the inherent difficulty of solving comparison problems (the notation has been changed slightly to be consistent with ours):

> ... there are P processors available, and therefore P comparisons can be performed simultaneously. The processors are synchronized so that within each time interval each of them completes a comparison. At the end of the interval the algorithm decides, by inspecting the ordering relationships that have already been established, which P (not necessarily disjoint) pairs of elements are to be compared during the next interval, and assigns processors to them. The computation terminates when the relationships that have been discovered are sufficient to specify the solution to the given problem.

Under this model Valiant presented algorithms for merging and sorting that were faster than any previously known.

More realistic models of parallel computation are shared-memory machines. One particular model of shared-memory machine is the CREW P-RAM which in a single cycle allows the processors to perform concurrent reads from the same location but not concurrent writes (Concurrent Read Exclusive Write Parallel Random Access Machine). Although shared-memory machines with single cycle memory access are not physically constructible -- at least for very large numbers of processors -- the study of their performance can be a powerful tool for gaining insight into the nature of parallel computation. Borodin and Hopcroft [1] showed that Valiant's merging and sorting algorithms can, in fact, be implemented on a CREW P-RAM.

We improve on the results of Valiant. For example, we present a merging algorithm that is optimal up to a constant factor when merging two lists of equal size, independent of the number of processors; in particular with N processors it merges two lists each of size N in $1.893 \lg\lg N + 4$ comparison steps. We then use the merging algorithm to obtain a sorting algorithm which, in particular, sorts N values with N processors in $1.893 \frac{\lg N \lg\lg N}{\lg\lg\lg N}$ (plus lower order terms) comparison steps. All of our algorithms can be implemented on a CREW P-RAM.

We define $\text{Search}_P(N)$ to be the number of comparison steps required by P processors to search a sorted list of N elements for some specified value, $\text{Merge}_P(M,N)$ to be the number of comparison steps required by P processors to merge two sorted lists of sizes M and N, and $\text{Sort}_P(N)$ to be the number of comparison steps required by P processors to sort N elements. Throughout this paper, we write $\lg x$ for $\log_2 x$ and $\ln x$ for the natural logarithm $\log_e x$.

Searching and Merging

This section contains results for parallel searching and merging. The general outline of the section closely follows Valiant [7] (Section 2), and several of the algorithms are improvements of Valiant's.

The following theorem generalizes the sequential algorithm for searching a sorted list to the parallel case.

Theorem 1. $\text{Search}_P(N) \leq \left\lceil \frac{\log(N+1)}{\log(P+1)} \right\rceil$. Furthermore, the bound is tight.

Proof. We show by induction that in k comparison steps we can search a sorted list of size $(P+1)^k-1$: The formula certainly holds for k=0. Assume it holds for k-1. Then to search a list of size $(P+1)^k-1$, we can compare the element being searched for to the elements in the sorted list subscripted by $i\cdot(P+1)^{k-1}$ for $i = 1,2,\ldots$. There are no more than P such elements [since $(P+1)(P+1)^{k-1} = (P+1)^k > (P+1)^k-1$]. Thus the

This work was supported in part by the National Science Foundation under Grant No. NSF-MCS81-05896.

0190-3918/82/0000/0196$00.75 © 1982 IEEE

comparisons can be performed in one step, and the problem is reduced to searching a list of size $(P+1)^{k-1}-1$. In general to search a list of size N in k comparison steps we need

$$(P+1)^k - 1 \geq N$$

or $$k \geq \frac{\log(N+1)}{\log(P+1)}$$

or $$k = \left\lceil \frac{\log(N+1)}{\log(P+1)} \right\rceil .$$

We now show that the bound is tight. Given a sorted list of N elements, during the first step any algorithm can examine only P elements. Some segment of unexamined elements must have length at least

$$\left\lceil \frac{N-P}{P+1} \right\rceil \geq \frac{N-P}{P+1} = \frac{N+1}{P+1} - 1 .$$

By induction after the k-th step the problem must have size at least $\frac{N+1}{(P+1)^k} - 1$. Thus the number of steps required by any algorithm is at least the minimum k for which

$$\frac{N+1}{(P+1)^k} - 1 \leq 0$$

or $$k \geq \frac{\log(N+1)}{\log(P+1)}$$

or $$k = \left\lceil \frac{\log(N+1)}{\log(P+1)} \right\rceil .$$

Corollary 1. $\mathrm{Merge}_P(1,N) \leq \left\lceil \frac{\log(N+1)}{\log(P+1)} \right\rceil$. Furthermore, the bound is tight.

Corollary 2. For $1 \leq M \leq P$ and $M \leq N$,

$$\mathrm{Merge}_P(M,N) \leq \left\lceil \frac{\log(N+1)}{\log(\lfloor P/M \rfloor + 1)} \right\rceil .$$

Proof. Assign $\lfloor P/M \rfloor$ processors to each element in the smaller list and merge as in Corollary 1.

Corollary 3. For $1 \leq P \leq M \leq N$,

$$\mathrm{Merge}_P(M,N) \leq \lceil M/P \rceil \lceil \lg(N+1) \rceil .$$

Proof. Assign $\lceil M/P \rceil$ elements in the smaller list to each processor and merge as in Corollary 1.

Theorem 2. For $P = \lfloor M^{1-1/k} N^{1/k} \rfloor$, integer $k \geq 2$, and $2 \leq M \leq N$,

$$\mathrm{Merge}_P(M,N) \leq k \left\lceil \frac{\lg \lg M}{\lg k} + 1 \right\rceil$$

$$\leq \frac{k}{\lg k} \lg \lg M + k + 1 .$$

Summary of proof. The proof proceeds inductively, by showing, given $\lfloor M^{1-1/k} N^{1/k} \rfloor$ processors, how we can in k comparison steps reduce the problem of merging two lists of length M and N to the problem of merging a number of pairs of lists, where each pair's shorter list has length less than $M^{1/k}$. The pairs of lists are so created that we can distribute the $\lfloor M^{1-1/k} N^{1/k} \rfloor$ processors amongst them in such a way as to ensure that for each pair there will be enough processors allocated to satisfy the induction hypothesis.

The above theorem is a generalization of Valiant's Theorem 3. Valiant's result is the special case of k=2, whereas the formula is minimized when k=3:

Corollary 4. For $P = \lfloor M^{2/3} N^{1/3} \rfloor$ and $2 \leq M \leq N$,

$$\mathrm{Merge}_P(M,N) \leq 3 \left\lceil \frac{\lg \lg M}{\lg 3} + 1 \right\rceil$$

$$\leq \frac{3}{\lg 3} \lg \lg M + 4$$

$$\cong 1.893 \lg \lg M + 4 .$$

The following corollary is very similar to Valiant's Corollary 5. However, besides having the obviously better constant factor for k=3, the algorithm is slightly more natural and has a smaller additive constant. The proof is very similar to the proof of Theorem 2.

Corollary 5. For $P = \lfloor r M^{1-1/k} N^{1/k} \rfloor$ and $2 \leq r \leq M \leq N$,

$$\mathrm{Merge}_P(M,N) \leq \frac{k}{\lg k} (\lg \lg M - \lg \lg r) + k + 1 .$$

Corollary 6. For $2 \leq P \leq M+N$,

$$\mathrm{Merge}_P(M,N) \leq \frac{M+N}{P} + \lg\left(\frac{M+N}{P}\right) + \frac{3}{\lg 3} \lg \lg P + 6 .$$

Corollaries 5 and 6 together define a merging algorithm which, for M=N and all P, is optimal up to a constant factor; this optimality is a consequence of the lower bound for merging given in [1] and the fact that no parallel algorithm can be more than P times faster than its sequential counterpart.

Basically, all of these algorithms can be implemented on an CREW P-RAM in time equal in order to their number of comparison steps -- see [1].

Sorting

The merging algorithm of Corollary 6 allows us to obtain fast sorting algorithms by using an idea of Preparata [5]. In general, our sorting algorithms are enumeration sorts, i.e. the rank of an element is determined by counting the number of smaller elements.

We present here the general idea of the sorting algorithm under the simplifying assumption

that all variables are continuous. Let G be some constant (dependent on P and N). The algorithm works as follows:

If N = 1 the list is sorted, while if P = 1 sort in $Sort_1(N)$ comparison steps using the best sequential sorting algorithm [it is well known that $Sort_1(N) = N \lg N - O(N)$]. Otherwise apply the following procedure:

(1) Split the processors into G groups with $P/G \geq 1$ processors in each, and split the elements into G groups with $N/P \geq 1$ elements in each.

(2) Recursively sort each group independently in parallel.

(3) Merge every sorted list with every other sorted list.

(4) Sort the entire list by taking the rank of each element to be the sum of its ranks in each merged list it appears in, less G-2 times its rank in its own list.

Noting that step (4) requires $\binom{G}{2} = \frac{G(G-1)}{2}$ independent merges, we are led to the following recurrence relation for the time $S_P(N)$ it takes this algorithm to sort N elements with P > 1 processors.

$$S_P(N) = S_{\frac{P}{G}}\left(\frac{N}{G}\right) + \text{Merge}_{\frac{2P}{G(G-1)}}\left(\frac{N}{G},\frac{N}{G}\right)$$

$$\leq S_{\frac{P}{G}}\left(\frac{N}{G}\right) + \frac{N(G-1)}{P} + \lg\left(\frac{N(G-1)}{P}\right) + \frac{3}{\lg 3}\lg\lg\left(\frac{2P}{G(G-1)}\right) + O(1) .$$

Let M be the minimum of P and N. Then not counting the sequential sorting (after the final recursive call), the above algorithm requires approximately

$$\frac{\lg M}{\lg G}\cdot\left(\frac{N(G-1)}{P} + \lg\left(\frac{N(G-1)}{P}\right) + \lg\lg P\right) ,$$

comparisons. This is minimized for

$$G \cong \max\left(\frac{3}{\ln 3}\frac{P \lg\lg P}{N \lg G}, 2\right) . \qquad (*)$$

In [4] we show that this algorithm can be made rigorous for P, N, and G all integers. Furthermore, we show that these algorithms can be implemented on a CREW P-RAM. This yields the following specific results.

When N=P, $G = \frac{3}{\ln 3}\frac{\lg\lg N}{\lg\lg\lg N}$ by equation (*), so

$$Sort_N(N) \leq 1.893\frac{\lg N \lg\lg N}{\lg\lg\lg N}\cdot\left(1 + O\left(\frac{\lg\lg\lg\lg N}{\lg\lg\lg N}\right)\right) .$$

This is an improvement on the $2 \lg N \lg\lg N$ obtained by Valiant.

For G=2, which is the optimal choice for G when $N \geq \frac{3}{2\ln 3} P \lg\lg P$,

$$Sort_P(N) \leq \frac{N \lg N}{P} + \frac{3}{\lg 3}\lg P \lg\lg P + O\left(\frac{N}{P} + \lg P \lg\frac{2N}{P}\right) .$$

Note that for G=2 the algorithm is a pure comparison sort, i.e. it is no longer an enumeration sort.

Finally, if $P = N(\lg N)^{1/k}$ then

$$Sort_P(N) \leq 1.893 k \lg N + o(k \lg N) .$$

This in an improvement on Hirschberg [3] which showed that $N^{1+1/k}$ processors can sort in $O(k \lg N)$ time, and a generalization of Preparata [5] which showed that $N \lg N$ processors can sort in $O(\lg N)$ time.

References

[1] A. Borodin and J. E. Hopcroft, "Routing, Merging and Sorting on Parallel Models of Computation" Proc. of ACM 14th Ann. Symp. on Theory of Computing, (May 1982), pp. 338-344.

[2] F. Gavril, "Merging with Parallel Processors", CACM, (Oct. 1975), pp. 588-591.

[3] D. S. Hirschberg, "Fast Parallel Sorting Algorithms", CACM, (Aug. 1978), pp. 657-661.

[4] Clyde P. Kruskal, "Searching, Merging, and Sorting on Parallel Models of Computation", manuscript, (Apr. 1982).

[5] Franco P. Preparata, "New Parallel-Sorting Schemes", IEEE Trans. on Computers, (July 1978), pp. 669-673.

[6] Yossi Shiloach and Uzi Vishkin, "Finding the Maximum, Merging and Sorting in a Parallel Computation Model", J. of Algorithms, (Mar. 1981), pp. 88-102.

[7] Leslie G. Valiant, "Parallelism in Comparison Problems", SIAM J. on Computing, (Sept. 1975), pp. 348-355.

ON COMPUTING WEAK TRANSITIVE CLOSURE IN O(LOG N) EXPECTED RANDOM PARALLEL TIME

Albert G. Greenberg
University of Washington
Seattle, Washington

Michael J. Fischer
Yale University
New Haven, Connecticut

Abstract -- We consider the time needed to compute the weak transitive closure of a Boolean matrix, with respect to a probabilistic model of unrestricted parallelism. Our principal result is an algorithm that computes the weak transitive closure of any n x n Boolean matrix in expected time O(log n) and in time $O((\log n)^2)$ in the worst case. Thus, the weakly connected components of any directed graph on n nodes, or the connected components of any undirected graph on n nodes, can be computed within these bounds.

1. INTRODUCTION

We define weak transitive closure in terms of reflexive and transitive closure. The reflexive and transitive closure of an n x n Boolean matrix A is the Boolean matrix $A^* = (I \vee A)^{n-1}$, where v denotes coordinate-wise disjunction. The weak transitive closure of A is the Boolean matrix $\bar{A} = (A \vee A^t)^*$, where A^t denotes the transpose of A. We can regard A as the adjacency matrix of a directed graph and $\bar{A}$ as a presentation of the graph's weakly connected components. Our interest is in the parallel time needed to compute weak transitive closure.

We use a probabilistic version of the Parallel Random Access Machine (P-RAM), a model of a synchronous parallel computer introduced by Fortune and Wylie [3]. An advantage of the model is that its power can be related to the power of a probabilistic Turing machine. A formalization of this relation and a recent result of Aleliunas et al. [1] lead to an algorithm that, given any symmetric n x n Boolean matrix A, computes A^* with probability of error $\leq 1/2$, in O(log n) time. We combine this algorithm with a deterministic algorithm for reflexive and transitive closure to obtain our main result: an algorithm that, given any n x n Boolean matrix A,

- computes $\bar{A}$ with no chance of error,
- runs in expected time O(log n), worst case time $O((\log n)^2)$, and
- uses $n^{O(1)}$ processors.

A more refined analysis than that given below indicates that $O(n^5 \log n)$ processors suffice.

Obviously, if $\bar{A} = A^*$, then our algorithm computes reflexive and transitive closure. Equality holds, for instance, if A is "Eulerian". An n x n Boolean matrix is Eulerian if the number of ones in its i-th row equals the number of ones in its i-th column ($1 \leq i \leq n$). In this case, we can regard A as the adjacency matrix of a

This material is based upon work supported by the National Science Foundation under Grants MCS80-03337 and MCS81-16678, and the Office of Naval Research under Contracts N00014-80-C-0221 and N00014-82-K-0154.

0190-3918/82/0000/0199$00.75 © 1982 IEEE

directed graph in which each strongly connected component is an Eulerian digraph [2]. The observation that $\overline{A} = A^*$ if A is a symmetric Boolean matrix has bearing on the following problem posed by Hirschberg et al. [5].

Let A be an n x n symmetric Boolean matrix. We can regard A as the adjacency matrix of an undirected graph G on nodes 1, 2, ..., n. As defined in [5], the problem of computing the connected components of G is to compute an n-vector c, where c[i] = min{j: $1 \leq j \leq n$, and j belongs to the same connected component as i in G}, for $1 \leq i \leq n$. Hirschberg et al. presented a deterministic algorithm, with respect to a parallel model similar to ours, that computes c in $O((\log n)^2)$ time in the worst case, using $O(n^2/\log n)$ processors.

Notice that c can be obtained from $A^*=(a^*_{i,j})$ by computing c[i] = min{j: $1 \leq j \leq n$, $a^*_{i,j}=1$}, for each i ($1 \leq i \leq n$). The min computations can be carried out in O(log n) time using n^2 processors. As $\overline{A} = A^*$, our results provide an alternative method for computing c that costs O(log n) expected time, $O((\log n)^2)$ worst case time, and $n^{O(1)}$ processors.

Some discussion is in order on how our results contrast with recent (independent) results of Reif [7]. Let us review a crucial result of Aleliunas et al. first. In [1], Aleliunas et al. gave an O(log n) space-bounded, $n^{O(1)}$ time-bounded sequential algorithm to decide undirected graph reachability; that is, to decide if two distinguished nodes in an n node undirected graph belong to the same connected component. The Aleliunas algorithm is probabilistic, with "one-sided" error probability: If the two nodes belong to the same component, then the algorithm decides correctly with probability $> 1/2$; otherwise, the algorithm always decides correctly. The chance of error can be eliminated, without change in the space or time bounds, if we allow the algorithm to be non-uniform in n (Cf. [7]).

Reif's work adapts the Aleliunas algorithm to run in O(log n) parallel time, using $n^{O(1)}$ processors, with one-sided error probability [7]. (The result of Theorem 2 below is similar.) Again, at the cost of a non-uniform construction, the chance of error can be eliminated, without change in the complexity bounds. Although our work is also based on the Aleliunas algorithm, we get a different result: graph reachability can be decided for every pair of nodes in an undirected graph, without error and without resort to a non-uniform construction, in O(log n) time on average, using $n^{O(1)}$ processors.

Reif defined a complexity class, Σ_* CSYMLOG, in terms of O(log n) space-bounded symmetric Turing machines [6], and established that every language in Σ_* CSYMLOG is recognizable in O(log n) time, using $n^{O(1)}$ processors, with probability of error $< p$, where p is any constant, $0 < p < 1$. (Again, a non-uniform construction can be used to eliminate the chance of error.) His analysis can be strengthened using our main result, to establish that every language in Σ_* CSYMLOG is recognizable, with no chance of error, in O(log n) expected time, using $n^{O(1)}$ processors. Reif showed that Σ_* CSYMLOG contains a number of interesting problems: invalidity testing of formulas in a restricted quantified Boolean logic, recognition of edges in a minimum spanning forest, recognition of k-connected vertex pairs in an undirected graph, and several graph recognition

problems.

2. PROBABILISTIC PARALLEL RANDOM ACCESS MACHINES

We begin with an informal description of a deterministic P-RAM [3]. An infinite number of processors P_0, P_1,... are available, with each having a local accumulator, a local program counter, and an infinite local memory. The processors share access to two infinite global register sets: the read-only input registers $In_1, In_2, \ldots$ and the work registers $W_1, W_2, \ldots$. A single program controls execution. In this setting, a program is a finite list of possibly labeled instructions. An instruction is of one of the following forms.

```
AC := x
x := AC
AC := AC + x
AC := AC - x
goto L
if AC=0 then goto L
fork L
HALT
```

AC refers to the accumulator of the processor that executes the instruction, x is an address, an indirect address, or a constant, and L is a label. One restriction applies to the use of global memory. At each step, at most one processor can write to a register. On the other hand, any number of processors can read the same register simultaneously.

Initially, a single processor P_0 is active. If at some step t, a processor P_i executes an instruction of the form "fork L", then at step t+1 a new processor P_j begins execution at the instruction labeled L, with the accumulator of P_j set to the value in the accumulator of P_i at step t. Hence in t steps up to 2^t processors can be activated. By convention, for $i < j$, if P_i is activated at step t_1 and P_j at t_2, then $t_1 \leq t_2$. The P-RAM terminates when P_0 executes a HALT instruction.

As defined in [3], a deterministic P-RAM computes a function from $\{0,1\}^*$ to $\{0,1\}$. We make a small change to allow for the computation of functions from $\{0,1\}^*$ to $\{0,1\}^*$. We add an infinite global set of write-only output registers Out_1, Out_2,..., and fix input/output conventions as follows. An input $x=x_1x_2\ldots x_s$ of s bits is presented one bit per register in $In_1, In_2, \ldots, In_s$, and s is presented in processor P_0's accumulator. All other registers have value 0. We say the machine outputs $y=y_1y_2\ldots y_t$ of t bits, iff when P_0 halts

1. $t=\max\{a,0\}$ where a is the number in P_0's accumulator, and
2. $y_i=0$ iff $Out_i=0$, for $1 \leq i \leq t$.

In a deterministic P-RAM, no two instructions can have the same label. In a probabilistic P-RAM, a given label can be associated with at most two instructions. If two instructions I_1 and I_2 have the same label L and a processor P_i executes a jump to L, then with probability 1/2 P_i jumps to I_1; otherwise (with probability 1/2) P_i jumps to I_2 (independently of any other step of P_i or any other processor).

A probabilistic P-RAM computes a random function [8], defined as follows. A random function F from X to Y is characterized by a function p_F: X x Y --> [0,1] such that, for all x in X, $\sum_{y \text{ in } Y} p_F(x,y) \leq 1$. The expression F(x)=y means F applied to x equals y with probability $p_F(x,y)$. A probabilistic P-RAM Q is said to compute F iff $X=Y=\{0,1\}^*$ and on input x, Q outputs y with probability $p_F(x,y)$, for all x in X and y

in Y. Also, Q does not terminate with probability $1 - \sum_{y \text{ in } Y} p_F(x,y)$, for all x in X. We say Q runs in expected time T(n) (in time T(n)) iff, on every input of n bits, Q terminates in expected time $\leq$ T(n) (Q always terminates in $\leq$ T(n) time).

A probabilistic Turing machine also computes a random function from $\{0,1\}^*$ to $\{0,1\}^*$ [4, 8]. Fortune and Wylie have shown that with respect to computing 0-1 functions, time on a deterministic P-RAM is at least as powerful as space on a deterministic Turing machine. A small modification of their proof gives us a useful connection between probabilistic Turing machines and probabilistic P-RAM's.

Lemma 1: Let F be any random function computable on an S(n) space-bounded, T(n) time-bounded multitape probabilistic Turing machine, where log T(n) = O(S(n)) and S(n) $\geq$ log n. Then there is a probabilistic P-RAM that computes F in time O(S(n)).

Proof: (Outline)

Consider a probabilistic Turing machine M that computes F in S(n) space and T(n) time. We may assume that M never enters the same configuration twice, as without more than a constant factor loss in time or space M can maintain a "clock" on a work tape. Fix an input x of n bits. M can assume up to $2^{d*S(n)}$ configurations on x, where d is a constant.

We adapt a technique of [3] to construct a probabilistic P-RAM P that simulates M on x so as to compute F(x) in O(S(n)) time. Suppose that the function S(n) is itself computable in O(S(n)) time. P forks $2^{d*S(n)}$ processors, giving a processor P_i for each configuration c_i of M on x. P_i chooses a successor c_j of c_i uniformly at random from among the possible successors of c_i and writes j into global register W_i, for all i ($1 \leq i \leq 2^{d*S(n)}$). In effect, this constructs a graph in which a given node $\hat{c}_i$ corresponds to configuration c_i. At most one edge emanates out of $\hat{c}_i$, to $\hat{c}_j$, where j is the value of W_i.

The graph contains a directed path leading out of the node corresponding to the start configuration, which describes an execution of M on input x. P uses global memory to mark each node on this path. As M runs in S(n) space, the last node on the path must correspond to a final configuration. Lastly, the output registers are written in accordance with the marked nodes. The essential details involved in implementing these steps in O(S(n)) time can be found in [3].

To carry the proof through without assuming that S(n) is computable in O(S(n)) time, we use the usual trick of trying the simulation for S(n) = 2,4,8,... until we reach a trial value large enough so that M reaches a final state in the simulation. In performing each trial, if any successor of c_i requires more than the currently allowed amount of space, we leave W_i undefined, and if W_i is defined from a previous trial, we do not change it. The latter condition is necessary to maintain the right probabilistic behavior. Otherwise the simulation would be biased towards choices that lead to rapid halting.

□

3. THE ALGORITHM FOR WEAK TRANSITIVE CLOSURE

We begin with a theorem describing a random function which we refer to as random closure. For any symmetric Boolean matrix A, the random closure of A equals the reflexive and transitive closure of A with probability $\geq 1/2$. We will exploit this and other properties of random closure to arrive at an efficient algorithm for weak transitive closure.

One more notation is of use. We write $A \leq B$ to signify that the Boolean matrices A and B satisfy $B = B \vee A$.

Theorem 2: There is a probabilistic P-RAM P that computes a random function R from symmetric Boolean matrices to Boolean matrices such that for every n x n symmetric Boolean matrix A:

1. P computes R(A) in time O(log n).
2. If $p_R(A,B) \neq 0$, then B is an n x n Boolean matrix with $A \leq B \leq A^*$.
3. $p_R(A,A^*) \geq 1/2$.

We refer to R(A) as the random closure of A.

The second property implies that $B^* = A^*$. The third indicates that R(A) is likely to equal A^*.

Proof: By lemma 1, it suffices to give an O(log n) space-bounded, $n^{O(1)}$ time-bounded probabilistic Turing machine that computes a random function satisfying the second and third properties. We appeal to a recent result involving random walks in graphs, defined as follows. A random walk in an undirected graph G starts at an arbitrary node in G. At each step beginning at a node v, we choose an edge uniformly at random from the edges emanating out of v and traverse it. Now let G be an undirected graph having e edges. Also let i and j be any two nodes in G, and let $d_{i,j}$ be the length of the shortest path from i to j. The analysis of Aleliunas et al. shows that the expected number of steps in a random walk in G starting at i before reaching j is at most $2d_{i,j}e$ [1].

Let A be any n x n symmetric Boolean matrix and G the graph (having n nodes and e edges) corresponding to A. Using the methods in [1], we can construct an O(log n) space-bounded, $n^{O(1)}$ time-bounded probabilistic Turing machine M which on input i,j,A simulates step by step a random walk of $4(n-1)e$ steps starting at node i in G ($1 \leq i,j \leq n$). If the walk reaches j then M outputs 1; otherwise M outputs 0. Hence if i and j are in the same connected component of G, M outputs 1 with probability $\geq 1/2$. M always outputs 0 otherwise.

To complete the proof, for $1 \leq i,j \leq n$, compute $b_{i,j}$ as the disjunction of $a_{i,j}$ and the $\lceil \log_2(2n^2) \rceil$ results of running M $\lceil \log_2(2n^2) \rceil$ times, on input i,j,A. The computation requires O(log n) space and $n^{O(1)}$ time. $A \leq B$ follows by construction. Also, $B \leq A^*$ holds, since $b_{i,j}=1$ is possible only if j is reachable from i in the graph corresponding to A. Finally, it can be verified that $B = A^*$ with probability $\geq (1 - 1/2^{\lceil \log_2(2n^2) \rceil})^{n^2} \geq (1 - 1/(2n^2))^{n^2} \geq 1/2$. □

We note that a similar argument can be used to obtain a random function that maps Eulerian matrices to Boolean matrices and satisfies the three properties given in the theorem.

Now, we give our main result.

Theorem 3: There is a probabilistic P-RAM that, on every n x n Boolean matrix A, computes $\overline{A}$ without error, runs in expected time O(log n), in time $O((\log n)^2)$ in the worst case, and uses $n^{O(1)}$ processors.

Proof: The desired P-RAM executes the following algorithm. R denotes random closure (Cf. Theorem 2).

```
S := I v (A v A^t);
repeat
  T := R(S);
  S := (T v T^t)^2
until S = T.
```

At the bottom of the repeat loop we have $I \vee (A \vee A^t) \leq T \leq (A \vee A^t)^* = \overline{A}$. When the algorithm halts $T=(T \vee T^t)^2$, so it follows that $T = \overline{A}$ holds. Each iteration of the loop costs O(log n) time using $n^{O(1)}$ processors. Because of the squaring, the loop will be repeated at most O(log n) times, giving the desired worst case time bound.

To complete the proof it suffices to show that the expected number of iterations of the loop is O(1). By theorem 2, we have a sequence of trials, each with probability of success $\geq 1/2$, ending on the first success. The expected length of such a sequence is ≤ 2.

□

Acknowledgement

We are indebted to Richard Ladner for bringing reference [1] to our attention early in this research, and to Martin Tompa for helpful comments on preliminary versions of this paper.

REFERENCES

1. Aleliunas, R., Karp, R.M., Lipton, R.J., Lovász, L., and Rackoff, C., Random Walks, Universal Traversal Sequences, and the Complexity of Maze Problems, 20th Annual Symposium on Foundations of Computer Science (October 1979), 218-223.

2. Anderson, S., "Graph Theory and Finite Combinatorics", Markham Publishing Company, Chicago, 1970.

3. Fortune, S. and Wylie, J., Parallelism in Random Access Machines, Proceedings of the Tenth Annual ACM Symposium on Theory of Computing (May 1978), 114-118.

4. Gill, J., Computational Complexity of Probabilistic Turing Machines, SIAM Journal on Computing 6, 4 (December 1977), 675-695.

5. Hirschberg, D.S., Chandra, A.K., and Sarwate, D.V., Computing Connected Components on Parallel Computers, Communications of the ACM 22 (1979), 461-464.

6. Lewis, H. and Papidimitriou, C., Symmetric Space-Bounded Computation, Technical Report TR-08-80, Harvard University, Aiken Computation Laboratory, August, 1980.

7. Reif, J., Symmetric Complementation, Proceedings of the Fourteenth Annual Symposium on Theory of Computing, San Francisco, CA (May 1982), 201-214.

8. Santos, W.J., Probabilistic Turing Machines and Computability, Proceedings American Mathematical Society 22 (1969), 704-710.

ALTERNATIVE APPROACHES TO MULTIPROCESSOR GARBAGE COLLECTION

Newman I.A., Woodward M.C.
Department of Computer Studies,
Loughborough University of Technology,
Loughborough, Leicestershire, U.K.

Abstract -- This paper considers the problem of performing garbage collection in a list processing system in parallel with list updating. One previously published method for searching an existing list system and marking all reachable nodes is outlined and a new method for performing the same function is described. The two methods are then compared with respect to both the number of nodes that need to be visited to complete a marking phase, and the number of synchronisations that are needed when there is more than one marker working in parallel. A number of different list structures are postulated, and results are presented of the predicted performance of the two algorithms.

Introduction

Within list processing systems, nodes are repeatedly added to and removed from a number of lists. The storage locations in the memory space available to the list processing system tend to be allocated for use in a particular list and then freed. It is clearly desirable to reclaim these freed cells for subsequent use, and there are a number of techniques whereby this may be accomplished. The technique that is considered in this report is Garbage Collection which was first proposed by McCarthy [3] and used in the LISP 1.5 system [4].

Using this technique the problem of storage reclamation is (often) ignored until the list of available cells (free list) becomes empty. When this arises, the list processing is temporarily suspended and a garbage collection process locates cells which have become free and adds them to the free list.

The basic garbage collection algorithm falls into four phases:-

1) Marking phase — in which all accessible nodes are marked.

2) Relocate phase — in which all accessible nodes are compacted into a single contiguous area.

3) Update phase — in which all pointers to relocated nodes are changed.

4) Reclaim phase — in which the inaccessible cells are collected to form the new free list.

A perfectly satisfactory garbage collection scheme need only consist of phases 1 and 4 and it is this scheme that will be considered further in the remainder of the paper.

Steele [6] suggested that garbage collection could be performed in parallel with list processing using two processors, one garbage collecting and one performing the processing operations. Under these conditions the user(s) would be spared the delay which would otherwise occur when the free list becomes empty. A workable solution to this problem, which prevented interference between the two processors was developed by Dijkstra et al [1]. This solution was extended to incorporate multiple list processors (mutators) and multiple garbage collectors by Lamport [2]. Lamport pointed out that interference between the mutators and the garbage collectors was potentially high in the marking phase (phase 1) but that the reclaim phase (phase 4) should involve negligable interference since the nodes being reclaimed cannot, by definition, be accessed by any mutator. If all the reclaimed nodes are gathered into an independent list then the only possible interaction occurs when this list is added to the free list and this is a single operation.

In this paper, therefore, the reclaim phase will be ignored and a new algorithm for marking reachable nodes will be developed and compared with that devised by Lamport.

Firstly, the terminology will be introduced, and the algorithm adopted by Lamport will be outlined. The new solution is then presented together with results showing the performance of the two algorithms.

Definition of Terminology

The list structure to which consideration will be given consists of a collection of list cells (nodes). Each node consists of some (possibly no) data fields and an ordered sequence of pointers to other nodes (edges). The node from which an edge emanates will be called its source and that to which it points the destination. Some of the edges are distinguishable as null edges, that is the edge does not connect two nodes but acts as a terminator.

If an edge connecting nodes A and B exists and B is the destination of the edge then B is (one of) the successors of node A and A is a predecessor of B. Nodes having no successors are called terminal nodes (or terminals).

0190-3918/82/0000/0205$00.75 © 1982 IEEE

Some of the nodes, known as root nodes, are fixed. A node is said to be reachable (or accessible) if there is a path to it from a root via reachable nodes. A non-reachable node is called a garbage node.

Lamport's Algorithm

Lamport introduces an extra field into the nodes for use during the marking phase. This field is intended to hold a colour which may be one of black, grey or white, and indicates at which of the stages of the marking phase the node is.

Operations are introducted to change the colour of a node to a specific value. Also introduced is a shading operation which changes a white node to grey but leaves other colours unchanged. These operations on a node are required to be indivisible with respect to the list processing system (i.e. they must be point operations). The node space is divided into several (not necessarily disjoint) subsets. A marking process (marker) is assigned to each of the subsets. No details are given as to the method of division, so a physical division seems simplest. Initially, all nodes are marked white.

The operation of the marking algorithm commences with the roots being shaded. Each marker then searches its subset of nodes. When a grey node is located by any one of the markers all the successors of that node are shaded and the original node is coloured black. All the markers are then requested to restart the search of their portion of the node space. The marking terminates when no grey nodes exist, i.e. all reachable nodes have been coloured black. The garbage (unreachable) nodes are those that remain white.

Several comments may be made upon this algorithm. Firstly, no attempt is made to use the structure of the list within the algorithm itself. All reachable nodes may be located by chaining down the list structure from the roots. This leads to a second point, that all the garbage nodes will have to be inspected, possibly several (and in some cases many) times. This time is, of necessity, "wasted" since a garbage node, by definition, cannot become grey. This is an inevitable consequence of dividing the node space into physical subsets.

Further, the synchronisation between the markers is non-trivial. The need for one marker, on discovering a grey node and shading its successors, to cause all others to restart the search of their subspace requires a "communication path" between every pair of markers. Also, when a marker completes the search of its subspace, no guarantee can be given that it has completed its work as another marker may later discover a grey node. Only when all markers have completed searching their own subspaces can the marking process terminate. This requires each marker to monitor the state of all other markers in some way.

Irrespective of the method that is used to implement the intercommunication there is bound to be a considerable waste of processor time. This may be caused by unnecessary searching of the list structure, by waiting to be informed whether to restart the search or by both of these eventualities. If a marker pre-empts all the others, forcing them to restart their searches as soon as it has marked one node, then the uncompleted searches are wasted. If on the other hand all markers are allowed to finish their search then any marker finding nothing has been wasting time. In either case it is necessary to have some way for markers to indicate whether they have completed or not so that it is possible to determine the end of the marking phase. If messages between the processors are used then every marker must send a message to every other marker when it has finished a search without finding a shaded node. Although the number of messages could be kept to a minimum some are bound to be sent unnecessarily. The alternative would be to use one shared location to record the state of each processor, in which case processors would need to loop inspecting the value of the locations for the other machines once they had finished an unsuccessful search. Any such looping would, of course, represent wasted processor time.

Chaining Algorithm

The algorithm described above was based on a physical sub-division of the node space. An alternative algorithm is described below which marks the reachable nodes by searching down the list structure and has hence been given the name Chaining Algorithm.

In order to partition the list space, and thus enable several markers to operate, the concept of a sublist is introduced. Each marker is allocated a section of the total list structure and marks the nodes contained in this sublist. Once a marker has a sublist, it may proceed independently of the other markers (thus reducing the synchronisation overheads). However, to enable marking to be equitably distributed between the markers an additional list is introduced.

This list, the subroot list, holds the roots of unmarked sublists. Initially, it contains the roots of the whole structure. The list can be kept short, with possibly one entry for each marker since this list represents work yet to be allocated to a marker. The colour yellow is introduced for a node contained within the subroot list, so the roots of the list structure are initially coloured yellow. Also, the term "uncoloured" is introduced for a node which is either white or grey.

When a marker is started, or whenever it has completed the marking of a sublist, it removes a node from the subroot list to discover the section of the list it is to process. This node is shaded. The marker then refills the

subroot list by adding the uncoloured successors of the subroot it has obtained to the list until either the list is filled or only one uncoloured successor remains. The nodes added to the subroot list are coloured yellow. At all stages in the remainder of the algorithm yellow nodes are treated as black when encountered by a marker since the nodes following are guaranteed to be marked at a later stage.

The remainder of the algorithm, shown in outline in Figure 1, is as follows.

```
marker =
       begin
         while  subroot list is not empty do
                remove node from subroot list;
                shade node;
                refill subroot list;
                while  subroot is not black do
                  while  number of uncoloured
                           successors = 1 do
                         shade successor;
                         colour node black;
                         advance to successor
                           setting as subroot
                  od;
                  while  number of uncoloured
                           successors  >0 do
                         choose one successor;
                         shade sucessor;
                         advance to successor
                  od;
                  colour current black;
                  current:=subroot
                od
         od
       end;
```

FIGURE 1: Algorithm for a Marker

The marker maintains two pointers to the sublist it is processing, one to the subroot and one to the node it is currently inspecting. Both of these initially point to the root of the sublist. If only one uncoloured successor of the current node exists then the node is shaded, the current node is coloured black and both the subroot and current pointers are advanced to the successor. This process is repeated until a node with several or no uncoloured sucessors is met. If the current node has some uncoloured successors then one is chosen. It is shaded and the current pointer is advanced to it. This shading and advancing is repeated until the current node has no uncoloured successors. When this situation arises, the current node is coloured black and the current pointer is set to the subroot. The whole of this procedure is then repeated until the subroot is coloured black. When that occurs the sublist for which the marker was responsible has been marked and a new root is chosen from the subroot list. The marker terminates when it cannot obtain a node from the subroot list.

With a simply connected list structure (that is one containing no closed loops and no interconnection between sublists), the algorithm is guaranteed to be correct and to terminate. The list structure appears as many independent lists each with its own marker. Furthermore, the only synchronisation required between the markers is when accessing the subroot list. The synchronisation overhead may be kept to a minimum by allowing one marker to be filling the list independently of markers which are removing nodes from the list. A marker which attempts to remove a node may still have to wait either for another marker removing a node or if the list is apparently empty because it is in the process of being refilled. The markers can, however, be prevented from interfering with one another during the refilling stage if, when one marker is attempting to refill the subroot list then further markers are allowed to by-pass the refilling stage of the algorithm.

If the list structure is not simply connected but the sublists have common nodes (but still without loops) then consideration must be given to the possible events at the intersection points. The simplest possibility to consider is that one marker colours the common node yellow or black before any other marker accesses that node. When another marker reaches that node, it will proceed no further. If the intersection node is white or grey then the structure beyond the node needs to be inspected and several markers may attempt to colour the sublist. This will have the same effect as several passes down the branch by a single marker, that is, the several markers will jointly colour the nodes below the intersection point.

If two markers attempt to update the colour of the intersection node simultaneously, then one must complete its update after the other. The node then becomes that colour. Whichever colour is finally given to the node, it is valid for at least one of the markers, and this marker will complete the colouring.

However, with the algorithm as described, a list structure containing cycles (closed loops between edges) may cause a marker to permanently loop. To overcome this, some intelligence may be given to the markers. If, while chaining down through the successors, the marker visits an excessive number (e.g. more than the maximum height of the structure or more than the total number) of nodes without reaching a terminal (or a yellow or black node), then it may assume that a loop exists and arbitrarily colour the current node yellow and add it to the subroot list. In

this way, a terminating condition is placed within the loop. Loops will then only reduce the efficiency of the algorithm due to wastage in identifying them.

Comparison of the Marking Algorithms

Empirical testing of the algorithms was carried out using a simulated multiprocessing system. The algorithms were used on a number of types of list structure. Four types of structure were chosen to exercise the algorithms under a variety of conditions. The types were:-

a) Linear List

b) Curtain

This structure consists of many linear lists emanating from a single root.

c) Highly Interconnected

In this structure, each node has many branches with a large number of nodes being shared between sublists. Two versions of each structure were generated, the second being the mirror image of the first, that is the sublists that were placed left to right for a node in one version were placed right to left in the other.

d) Random

The interconnection was generated randomly.

Each of the first threee list structures were used with both a high and a low proportion of the node space consisting of reachable nodes. All structures were loop free. Lamport's algorithm was performed twice, once with the markers searching from the low addresses to high addresses and secondly from high addresses to low. Table 1 shows some of the results obtained from the simulation studies when the node space consisted of 100 nodes.

From the table it can be seen that, with one exception, the Chaining Algorithm performs better than that of Lamport on each of the values tabulated. In most cases, the number of nodes visited is vastly reduced (often by a factor of 50 or more). Also the costs of synchronisation between the markers is reduced. The overall improvement obtained from the Chaining Algorithm can be observed from the elapsed times given in the table.

The structure with which the Chaining Algorithm performs least well is one with high interconnectivity. Yet even with this structure the synchronisation overheads are minimal. This is of great advantage since a synchronisation will (in general) be much more expensive than a node visit. The first highly interconnected structure represents a 'worst' case for the Chaining Algorithm as implemented. In order for the blackening of the nodes from the terminal nodes towards the subroots to take place, the sublists need to be traversed many times. This is partly due to the high interconnection which will yield a high degree of overlapping sublists and partly due to the greater number of successors which each node has. The pathological nature of the structure can be seen in the fact that the image structure is traversed at about one third of the cost.

The effect of synchronisations is not fully revealed in the elapsed times recorded in Table 1, since a synchronisation is approximately as expensive as a node visit in the simulation program, whereas it would probably be substantially more expensive in practice. Furthermore, in the results for Lamport's algorithm a synchronisation does not cause waiting which it would do for some processors in some cases in practice. The very much higher level of synchronisation in Lamport's algorithm for most list structures could therefore be assumed to result in a further time advantage for the Chaining Algorithm in practice.

One possible improvement to the Chaining Algorithm would require the introduction of backward as well as forward pointers in the list. The algorithm could then move back up the sublist from a terminal node marking as it goes. This would save successive searches down the list if there was a large fan out from the sub-node. Unfortunately, this structure would require a much more elaborate algorithm since it would be possible for an ascending marker to be unable to find the subnode from which it started due to the action of mutators changing the sub-tree. Explicit synchronisation between markers and mutators might now be required and the possibility of processing a section of the tree which has been rendered garbage is also highlighted.

Conclusions

The chaining algorithm appears to provide a substantially faster multiprocessor garbage collection system with fewer synchronisations than was available previously. This contention is being tested in practice by the implementation of a simple list processor system on a four processor machine with shared memory [5]. The results obtained do confirm the predictions.

Marking Algorithm - Comparison Table

			Chaining Algorithm			Lamport					
						Up			Down		
Type	G. N.	M	Node Vstd	W. P.	Time	Node Vstd	Syn	Time	Node Vstd	Syn	Time
Linear List	0	1	100	0	0:26	5150	100	8:56	5150	100	9:00
Dense	0	5	100	24	1:19	5710	500	9:26	5730	500	9:28
Linear List	95	1	5	0	0:02	305	5	0:32	400	5	0:42
Sparse	95	5	5	24	0:06	125	25	0:13	600	25	1:01
Curtain	0	1	103	0	0:28	5150	100	9:11	5150	100	9:14
Dense	0	5	103	27	0:33	2251	200	3:58	3370	350	5:46
Curtain	84	1	19	0	0:06	908	16	1:37	908	16	1:38
Sparse	84	5	19	27	0:09	959	80	1:40	987	80	1:42
High	1	1	2310	0	6:03	5148	99	9:02	5051	99	8:57
Inter-			1086	0	3:05						
Connect	1	5	4335	42	12:46	4110	400	6:52	3875	400	6:31
Dense			1297	45	4:17						
High	51	1	55	0	0:16	2717	49	4:50	2432	49	4:44
Inter-			250	0	0:54						
Connect	51	5	90	38	0:34	2346	215	4:00	2704	245	4:38
Sparse			105	27	0:36						
Random	40	1	186	0	0:44	4060	60	7:07	2200	60	3:56
One	40	5	189	42	0:55	1465	170	2:32	2612	255	4:28
Random	79	1	84	0	0:18	1893	21	3:19	428	21	0:47
Two	79	5	84	24	0:52	1227	100	2:04	1201	100	2:02
Random	75	1	82	0	0:17	1943	25	3:24	782	25	1:24
Three	75	5	82	24	0:52	1219	105	2:04	1380	125	2:20

Table 1 Simulation Results for Marking Algorithms

KEY

Type The formation of the list structure.
G.N. The number of garbage nodes in the structure.
M The number of markers employed.
Node Vstd The number of nodes visted during the marking phase.
W.P. The number of time steps during which a marker was waiting on the 'listfront' semaphore.
Time The elapsed time (in minutes and seconds) for the simulation of the marking phase.
Syn The number of times, in total, that the markers were restarted at the beginning of their subspace.

Notes on Table 1

1) The simulated time for a multiprocessor solution represents the sum of the times taken by the processors. This is necessarily greater than the simulated time for an optimum uniprocessor solution. The actual elapsed time would be rather more than one fifth of the total time.

2) The simulated version of Lamport's algorithm permitted every marker to finish its search each time. This extra marking on each pass accounts for the lower elapsed time of some multiprocessor trials.

References

[1] E.W. Dijkstra, et al: "On the Fly Garbage Collection: An Exercise in Co-operation," CACM, Vol 21, No 11 (1978), pp 966-975.

[2] L. Lamport, "Garbage Collection with Multiple Processes: An Exercise in Parallelism," Proceedings of the International Conference on 'Parallel Processing', Walden Woods (1976), pp 50-54.

[3] J. McCarthy, "Recursive Functions of Symbolic Expressions and their Computation by Machine," CACM, Vol 3, No 4, (1960), pp 184-195.

[4] J. McCarthy, et al: "LISP 1.5 Programmer's Manual," MIT Press, Cambridge, Mass. (1962).

[5] I.A. Newman, R. Stallard and M.C. Woodward, "An Implementation of a Multiprocessor List Manipulation System," Dept of Computer Studies, Loughborough University of Technology Loughborough, Leics. U.K. (1982).

[6] G.L. Steele, "Multiprocessor Compactifying Garbage Collection," CACM, Vol 18, No 9, (1975), pp 495-508.

CONCURRENT DISK ACCESSING FOR PARTIAL MATCH RETRIEVAL

H.C. Du
Department of Computer Science
University of Minnesota
Minneapolis, Minnesota, MN 55455

Abstract -- Since a file is usually large and can not reside in primary memory, the response time to a query is dominated by the disk access time. In order to reduce the disk access time, and hence the response time, a file can be stored on several independently accessible disks. In this paper, we discuss the problem of allocating buckets in a file among disks such that the maximal disk access concurrency can be achieved. We are particularly concerned with the disk allocation problem for binary Cartesian product files, which have been shown to be effective for partial match retrieval. A heuristic allocation method is first proposed for the cases where the number (m) of available disk units is a power of 2. Then it is extended to fit the cases where m is not a power of 2. The proposed heuristic allocation method has a "near" strict optimal (hence optimal) performance for a partial match query in which the number of unspecified attributes is greater than a small number (5 or 6).

1. Introduction

In an information retrieval system, a basic individual unit of information is defined as a record and a collection of records is called a file. If the number of records in a file is large enough (cannot reside in primary memory), the whole file must be stored on a secondary storage device such as a magnetic disk unit. Therefore, we can also assume that the file is divided into buckets and each time the secondary storage device is accessed, a whole bucket is brought into primary memory.

Since the disk access time is considerably longer than the instruction execution time and primary memory access time, the time taken to respond to a query can be simply measured in terms of distinct disk accesses which must be issued. The number of distinct disk accesses that must be issued is equal to the number of buckets which contain at least one record satisfying the query.

The file design problem for a particular type of queries can, therefore, be stated as follows : Given a file (a set of records), arrange records into buckets in such a way that the average number of buckets to be examined, over all concerned queries, is minimized.

The response time to a given query can be further reduced if the file is stored on several independently accessible disks. Several buckets can be accessed at one disk access time, if they are on different disks and only one bucket can be accessed at a time if they are on the same disk. The response time to a given query in this case is no longer proportional to the total number of buckets needed to be examined, but becomes proportional to the maximum number of buckets needed to be examined on a particular disk. Given a file designed primarily for some type of queries and an m-disk (m>1) system, in order to reduce the average response time, it is necessary to arrange all buckets into m disks in such a way that the maximal possible disk access concurrency is achieved when examining the required buckets.

In this paper, we discuss the problem of allocating all buckets in a file, which is designed for partial match retrieval, to m disks. Particularly we concentrate on the allocation problem for binary Cartesian product files. In section 2, the relations between partial match queries and Cartesian product files are discussed. The existing Disk Modulo (DM) allocation method which has been shown to be effective for Cartesian product files is reviewed in section 3 and its performance is shown to be poor for binary Cartesian product files. In section 4, we first propose a heuristic allocation method for binary Cartesian product files when the number (m) of available disk units is a power of 2. Then it is extended to fit the cases where m is not a power of 2. The performance of the proposed allocation method under various conditions is compared with that of an "ideal" strict optimal and Disk Modulo allocation methods in section 5.

2. Partial Match Queries and Cartesian Product Files

A record may consist of a single attribute or multiple attributes. A k-attribute record r is an ordered k-tuple $(r_1, r_2, \ldots, r_k)$, each $r_i \in D_i$ for $1 \le i \le k$, where D_i is the domain of the i-th attribute. A partial match query q for a k-attribute file is of the form $(A_1 = a_1, A_2 = a_2, \ldots, A_k = a_k)$, where for $1 \le i \le k$, a_i is either a key belonging to D_i, the domain of the i-th attribute, or is unspecified (i.e., a don't care condition), in which case it is denoted by * and where the number of unspecified attributes is j where $1 \le j \le k-1$.

A response to query q is a list of all records in which the i-th attribute is equal to a_i if a_i is not a *. We also say "a record r satisfies a given query q" if and only if record r is in the response list of query q. Recently much attention has been paid to the multi-attribute file design problem for partial match queries ([1]-[7] and [10]-[13]). In this paper we shall also limit ourselves to multi-attribute files.

0190-3918/82/0000/0211$00.75 © 1982 IEEE

In the following we are going to define the Cartesian product file concept which has been shown to be effective for partial match queries [5]. A file F is called a Cartesian product file, if it satisfies the definition in below.

Definition : Let D_i denote the i-th attribute domain of a k-attribute file and let each D_i be partitioned into m_i disjoint subsets $D_{i0}, D_{i1}, ..., D_{i(m_i-1)}$. We call F a Cartesian product file if all records in every bucket are in $D_{1i_1} XD_{2i_2} X...XD_{ki_k}$, where each D_{ij} is one of the subsets $D_{j0}, D_{j1}, ..., D_{j(m_j-1)}$. The bucket b $\subseteq D_{1i_1} XD_{2i_2} X...XD_{ki_k}$ is denoted by $[i_1, i_2, ..., i_k]$.

As an example, let $D_1 = D_2 = \{a,b,c,d\}$, $D_{10} = D_{20} = \{a,b\}$ and $D_{11} = D_{21} = \{c,d\}$. Then the following is a Cartesian product file :
Bucket [0,0]= $D_{10} XD_{20}$ ={(a,a),(a,b), (b,a),(b,b)}
Bucket [0,1]= $D_{10} XD_{21}$ ={(a,c),(a,d), (b,c),(b,d)}
Bucket [1,0]= $D_{11} XD_{20}$ ={(c,a),(c,b), (d,a),(d,b)}
Bucket [1,1]= $D_{11} XD_{21}$ ={(c,c),(c,d), (d,c),(d,d)}

Although in the above example all possible records are included, it should be noticed that a Cartesian product file as defined above is a subset of $D_1 XD_2 X...XD_k$ and does not necessarily contain all possible records. The Cartesian product file concept also satisfies a common property of all "good" file design methods described by Lin, Lee and Du [10] : that is , records in one bucket are similar to one another. The Cartesian product file concept is a simple and natural way to cluster similar records into the same bucket.

Many good file systems such as those designed by Rivest [12], Rothnie and Lozano [13], as well as Liou and Yao [11], are all Cartesian product files. Aho and Ullman [1] explored the file design problem of partial match queries with the assumption that each attribute in a partial match query has a probability of being specified, and their file structure is also a Cartesian product file. The main difference among these file systems is the way each attribute domain D_i is partitioned. In [5] many properties of Cartesian product files were discussed.

3. Disk Modulo Allocation Method

Since Cartesian product files have been shown to be effective for partial match queries and have been widely (though sometimes implicitly) described in the literature, it would be worthy considering the bucket allocation problem for Cartesian product files. Du and Sobolewski [9] proposed a Disk Modulo allocation method for Cartesian product files.

Definition : Let file $F \subseteq D_1 XD_2 X...XD_k$ be a Cartesian product file, where each D_i is partitioned into m_i disjoint subsets $D_{i0}, D_{i1}, ..., D_{i(m_i-1)}$, and m be the number of available disk units (labeled as 0, 1,..., m-1). Let $[i_1, i_2, ..., i_k]$ denote the bucket $F \cap (D_{1i_1} XD_{2i_2} X...XD_{ki_k})$, where $1 \le i_j < m_j$ for $1 \le j \le k$. In the Disk Modulo (DM) allocation method each bucket $[i_1, i_2, ..., i_k]$ in file F is assigned to disk unit $(i_1 + i_2 + ... + i_k)$ mod m.

Given a file with n buckets, there are m^n ways to allocate n buckets into m disks. The problem of finding an optimal allocation method is very difficult. Du and Sobolewski, therefore, compared the Disk Modulo allocation method with an ideal "strict" optimal allocation method.

Definition : An allocation method is said to be strict optimal to a query if a maximum of $\lceil n/m \rceil$ buckets need to be accessed on any one of m independently accessible disks in order to examine the n buckets in response to the query. If an allocation method is strict optimal for all possible queries, it is called a "strict optimal" allocation method.

Note that a strict optimal allocation method is optimal. The Disk Modulo allocation method has been shown to be strict optimal for the following cases [9] :
1) all partial match queries with only one unspecified attribute,
2) all partial match queries with at least one unspecified attribute j for which m_j mod m=o,
3) all possible partial match queries when m_i mod m =0 or m_i =1 for all $1 \le i \le k$,
4) all possible partial match queries when m=2 or 3.

More properties of Disk Modulo allocation method can be found in [9]. Although the Disk Modulo allocation is strict optimal for the above cases, the following example shows that it is not strict optimal (or even optimal) in general. However, Du also proved that the Disk Modulo allocation method is asympototic strict optimal if all m_i's increase to infinity [8].

Example 3.1 : Table 3.1 shows the distribution of assigning all buckets among m disks using the DM method for a Cartesian product file F in which $m_1 = m_2 = m_3 = 2$ and m>7. As can be seen, the distribution is not uniform and, in fact, disks 4 to m are never used. The obvious optimal allocation method in this case is to assign each bucket into a different disk.

Generally speaking, the Disk Modulo allocation method has a good performance (close to strict optimal) when each m_i is either 1 or a large number. On the other hand, its worst case occurs when each m_i is small but not equal to 1 (one of such cases is m_i =2 for all i). The above conclusion is supported by the following facts :
1) The Disk Modulo allocation is asympototic strict optimal when all m_i's increase to infinity [8].
2) let $F \subseteq D_1 XD_2 X...XD_k$ be a Cartesian product file with D_i partitioned into m_i disjoint subsets and F' $\subseteq D'_1 XD'_2 X...XD'_k$ be a similar file but with a greater number of records (and therefore buckets) in which D'_i is partitioned into m'_i disjoint subsets where $m'_i \ge m_i$ and m_i mod m = m'_i mod m for $1 \le i \le k$. Du and Sobolewski showed that the performance of the Disk Modulo method for the file F' is better than (closer to strict optimal)

that for the file F [9].
3) From Table 3.1, we know the Disk Modulo allocation method has a relatively poor performance for large m and small m_i's.

4. A Heuristic Allocation Method for Binary Cartesian Product Files

In a Cartesian product file F if each attribute domain D_i contains only two elements, then file F is a binary Cartesian product file. Since there are only two elements in each attribute domain, each attribute domain can be partitioned into either 1 or 2 disjoint subsets (i.e., m_i =1 or 2). Therefore, a binary Cartesian product file can be characterized by the following two properties :
1) The number of attributes (k) is usually very large.
2) The number of disjoint subsets partitioned from each D_i (m_i) is either 1 or 2.

In this section we study the allocation problem for binary Cartesian product files due to the following reasons :
1) Binary files are important. Any record type can be encoded as a binary string and it was pointed out by Rivest [12] that binary files seem to be the hardest type for which to design an "optimal" file structure, since the number of attributes is usually large and the user has the greatest flexibility in specifying queries.
2) Several papers ([3], [4] and [12]) concerning the file design problem for partial match queries are concentrated on binary files and certain types of binary Cartesian product files have been shown to be "good" file structures for partial match queries.
3) Unfortunately, Disk Modulo allocation method has a relatively poor performance for binary Cartesian product files.

In the rest of this section we first consider the cases where m is a power of 2 and a heuristic allocation method which has a better performance for binary Cartesian product files is proposed.

Definition : Let F be a k-attribute binary Cartesian product file (therefore, m_i = 1 or 2 for $1 \le i \le k$), and let m be the number of available disk units and m is a power of 2. Let $T=\{j_1, j_2, \ldots, j_h\}$ be the set of all attributes i with $m_i=2$. For convenience and without loss of generality, we shall assume that $j_i = i$ for $1 \le i \le h$. A heuristic allocation method (HEU) is defined as follows :
Bucket $[i_1, i_2, \ldots, i_k]$ (note i_j=0 or 1 for $1 \le i \le k$) is assigned to disk unit $(\Sigma^k_{j=1} i_j \cdot p_j) \bmod m$, where $p_j = 2^{(j \bmod \log m)}$ for $1 \le j \le h$ and p_j=1 for $h+1 \le j \le k$.

Example 4.1 Let F be a binary Cartesian product file with m_i=2 for $1 \le i \le 4$ and m_5=1, and m=4. Then $\log_2 m$ =2, $p_1 = 2^{(1 \bmod 2)}=2$, $p_2 = 2^{(2 \bmod 2)}=1$, $p_3 = 2^{(3 \bmod 2)}=2$, $p_4 = 2^{(4 \bmod 2)} = 1$, and p_5=1 (since m_5=1).

Table 4.1 shows the distribution of all buckets in the above example. Note that all 16 buckets are uniformly distributed among 4 disks. Let S be the set of all attributes j with m_j=1 and S_i, for $0 \le i < \log_2 m$, be the set of all attributes j with i=j mod $\log_2 m$ and $j \varepsilon T$ (i.e., m_j=2). In the previous example $S=\{5\}$, $S_0=\{2,4\}$ and $S_1=\{1,3\}$. By the definitions of S and S_i for $0 \le i < \log_2 m$, it is not hard to see that in the heuristic allocation method p_j=1 for each $j \varepsilon S$ and $p_j = 2^i$ for each $j \varepsilon S_i$, where $0 \le i < \log_2 m$.

There are two interesting properties depicted in Example 4.1 :
1) For each bucket $[i_1, i_2, \ldots, i_k]$, $0 \le i_j < m_j$ for $1 \le j \le k$. Therefore, i_j=0 for each $j \varepsilon S$ (since m_j=1) and $\Sigma^k_{j=1}(i_j \cdot p_j) \bmod m = \Sigma^k_{j=1, j \notin S}(i_j \cdot p_j) \bmod m$. That means the value of p_j for each $j \varepsilon S$ is of no importance. Since m_j=1 for each $j \varepsilon S$, the number of buckets needed to be examined to respond to a query will be the same no matter the j-th attribute is specified or not. Furthermore, let $F \subseteq D_1 X D_2 X \ldots X D_k$ be a binary k-attribute Cartesian product file and each D_i is partitioned into m_i subsets for $1 \le i \le k$ with m_j=1 for some j. For simplicity, let us assume j=1. Let $F' \subseteq D_2 X D_3 X \ldots X D_k$ be a (k-1)-attribute Cartesian product file and each D_i is partitioned into m_i subsets for $2 \le i \le k$. If bucket $[i_1, i_2, \ldots, i_k]$ in file F is assigned to disk d by the proposed allocation method, then bucket $[i_2, \ldots, i_k]$ in file F' is also assigned to disk d (since m_1=1 and i_1=0). Given a query $q=(A_1=a_1, A_2=a_2, \ldots, A_k=a_k)$ in file F, there is a query $q' = (A_2=a_2, \ldots, A_k=a_k)$ in file F' corresponding to query q and the responses to query q and q' contain the same number of buckets. If bucket $[i_1, i_2, \ldots, i_k]$ in file F is in the response to query q, then bucket $[i_2, \ldots, i_k]$ in file F' is in the response to query q'. Thus, from the performance point of view there is not much difference between file F and F'. For simplicity in the rest of this paper we are going to assume that S is empty (i.e., m_i=2 for all i) for each binary Cartesian product file.
2) For a given partial match query which contains at least two unspecified attributes, one belongs to S_0 and the other belongs to S_1, then the heuristic allocation method is strict optimal for the query. For instance, in Example 4.1 queries $q=(A_1=*, A_2=*, A_3=a_3, A_4=a_4, A_5=a_5)$, where $a_i \varepsilon D_i$ for $3 \le i \le 5$, and queries $q'=(A_1=*, A_2=a_2, A_3=*, A_4=*, A_5=a_5)$, where $a_2 \varepsilon D_2$ and $a_5 \varepsilon D_5$ are strict optimal. The readers can verify this by themselves.

Before giving a theorem which shows that the heuristic allocation method is strict optimal under certain conditions, let us define some notations first. Let $\langle b_1, b_2, \ldots, b_k \rangle$ denote the set of buckets $[i_1, i_2, \ldots, i_k]$ with $i_j = b_j$ if b_j is not a * or $0 \le i_j \le m_j - 1$ if b_j is *, where b_i is either * or $0 \le b_i \le m_i - 1$ for $1 \le i \le k$. For example $\langle *, 0, 1, * \rangle = \{[0,0,1,0], [0,0,1,1], [1,0,1,0], [1,0,1,1]\}$ if $m_1 = m_4 = 2$. $\langle b_1, b_2, \ldots, b_k \rangle$ is also the response list of a partial match query $q= (A_1=a_1, A_2=a_2, \ldots, A_k=a_k)$, where a_i=* if b_i=* or $a_i \varepsilon D_{ib_i}$ if b_i is not a * for $1 \le i \le k$. It is not hard to see

$<b_1,\ldots,b_{i-1},*,b_{i+1},\ldots,b_k> = \cup_{t=0}^{m_i-1} <b_1,\ldots,b_{i-1},t,b_{i+1},\ldots,b_k>$.

Let $G=\{<b_1,b_2,\ldots,b_k> \mid <b_1,b_2,\ldots,b_k>$ has exactly $\log_2 m$ unspecified attributes, one from each S_i, where $0\le i<\log_2 m\}$. The following lemma shows that all buckets in each element of G are "uniformly" distributed among m disks.

Lemma 4.1 Assume that $m=2^h$ for some non-negative integer h and $<b_1,b_2,\ldots,b_k>\varepsilon G$. All buckets in $<b_1,b_2,\ldots,b_k>$ are "uniformly" distributed among m disks, one in each disk, by the proposed heuristic allocation method (Due to the space limitation, all proofs in this paper are omitted.).

For instance, <*,*,0,0,0> is the response list to a query $q=(A_1=*,A_2=*,A_3=a_3,A_4=a_4,A_5=a_5)$, where $a_i\varepsilon D_{i0}$ for $3\le i\le 5$, in Example 4.1. Since $m=2^2=4$ and $1\varepsilon S_1$, and $2\varepsilon S_0$, $<*,*,0,0,0>\varepsilon G$. All four buckets [0,0,0,0,0], [0,1,0,0,0], [1,0,0,0,0] and [1,1,0,0,0] in <*,*,0,0,0> are assigned to disk 0,1,2 and 3 respectively.

Theorem 4.1 Let $m=2^h$ and q be a partial match query containing at least one unspecified attribute from each S_i for $0\le i<\log_2 m=h$. Then the heuristic allocation method is strict optimal for query q.

Assume $m=2^h$. By definition of the heuristic allocation method if $m_i=2$ for $1\le i\le k$ and k is a multiple of h, then there are exactly k/h attributes belonging to each S_i for $0\le i<h$.

Corollary 4.1 Let $m_i=2$ for $1\le i\le k$ and $m=2^h$, and k is a multiple of h. The heuristic allocation method is strict optimal for all partial match queries which contain more than (h-1).k/h unspecified attributes.

The above theorem and corollary are quite useful in practice. Let us consider a special case $m=2^2=4$ and k=20 (there are about a million possible records in the file). A query being asked usually has less than 10 attributes being specified (most likely no more than 5). Thus by the Corollary 4.1 the heuristic allocation method is strict optimal for this query.

In an information retrieval system, usually some number of attributes will never or have a very little chance being specified in a query. If there are $\log_2 m=h$ attributes never being specified, then we can assign one of such attributes to each S_i for $0\le i<h$ and the heuristic allocation method is becoming strict optimal for all possible partial match queries (with those attributes unspecified). Even if the number of such attributes is less than $\log_2 m$, the performance of the heuristic allocation method can be improved by assigning one of such attributes to each S_i and assigning the rest of attributes to those S_i's which contain no such attributes.

In the above allocation method, bucket $[i_1,i_2,\ldots,i_k]$ is assigned to disk $(\Sigma^k_{j=1} i_j.p_j) \bmod m$, where $p_j=2^{(j \bmod \log m)}$ if $m_j=2$ for $1\le j\le k$. We can extend the above allocation method by replacing $p_j=2^{(j \bmod \log m)}$ with either $p_j=2^{(j \bmod \lfloor \log m \rfloor)}$ or $p_j=2^{(j \bmod \lceil \log m \rceil)}$ to fit the cases where m is not a power of 2. The extended allocation method has almost the same performance as the original one. This will be shown in the next section.

5. Analysis and Comparisons

In this section the performance of the proposed allocation method under various conditions is compared with those of Disk Modulo and an "ideal" strict optimal allocation methods.

Let $F=D_1 XD_2 X\ldots XD_k$ be a binary Cartesian product file. We shall still assume that $m_i=2$ for $1\le i\le k$. Therefore, each D_i is partitioned into 2 single element subsets D_{i0} and D_{i1}. The number of queries with j unspecified attributes equals to $(C(k,j).2^{k-j})$, where C(k,j) is the number of ways to choose j elements from a pool with k elements.

Let $q=(A_1=a_1,\ldots,A_k=a_k)$ be a partial match query with j unspecified attributes $i_1,i_2,\ldots,i_j$ and $<c_1,\ldots,c_k>$ be the response to query q, where $c_i=a_i=*$ if $i\varepsilon\{i_1,i_2,\ldots,i_j\}$ or $a_i\varepsilon D_{ic_i}$ if $a_i\ne *$. Since $m_i=2$ for $1\le i\le k$, there are 2^j buckets in $<c_1,c_2,\ldots,c_k>$. Let $A_K(q)=(n_0,n_1,\ldots,n_{m-1})$ be an m-tuple with n_i denotes the number of buckets in $<c_1,c_2,\ldots,c_k>$ which are assigned to disk i by allocation method K. Thus, $A_{DM}(q)$ and $A_{HEU}(q)$ denote the distributions of all buckets in the response to query q among m disks when Disk Modulo and the proposed heuristic allocation methods are applied respectively.

Let $N=(n_0,n_1,\ldots,n_{m-1})$ be an m-tuple with n_i being non-negative integer. We define $N^{(i)}$ to be the m-tuple formed by a right circular shift of i positions of all m components of N. For example, if N=(1,2,3,4), then $N^{(1)}=(4,1,2,3)$, $N^{(2)}=(3,4,1,2)$, $N^{(3)}=(2,3,4,1)$ and $N^{(4)}=(1,2,3,4)=N$. Let us also define m functions $f_i(N)=N+N^{(i)}$ for $1\le i\le m$. Also for convenience, let $f_{i_1,i_2,\ldots,i_j}(N)$ denote $f_{i_1}(f_{i_2}(\ldots(f_{i_j}(N))\ldots))$. The following theorem shows that given a query q how to compute $A_{DM}(q)$ and $A_{HEU}(q)$.

Theorem 5.1 Let $q=(A_1=a_1,A_2=a_2,\ldots,A_k=a_k)$ be a partial match query with j unspecified attributes $i_1,i_2,\ldots,i_j$ and $<c_1,c_2,\ldots,c_k>$ be the response to query q. Let $t=(\Sigma_{c_i\ne *} c_i) \bmod m$ and $t'=(\Sigma_{c_i\ne *} c_i.p_i) \bmod m$, where $p_i=2^n$ if $i\varepsilon S_n$ as defined in the heuristic allocation method. Let N and N' be an m-tuple with all components to be 0 except the t-th and t'-th component to be 1 respectively. Then $A_{HEU}(q)=f_{p_{i_1},p_{i_2},\ldots,p_{i_j}}(N')$ and $A_{DM}(q)=f_{1,1,\ldots,1}(N)$, where there are j 1's in the expression.

For example, let $m_i=2$ for $1\le i\le 5$ and m=4. Since $S_0=\{2,4\}$ and $S_1=\{1,3,5\}$ (recall the definitions for S_0 and S_1 in the previous section), $p_1=p_3=p_5=2$ and $p_2=p_4=1$. Let $q=(A_1=*,A_2=*,A_3=*,A_4=a_4,A_5=a_5)$, where $a_4\varepsilon D_{40}$ and $a_5\varepsilon D_{51}$. Then t=1, t'=2, $A_{DM}(q)=f_{1,1,1}(N=(1,0,0,0))=(1,3,3,1)$ and $A_{HEU}(q)=f_{2,1,2}(N'=(0,1,0,0)))=(2,2,2,2)$.

Let $A_K(q) = (n_0, n_1, \ldots, n_{m-1})$, where n_i denotes the number of buckets, among those needed to be examined to respond to query q, being allocated to disk i by allocation method K. Thus, the time required to respond to query q is $\max\{n_0, n_1, \ldots, n_{m-1}\}$.

Now let us consider the performance of the proposed heuristic allocation method under various conditions. First assume $m = 2^h$. Given a partial match query q, from Theorem 4.1, if there exists at least one unspecified attribute from each S_i for $0 \le i < h$, then the proposed allocation method is strict optimal for query q. Let k be a multiple of h. Thus, the number (b) of elements in each S_i for $0 \le i < h$ is the same (i.e., $b = k/h$).

Assume that there are j unspecified attributes in a query q. Let Prob(i,j) denote the probability of having all j unspecified attributes in i out of h S_r's, where $0 \le r < h$.
Then Prob(1,j) =(C(h,1).C(b,j)) / C(k,j) if $j \le b$ or 0 if $j > b$.
Prob(2,j) = ((G(h,2) .C(2b,j)) / C(k,j)) - Prob(1,j) if $j \le 2b$ or 0 if $j > 2b$.
In general Prob(i,j) = ((C(h,i).C(i.b, j)) / C(k,j)) $-\Sigma^{i-1}_{r=1}$ Prob(r,j) if $j \le (i.b)$ or 0 if $j > (i.b)$.
Note $\Sigma^{i-1}_{r=1}$ Prob(r,j) = (C(h,i-1). C((i-1).b,j)) /C(k,j) if $j \le ((i-1).b)$ or 0 if $j > ((i-1).b)$.
Thus, the probability for the proposed allocation method to be strict optimal for query q is
Prob(h,j) =1- (C(h,h-1). C((h-1).b,j)) /C(k,j) if $j \le ((h-1).b)$ or 1 if $j > ((h-1).b)$.

For example, let k=8 and m=4. Given a partial match query q with 4 unspecified attributes, the probability of the proposed allocation method to be strict optimal for query q equals to Prob(h=2,j=4) =1- (C(2,1).C(4,4)) /C(8,4) =34/35≈ 0.9714.

Tables 5.1, 5.2 and 5.3 show the comparisons of the performance of an "ideal" strict optimal, the proposed heuristic and Disk Modulo allocation methods when m is a power of 2. Let SO, HEU and DM represent a strict optimal, the proposed heuristic and Disk Modulo allocation methods respectively. Let T_K and RE_K denote the average response time and the relative efficiency, respectively, of a concerned partial match query when allocation method K is employed. RE_K is defined as (100. T_{SO}) / T_K, which shows the degree of closeness of the performance of allocation method K to that of an "ideal" strict optimal allocation method. Note it is different from the probability of allocation method K to be strict optimal for a given partial match query. Also note that the case where all k attributes are unspecified is considered in Tables 5.1, 5.2 and 5.3, although there is no such partial match query according to our definition.

In comparing these results, the following can be concluded :
1) The proposed heuristic allocation method has a better performance than Disk Modulo allocation method in all cases.
2) For fixed m and k, the performance of the proposed heuristic allocation method for a query q with j unspecified attributes will first degrade as j increases. However, it will improve rapidly after j is greater than $\log_2 m$. In fact it becomes strict optimal when the number of unspecified attributes is greater than $\lceil(\log_2 m - 1).k / \log_2 m\rceil$.
3) For a fixed k but different m, the average response time to a query q with j unspecified attributes is almost the same when Disk Modulo allocation method is applied. That means the relative efficiency of Disk Modulo allocation method will decrease as the number of available disk units increases. Fortunately this is not true for the proposed heuristic allocation method.
4) The proposed allocation method has a "near" strict optimal performance for a query q in which the number (j) of unspecified attributes is greater than a small number (5 or 6).

Since the average response time to a query q is proportional to the number of unspecified attributes in q, this last point is very important. For instance, in Table 5.2 (b) RE_{HEU} =54.90 for j=3 and the difference between T_{HEU} and T_{SO} is less than one disk access. However, in the same table RE_{DM} =63.64 for j=15 and the difference between T_{DM} and T_{SO} is more than two thousand disk accesses.

When m is not a power of 2, the proposed allocation method can be extended by replacing $p_j = 2^{(j \bmod \log m)}$ with either $p_j = 2^{(j \bmod \lfloor \log m \rfloor)}$ or $p_j = 2^{(j \bmod \lceil \log m \rceil)}$. Let HEU1 and HEU2 denote the former and the latter modified heuristic allocation methods respectively. Tables 5.4 and 5.5 show the comparisons of the performance of HEU1, a strict optimal and Disk Modulo allocation methods for m=5 and 6. Similar comparisons are shown in Tables 5.6 and 5.7 for HEU2 when m=6 and 7. Although those nice results in the previous section cannot be applied to HEU1 and HEU2 any more, their performance is still fairly close to strict optimal when j is greater than 5 or 6. Some more comparisons of the performance of a strict optimal, HEU1, HEU2 and Disk Modulo allocation methods are shown in Table 5.8. The results in Table 5.8 are derived under the assumption that each query has an equal probability being asked. When m is a power of 2, both HEU1 and HEU2 allocation methods become HEU allocation method. It is not hard to see that the performance of HEU1 and HEU2 allocation methods for the cases where m is not a power of 2 is not inferior to that of HEU allocation method for the cases where m is a power of 2. Although it can not guarantee the best result, when m is not a power of 2 the suggested criterion to choose either HEU1 or HEU2 allocation methods depends on the difference between $\log_2 m - \lfloor \log_2 m \rfloor$ and $\lceil \log_2 m \rceil - \log_2 m$. If $\log_2 m - \lfloor \log_2 m \rfloor < \lceil \log_2 m \rceil - \log_2 m$ then choose HEU1 otherwise choose HEU2.

6. Summary

If a file is large and can not reside in primary memory, it is stored on a secondary storage

access concurrency, all buckets in a file need to be carefully allocated to a multiple disk system.

In this paper we are concerned with the disk allocation problem for partial match retrieval. Any record type can be encoded as a binary string and binary files are probably the hardest type for which to design a good file structure. Also, it has been shown that Cartesian product files are effective for partial match queries. Therefore, we particularly concentrate on the allocation problem for binary Cartesian product files.

Since the performance of the existing Disk Modulo allocation method is first shown to be poor for binary Cartesian product files, a heuristic allocation method for binary Cartesian product files when the number of disks is a power of 2 is first proposed. Then the proposed heuristic allocation method is extended to fit the cases where m is not a power of 2. The proposed heuristic allocation method is shown to be "near" strict optimal for a partial match query in which the number of unspecified attributes is greater than a small number (5 or 6). A systematic way to compute the response time to a given query when the proposed heuristic and Disk Modulo allocation methods are employed is also given.

7. References

1. Aho, A.V. and Ullman, J.D., "Optimal Partial-match Retrieval When Fields are Independently Specified," ACM Trans. Database Systems, vol. 4, no. 2, June 1979, pp. 168-179.

2. Bolour, A., "Optimality Properties of Multiple-key Hashing Functions," JACM, vol. 26, no. 2, April 1979, pp. 196-210.

3. Burkhard, W.A., "Hashing and Trie Algorithms for Partial Match Retrieval," ACM Trans. Database Systems, vol. 1, no. 2, June 1976, pp. 175-187.

4. Burkhard, W.A., "Partial Match Hashing Coding : Benefits of Redundancy," ACM Trans. Database Systems, vol. 4, no. 2, June 1979, pp. 228-239.

5. Chang, C.C., Lee, R.C.T. and Du, H.C., "Some Properties of Cartesian Product Files," Proc. ACM-SIGMOD 1980 Conf., Santa Monica, Calif., May 1980, pp.157-168.

6. Chang, J.M. and Fu, K.S., "Extended k-d Tree Database Organization : A Dynamic Multiattribute Clustering Method," IEEE Trans. Software Eng., vol. SE-7, no. 3, May 1981, pp. 284-290.

7. Chang, C.C., Lee, R.C.T. and Du, M.W., "Symbolic Gray Code As a Perfect Multi-attribute Hashing Scheme for Partial Match Queries," IEEE Trans. Software Eng, May 1982, pp. 235-249.

8. Du, H.C., "Some Design and Analysis Problems for Parallel Processing," Ph.D. Diss., University of Washington, Tech. Report 81-08-03.

9. Du, H.C. and Sobolewski, J.S., "Disk Allocation for Cartesian Product Files on Multiple Disk Systems," ACM Trans. Database Systems, March 1982, pp. 82-101.

10. Lin, W.C., Lee, R.C.T. and Du, H.C., "Common Properties of Some Multi-attribute File Systems," IEEE Trans. Software Eng., vol. SE-5, no. 2, March 1979, pp. 160-174.

11. Liou, J.H. and Yao, S.B., "Multi-dimensional Clustering for Data Base Organizations," Information Systems, vol. 2, 1977, pp. 187-198.

12. Rivest, R.L., "Partial-match Retrieval Algorithms," SIAM J. Comput., vol. 15, no. 1, March 1976, pp. 19-50.

13. Rothnie, J.B. and Lozano, T., "Attribute Based File Organization in a Paged Memory Environment," Comm. ACM, vol. 17, no. 2, Feb. 1974, pp. 63-69.

Table 3.1

Bucket	Assigned Disk
[0,0,0]	0
[0,0,1]	1
[0,1,0]	1
[0,1,1]	2
[1,0,0]	1
[1,0,1]	2
[1,1,0]	2
[1,1,1]	3

Disk #	# of Buckets Assigned to That Disk
0	1
1	3
2	3
3	1
4	0
.	.
.	.
.	.
m	0

Table 4.1

Bucket	Assigned Disk	Bucket	Assigned Disk
[0,0,0,0,0]	0	[1,0,0,0,0]	2
[0,0,0,1,0]	1	[1,0,0,1,0]	3
[0,0,1,0,0]	2	[1,0,1,0,0]	0
[0,0,1,1,0]	3	[1,0,1,1,0]	1
[0,1,0,0,0]	1	[1,1,0,0,0]	3
[0,1,0,1,0]	2	[1,1,0,1,0]	0
[0,1,1,0,0]	3	[1,1,1,0,0]	1
[0,1,1,1,0]	0	[1,1,1,1,0]	2

Disk #	# of Buckets Assigned to That Disk
0	4
1	4
2	4
3	4

Table 5.1 (a)

M=4 and K=8

# unspecified attributes	T_{SO}	T_{HEU}	RE_{HEU}	T_{DM}	RE_{DM}
1	1	1.0000	100.00	1	100.00
2	1	1.4286	70.00	2	50.00
3	2	2.2143	90.32	3	66.67
4	4	4.0857	97.90	6	66.67
5	8	8.0000	100.00	10	80.00
6	16	16.0000	100.00	20	80.00
7	32	32.0000	100.00	36	88.89
8	64	64.0000	100.00	72	88.89

Table 5.1 (b)

M=4 and K=16

# unspecified attributes	T_{SO}	T_{HEU}	RE_{HEU}	T_{DM}	RE_{DM}
1	1	1.0000	100.00	1	100.00
2	1	1.4667	68.18	2	50.00
3	2	2.3000	86.96	3	66.67
4	4	4.2308	94.55	6	66.67
5	8	8.1282	98.42	10	80.00
6	16	16.0699	99.56	20	80.00
7	32	32.0252	99.92	36	88.89
8	64	64.0056	99.99	72	88.89
9	128	128.0000	100.00	136	94.12
10	256	256.0000	100.00	272	94.12
11	512	512.0000	100.00	528	96.97
12	1024	1024.0000	100.00	1056	96.97
13	2048	2048.0000	100.00	2080	98.46
14	4096	4096.0000	100.00	4160	98.46
15	8192	8192.0000	100.00	8256	99.22
16	16384	16384.0000	100.00	16512	99.22

Table 5.2 (a)

M=8 and K=8

# unspecified attributes	T_{SO}	T_{HEU}	RE_{HEU}	T_{DM}	RE_{DM}
1	1	1.0000	100.00	1	100.00
2	1	1.2500	80.00	2	50.00
3	1	1.7321	57.73	3	33.33
4	2	2.6857	74.47	6	33.33
5	4	4.5357	88.19	10	40.00
6	8	8.2857	96.55	20	40.00
7	16	16.0000	100.00	35	45.71
8	32	32.0000	100.00	70	45.71

Table 5.2 (b)

M=8 and K=16

# unspecified attributes	T_{SO}	T_{HEU}	RE_{HEU}	T_{DM}	RE_{DM}
1	1	1.0000	100.00	1	100.00
2	1	1.2917	77.42	2	50.00
3	1	1.8214	54.90	3	33.33
4	2	2.8791	69.47	6	33.33
5	4	4.8571	82.35	10	40.00
6	8	8.8369	90.53	20	40.00
7	16	16.7334	95.62	35	45.71
8	32	32.5952	98.17	70	45.71
9	64	64.4073	99.37	126	50.79
10	128	128.2158	99.83	252	50.79
11	256	256.0678	99.97	462	55.41
12	512	512.0000	100.00	924	55.41
13	1024	1024.0000	100.00	1716	59.67
14	2048	2048.0000	100.00	3432	59.67
15	4096	4096.0000	100.00	6436	63.64
16	8192	8192.0000	100.00	12872	63.64

Table 5.3 (a)

M=16 and K=8

# unspecified attributes	T_{SO}	T_{HEU}	RE_{HEU}	T_{DM}	RE_{DM}
1	1	1.0000	100.00	1	100.00
2	1	1.1429	87.00	2	50.00
3	1	1.4286	70.00	3	33.33
4	1	1.9429	51.47	6	16.67
5	2	2.9571	70.00	10	20.00
6	4	4.5714	87.50	20	20.00
7	8	8.0000	100.00	35	22.86
8	16	16.0000	100.00	70	22.86

Table 5.3 (b)

M=16 and K=16

# unspecified attributes	T_{SO}	T_{HEU}	RE_{HEU}	T_{DM}	RE_{DM}
1	1	1.0000	100.00	1	100.00
2	1	1.2000	83.33	2	50.00
3	1	1.5786	63.35	3	33.33
4	1	2.2648	44.15	6	16.67
5	2	3.4725	57.59	10	20.00
6	4	5.6833	70.38	20	20.00
7	8	9.8084	81.56	35	22.86
8	16	17.8228	89.77	70	22.86
9	32	33.6608	95.07	126	25.40
10	64	65.3127	97.99	252	25.40
11	128	128.8132	99.37	462	27.71
12	256	256.3121	99.88	924	27.71
13	512	512.0000	100.00	1716	29.84
14	1024	1024.0000	100.00	3432	29.84
15	2048	2048.0000	100.00	6435	31.83
16	4096	4096.0000	100.00	12870	31.83

Table 5.4

M=5, K=16 and $P_j=2^j \bmod \lfloor \log m \rfloor$

# unspecified attributes	T_{SO}	T_{HEU1}	RE_{HEU1}	T_{DM}	RE_{DM}
1	1	1.0000	100.00	1	100.00
2	1	1.4667	68.18	2	50.00
3	2	2.2000	90.91	3	66.67
4	4	4.1538	96.30	6	66.67
5	7	7.3333	95.45	10	70.00
6	13	13.7622	94.46	20	65.00
7	26	26.4434	98.32	35	74.29
8	52	52.2962	99.43	70	74.29
9	103	103.4070	99.61	127	81.10
10	205	205.7622	99.63	254	80.71
11	410	410.3590	99.91	474	86.50
12	820	820.1538	99.98	948	86.50
13	1639	1639.2000	99.99	1807	90.70
14	3277	3277.4667	99.99	3614	90.68
15	6554	6554.0000	100.00	6995	93.70
16	13108	13108.0000	100.00	13990	93.70

Table 5.5

M=6, K=16 and $P_j=2^j \bmod \lfloor \log m \rfloor$

# unspecified attributes	T_{SO}	T_{HEU1}	RE_{HEU1}	T_{DM}	RE_{DM}
1	1.0000	1.0000	100.00	1.0000	100.00
2	1.0000	1.4667	68.18	2.0000	50.00
3	2.0000	2.2000	90.90	3.0000	66.67
4	3.0000	3.9077	76.77	6.0000	50.00
5	6.0000	6.7308	89.14	10.0000	60.00
6	11.0000	12.6643	86.86	20.0000	55.00
7	22.0000	23.7566	92.61	35.0000	62.86
8	43.0000	46.0398	93.40	70.0000	61.43
9	86.0000	89.5622	96.02	126.0000	68.25
10	171.0000	176.3846	96.95	252.0000	67.96
11	342.0000	348.5769	98.11	463.0000	73.87
12	683.0000	692.2154	98.67	926.0000	73.76
13	1366.0000	1377.4000	99.17	1730.0000	78.96
14	2731.0000	2746.3333	99.44	3460.0000	78.93
15	5462.0000	5481.5000	99.64	6555.0000	83.33
16	10923.0000	10950.0000	99.75	13110.0000	83.32

Table 5.6

M=6, K=16 and $P_j = 2^{j \bmod \log m}$

# unspecified attributes	T_{SO}	T_{HEU2}	RE_{HEU2}	T_{DM}	RE_{DM}
1	1.0000	1.0000	100.00	1.0000	100.00
2	1.0000	1.5417	64.86	2.0000	50.00
3	2.0000	2.3125	86.49	3.0000	66.67
4	3.0000	3.9148	76.63	6.0000	50.00
5	6.0000	6.6809	89.81	10.0000	60.00
6	11.0000	12.3613	88.99	20.0000	55.00
7	22.0000	23.0577	95.41	35.0000	62.86
8	43.0000	44.7051	96.19	70.0000	61.43
9	86.0000	87.3317	98.48	126.0000	68.25
10	171.0000	172.9492	98.87	252.0000	67.86
11	342.0000	343.6344	99.52	463.0000	73.87
12	683.0000	685.4478	99.64	926.0000	73.76
13	1366.0000	1368.3571	99.83	1730.0000	78.96
14	2731.0000	2734.2917	99.88	3460.0000	78.93
15	5462.0000	5465.3750	99.94	6555.0000	83.33
16	10923.0000	10927.0000	99.96	13110.0000	83.32

Table 5.7

M=7, K=16 and $P_j = 2^{j \bmod \log m}$

# unspecified attributes	T_{SO}	T_{HEU2}	RE_{HEU2}	T_{DM}	RE_{DM}
1	1.0000	1.0000	100.00	1.0000	100.00
2	1.0000	1.2917	77.42	2.0000	50.00
3	2.0000	2.0714	96.55	3.0000	66.67
4	3.0000	3.3764	88.85	6.0000	50.00
5	5.0000	5.6891	87.89	10.0000	50.00
6	10.0000	10.4472	95.72	20.0000	50.00
7	19.0000	19.6892	96.50	35.0000	54.29
8	37.0000	37.9409	97.52	70.0000	52.86
9	74.0000	74.6066	99.19	126.0000	58.73
10	147.0000	147.7451	99.50	252.0000	58.33
11	293.0000	293.8720	99.70	462.0000	63.42
12	586.0000	586.4203	99.93	924.0000	63.42
13	1171.0000	1171.4821	99.96	1717.0000	68.20
14	2341.0000	2341.4583	99.98	3434.0000	68.17
15	4682.0000	4682.0000	100.00	6451.0000	72.58
16	9363.0000	9363.0000	100.00	12902.0000	72.57

Table 5.8

K	M	T_{SO}	T_{HEU1}	RE_{HEU1}	T_{HEU2}	RE_{HEU2}	T_{DM}	RE_{DM}
8	4	2.6599	2.8579	93.07	2.8579	93.07	3.8071	69.87
8	5	2.5203	2.7107	92.98	2.7707	90.97	3.8046	66.24
8	6	2.2259	2.6053	85.44	2.6967	82.54	3.8046	58.51
8	7	2.1295	2.6041	81.77	2.2938	92.84	3.8046	55.97
8	8	1.5533	1.9975	77.76	1.9975	77.76	3.8046	40.83
8	9	1.5507	1.9676	78.81	2.0409	76.00	3.8046	40.76
8	10	1.5279	1.9619	77.88	2.0330	75.16	3.8046	40.16
8	11	1.4365	1.9470	73.78	1.8890	76.05	3.8046	37.76
8	12	1.4340	1.9321	74.22	1.9543	73.38	3.8046	37.69
8	13	1.4137	1.9321	73.17	1.7659	80.06	3.8046	37.16
8	14	1.4137	1.9321	73.17	1.7424	81.14	3.8046	37.16
8	15	1.4111	1.9321	73.04	1.6421	85.94	3.8046	37.09
8	16	1.1421	1.5431	74.01	1.5431	74.01	3.8046	30.02
16	4	24.9870	25.1243	99.45	25.1243	99.45	28.3653	88.09
16	5	20.4862	20.8778	98.12	21.1354	96.93	27.2701	75.12
16	6	17.1542	18.5109	92.67	18.1231	94.65	27.1515	63.18
16	7	14.9412	17.7862	84.00	15.4550	96.68	27.1451	55.04
16	8	12.5225	13.2763	94.32	13.2763	94.32	27.1450	46.13
16	9	11.6071	12.4597	93.16	12.9293	89.77	27.1450	42.76
16	10	10.4946	11.7945	88.98	12.2545	85.64	27.1450	38.66
16	11	9.4351	11.4304	82.54	11.0990	85.01	27.1450	34.76
16	12	8.8328	11.1344	79.33	10.5928	83.39	27.1450	32.54
16	13	8.1001	11.0870	73.06	9.6664	83.80	27.1450	29.84
16	14	7.8044	11.0582	70.58	9.1028	85.74	27.1450	28.75
16	15	7.4133	11.0512	67.08	8.4084	88.17	27.1450	27.31
16	16	6.3435	7.7227	82.14	7.7227	82.14	27.1450	23.37
24	4	246.1701	249.2461	99.97	249.2461	99.97	258.6822	96.32
24	5	199.8357	200.4949	99.67	201.8313	99.01	231.7746	86.22
24	6	166.6132	170.4278	97.76	168.5526	98.85	225.0612	74.03
24	7	143.0493	154.5536	92.56	144.0367	99.31	223.9369	63.88
24	8	124.5875	125.4996	99.27	125.4996	99.27	223.8135	55.67
24	9	111.2490	113.0549	98.40	114.9421	96.79	223.8050	49.71
24	10	100.1676	103.3942	96.88	105.3645	95.07	223.8046	44.76
24	11	91.1142	96.5833	94.34	95.4793	95.43	223.8046	40.71
24	12	83.5570	91.7665	91.05	88.3628	94.56	223.8046	37.33
24	13	77.0474	89.2935	86.29	80.9989	95.12	223.8046	34.43
24	14	71.8581	87.7426	81.90	75.3481	95.37	223.8046	32.11
24	15	67.1951	86.9325	77.30	69.8275	96.23	223.8046	30.02
24	16	62.3036	65.0619	95.76	65.0619	95.76	223.8046	27.84

ALGORITHMS FOR REPLACE-ADD BASED PARACOMPUTERS

Clyde P. Kruskal
Department of Computer Science
University of Illinois
Urbana, Illinois 61801

I. Introduction

Several groups are designing large-scale multiprocessors to take advantage of inexpensive, fast floating-point processors which will soon be available. One such project is the "NYU Ultracomputer" [3] for which much effort has gone into designing operating system algorithms [5], [10] and designing and implementing numerical algorithms (e.g. [6]). In this paper we present and analyze algorithms for solving nonnumerical problems on an idealized model of the Ultracomputer -- a "replace-add-based paracomputer".

A replace-add-based paracomputer is essentially a traditional shared memory machine augmented with an extra primitive -- the "replace-add". By exhibiting algorithms that make use of the replace-add to be faster than any algorithm for solving the same problem on a traditional shared memory machine, we show that this primitive enhances the model.

(N.B. The current Ultracomputer design is based on the "fetch-and-add" operation [4] rather than the replace-add. However, these two primitives are essentially equivalent, and all of our algorithms can be easily transferred to the newer model.)

II. The Paracomputer Model of Computation

An idealized parallel processor, dubbed a paracomputer by Schwartz [11] and classified as a WRAM by Borodin and Hopcroft [2], consists of autonomous processing elements (PEs) sharing a central memory. The model permits every PE to read or write a shared memory cell in one cycle. In particular, simultaneous reads and writes directed at the same memory cell are effected in a single cycle.

We augment the paracomputer model with the "replace-add" operation (described below) and make precise the effect of simultaneous access to the shared memory. To accomplish the latter we define the serialization principle: The effect of simultaneous actions by the PEs is as if the actions occurred in some (unspecified) serial order. For example, consider the effect of one load and two stores simultaneously directed at the same memory cell: The cell will come to contain some one of the quantities written into it. The load will return either the original value or one of the stored values, possibly different from the value the cell comes to contain. Note that simultaneous memory updates are in fact accomplished in one cycle; the serialization principle speaks only of the effect of simultaneous actions and not of their implementation.

The Replace-Add Operation

We now describe a simple yet very effective interprocessor synchronization operation, called replace-add, which takes two parameters C and E. This indivisible operation is defined to increment the value in cell C by the integer E and also return this sum to the executing PE. Moreover, replace-add must satisfy the serialization principle stated above: If C is a shared cell and many replace-add operations simultaneously address C, the effect of these operations is exactly what it would be if they occurred in some (unspecified) serial order, i.e. C is modified by the appropriate total increment and each operation yields the intermediate value of C corresponding to its position in this order.

The following example illustrates the semantics of replace-add: Assume during some cycle PE_i executes

$$\text{replace-add}(C,E_i) ,$$

PE_j executes

$$\text{replace-add}(C,E_j) ,$$

and no other operations are performed on C. Furthermore, let V be the value in C at the start of the cycle. Then, at the end of the cycle, C will contain $V + E_i + E_j$ and, depending on the serial order effected, either PE_i and PE_j will receive the values

$$V + E_i \quad \text{and} \quad V + E_i + E_j$$

respectively, or they will receive the values

$$V + E_i + E_j \quad \text{and} \quad V + E_j$$

respectively.

We stress that paracomputers, especially when augmented with the replace-add, must be regarded as idealized computational models since physical limitations such as restricted fan-in prevent their realization. However the "Ultracomputer group" at the Courant Institute of New York University is designing a parallel processor that approximates such a machine (see [3] for a

This work was supported in part by the Applied Mathematical Sciences Program of the U.S. Department of Energy under Contract No. DE-AC02-76ER03077, and in part by the National Science Foundation under Grant Nos. NSF-MCS79-21258 and NSF-MCS81-05896.

0190-3918/82/0000/0219$00.75 © 1982 IEEE

description of the architecture). A crucial aspect of the design is that multiple accesses to the same location (including replace-adds) are accomplished in approximately the same time as a single access to a location.

III. Algorithms

This section contains paracomputer algorithms for solving a wide class of problems. We consistently use N for the problem size, use P for the processor size, and denote the PEs as $PE_0,\ldots,PE_{P-1}$. Some of the algorithms assume, for the sake of clarity, that P divides N (i.e. N = LP); they are easily generalized for P not dividing N. We use the order notations O, Ω, Θ, and o as defined by Knuth [7]. The base of all logs can be assumed to be two unless otherwise specified. As in [11], we say that an algorithm is completely parallelizable if its speedup is $\Theta(P)$.

Since many algorithms have synchronization points (i.e. points that all PEs must reach before any PE passes), it is important to note the following constant-time algorithm for synchronization: Let C be an otherwise unused shared cell with initial value 0; each PE replace-adds 1 to C and waits until C has value P; the PEs are then synchronized and may continue.

Often a program will require many successive synchronizations. This can be achieved by having three synchronizing cells and rotating the synchronizations: Let C_1, C_2, C_3, be three (otherwise unused) shared cells with the values C_1 and C_2 initially 0. For the first synchronization point, each PE replace-adds 1 to C_1 and waits until C_1 has value P; the PEs are then synchronized. Some one of the PEs sets C_3 to 0. For the next synchronization C_2 is used for replace-adding and when its value reaches P, C_1 is set to 0. Note that C_3 is set to 0 before C_2 reaches P, so for the third synchronization C_3 may be used, and then C_2 set to 0. The initial state having been reestablished, we may again use C_1 and set C_3 to 0, etc.

Summing

Suppose that we are given an array $W = w_0,\ldots,w_{N-1}$ of N values, and wish to compute the partial sums $s_i = w_0 + \cdots + w_i$ for i = 0,...,N-1. This problem can be solved in time $\Theta(N/P + \log P)$ using standard algorithms for solving linear recurrences (e.g. see [11]). Thus summing is completely parallelizable for $N = \Omega(P \log P)$. Of course, the summing algorithm may be generalized by substituting any associative binary operation for addition. Note that if only s_{N-1} (the total "sum") is desired, more efficient algorithms exist for certain binary operations. For example, the maximum of N values can be determined in time $\Theta(N/P + \log\log P)$ (see [12], [13]).

Integer summing. When finding the partial sums of N integers, we can make heavy use of the replace-add to solve the problem in time $\Theta(N/P + \log\log P)$ by adapting Valiant's algorithm for finding the maximum [13].

First consider the case when $P = N(N-1)/2$. This problem is easily solved in constant time: Assign the first PE the task of finding s_1 (the second partial sum), the next two PEs the task of finding s_2, the next three PEs the task of finding s_3, etc. The partial sums s_i can be independently determined in constant time by initially setting $s_i = w_0$ and then replace-adding $w_1,\ldots,w_i$ to s_i.

Next assume merely $P \geq N$; this problem can be solved with the following algorithm:

If N = 1 set $s_0 = w_0$ and return. Otherwise perform the following five steps.

(1) Partition the N items into $g = \lceil N^2/(2P+N) \rceil$ groups $G_1,\ldots,G_g$ each of size $h \cong (2P+N)/N$, so that the first h items are in group G_1, the next h items in G_2, etc.

(2) Partition the PEs also into g groups with $h(h-1)/2 \cong P(2P+N)/N^2$ PEs in each.

(3) Assign each group of PEs to a distinct group of items, and solve the summing problem for each group independently using the preceding dependent-size integer summing algorithm.

(4) Apply this algorithm recursively to the total sums t_i of each group G_i, thereby producing $u_0,\ldots,u_{g-1}$ -- the partial sums of the t_i's.

(5) Add u_{i-1} (or 0 if i = 0) to each partial sum in G_i.

Steps (1), (2), (3), and (5) each requires constant time and, since $P \leq N(N-1)/2$, the depth of the recursion at step (4) is $\Theta(\log\log N - \log\log(P/N+1))$ (see Valiant [75]), so the entire algorithm requires time $\Theta(\log\log N - \log\log(P/N+1))$. In particular, the saturated problem (i.e. N = P) is solvable in time $\Theta(\log\log P)$.

Finally, consider the case when P < N, and use the following algorithm:

(1) Partition the items into P groups $G_0,\ldots,G_{P-1}$ each of size N/P, so that the first N/P items are in group G_0, the next N/P items in G_1, etc.

(2) Apply the sequential summing algorithm to each group independently.

(3) Apply the preceding saturated integer summing algorithm to the total sums t_i of each group G_i, thereby producing $u_0,\ldots,u_{P-1}$ -- the partial sums of the t_i's.

(4) Add u_{i-1} (or 0 if i = 0) to each partial sum in G_i.

Step (1) requires constant time, steps (2) and (4) require $\Theta(N/P)$ time, and step (3) requires $\Theta(\log\log P)$ time. Thus, the entire algorithm requires $\Theta(N/P + \log\log P)$ time and is completely parallelizable for $N = \Omega(P \log\log P)$.

Unordered integer summing. The unordered summing problem is the problem of finding the partial sums of some one unspecified permutation of the data. If the w_i are integers this sum can be formed in time $\Theta(N/P)$: initialize some temporary location T to 0 and replace-add every w_i to T. The result of the addition of w_i is the partial sum s_i.

Permutations

Suppose we are given an array $W = w_0,\ldots,w_{N-1}$ of size $N = PL$ and a permutation π of $0,\ldots,N-1$. Then the permutation problem is to permute W according to π.

Algorithm. One algorithm for solving this problem allocates a temporary array T of size N and performs the following two steps:

(1) Copy W directly into T (i.e., each PE_i moves w_{iL+j} into t_{iL+j} for $0 \leq j < L$).

(2) Copy T back into W according to π (i.e., each PE_i moves t_{iL+j} into $w_{\pi(iL+j)}$ for $0 \leq j < L$).

Analysis. Steps (1) and (2) both require time $\Theta(N/P)$. Thus the entire algorithm requires time $\Theta(N/P)$ and is completely parallelizable for $N = \Omega(P)$.

Variant. Unfortunately, the above algorithm requires extra storage proportional to N. When π is known in advance, i.e. π is not part of the data, the problem is solvable in time $\Theta(N/P)$ using extra storage proportional only to P: Partition W into R and S where $|R| = P$ and $|S| = N-P$. Copy R into a temporary array R′ (thus $W = R'$ (disjoint) UNION S). Store into R (from R′ and S) the items in $\pi^{-1}(R)$. (Note that π and hence π^{-1} are known in advance.) Store the items of R′, that have not been placed back into R, into the free locations of S. Now the problem has been reduced, in constant time, from W of size N to S of size N-P. N/P such iterations will effect the entire permutation.

Packing

Suppose we are given an array $W = w_0,\ldots,w_{N-1}$ of $N = PL$ items, some of which are marked. The packing problem is to move the i-th marked item to the i-th location of W.

Algorithm.

(1) Use integer summing (with marked items assigned 1 and unmarked items assigned 0) to determine the desired destinations of the marked items.

(2) Partition the array W into L blocks of P contiguous items. Perform the following two steps for $k = 0,\ldots,L-1$.

(a) Each PE_i stores the i-th item of the k-th block into a (distinct) temporary location t_i.

(b) Each PE_i whose associated item t_i is marked moves the item from t_i into its desired destination in W.

Analysis. Step (1) is integer summing and thus requires time $\Theta(N/P + \log\log P)$, and step (2) consists of N/P iterations of two $\Theta(1)$ operations and thus requires time $\Theta(N/P)$. Therefore, the entire algorithm requires time $\Theta(N/P + \log\log P)$ and is completely parallelizable for $N = \Omega(P \log\log P)$.

Variants. The unordered packing problem is the same as the packing problem, except that it is unnecessary for the marked items to maintain their original relative order. This problem can be solved in time $\Theta(N/P)$ by replacing summing with unordered summing in step (1) above. Thus the unordered packing problem is completely parallelizable for $N = \Omega(P)$.

Unfortunately, this algorithm requires extra storage proportional to N. Unordered packing, however, can be realized in time $\Theta(N/P)$ using extra storage proportional only to P: delete step (1) and begin step (2) with a replace-add to determine, at the k-th iteration of step (2), the desired destination of the items in the k-th block.

Merging and Sorting

In [8] we show that two lists of size m, n, where $m \leq n$ and $N = m+n$, can be merged in time $\Theta(N/P + \log\log m)$. Thus, when $m = n$ merging is completely parallelizable for $N = \Omega(P \log\log P)$.

We also show in [8] how this merging algorithm can be used to obtain a sorting algorithm which requires time

$$\Theta\left(\frac{\log N \log\log N}{\log\log\log N}\right) \quad \text{for } \Omega(P) = N = o(P \log\log P)$$

and

$$\Theta\left(\frac{N \log N}{P}\right) \quad \text{for } N = \Omega(P \log\log P).$$

Thus, sorting is completely parallelizable for $N = \Omega(P \log\log P)$.

An important special case. Suppose we wish to sort an array W consisting of N (not necessarily distinct) integers in the range 1 to N. The following algorithm solves this simpler problem in time $\Theta(N/P + \log\log P)$.

(1) Create an array C of size N initialized to 0.

(2) Count how many items have each value by incrementing (via replace-add) $C(w_i)$ for all $i \in \{1,\ldots,N\}$.

(3) Apply integer summing to C and then set $D(i) = C(i-1)$ (and $D(1) = 0$), so that D(i) is the number of items less than i.

(4) Copy W into a temporary array T.

(5) The final location j of the i-th original item is obtained as replace-add($D(t_i)$,1). Set w_j equal to t_i.

To illustrate this algorithm consider the problem of sorting the array W = (2,1,5,3,2). After step (2) above C = (1,2,1,0,1), where C(i) is the number of items with value i (e.g. two items have value 2 and no items have value 4). At step (3) summing is applied to transform C into (1,3,4,4,5); C(i) now represents the number of items less than or equal to i. D = (0,1,3,4,4) is derived from C by shifting the values of C right one position and represents the number of items less than i. At step (4) W is copied into T. Finally at step (5) the final location of the i-th item of W is determined by replace-adding 1 to $D(t_i)$. For example, the fourth item of T is 3 so its final destination in W is D(3)+1 = 4. More interestingly, since the first and fifth items of T are both 2, they both replace-add 1 to D(2) to determine their final destinations; one of them effects the replace-add first and its final destination in W is D(2)+1 = 2, and the other effects the replace-add second and its final destination in W is D(2)+1+1 = 3.

Steps (1), (2), (4), and (5) all require time $\Theta(N/P)$ and step (3) requires time $\Theta(N/P + \log\log P)$. Therefore the entire algorithm requires time $\Theta(N/P + \log\log P)$ and its speedup is $\Theta(N/(N/P + \log\log P))$. It is completely parallelizable for $N = \Omega(P \log\log P)$.

An alternate algorithm with good average-case behavior. We now describe a parallel version of quicksort and show that its average-case time complexity is $\Theta((N \log N)/P)$. Thus, using average-case analyses, comparison-exchange sorting is completely parallelizable for $N = \Omega(P)$.

Suppose we wish to sort an array W of N items. First consider the case when N = P.

If $N \leq 1$ then W is sorted. Otherwise perform the following steps.

(1) Choose an item M at random from W.

(2) Let S, E, B be the sets of items smaller than M, equal to M, and bigger than M, respectively. Apply unordered packing three times: first to pack the items of S to the beginning of W, then to pack the items of E immediately after, and finally to pack the items of B to the end of W.

(3) Assign |S| PEs to S and |B| PEs to B, and recursively apply the algorithm to S and B concurrently.

We now analyze this algorithm under the assumption that the items are all distinct, which cannot decrease the (average) execution time. Suppose that the item M chosen during step (1) is the i-th smallest item in W. Then the algorithm is recursively applied to sets of size i-1 and N-i. Since i is uniformly distributed over {1,...,N}, we are essentially constructing a random binary search tree of size N. The expected depth of the recursion is the expected height of this tree, which is known to be $\Theta(\log N)$ (see Robson [79]). Since only a constant amount of time is required for steps (1) and (2) (see section on packing), the entire algorithm requires time $\Theta(\log N)$; since N = P, the speedup is $\Theta(P \log P)/\Theta(\log P) = \Theta(P)$.

For N > P, we employ the above algorithm as if we had N PEs by assigning each PE the work performed by N/P PEs. This gives a time complexity of $\Theta(N/P)\,\Theta(\log N) = \Theta((N \log N)/P)$ and a speedup of $\Theta(P)$.

As a practical consideration, choosing M likely to be near the true median lowers the average-case time complexity of the algorithm (but not its order). One possibility is to use the median of a random sample of some R < N items. If we choose $R = O(\sqrt{P})$ the median can be found in only constant time by sorting.

Selection

Suppose we are given an array W of N items from an ordered set and an integer $1 \leq k \leq N$, and wish to find the k-th smallest item in the array. For $N \leq P$ we know of no algorithm faster than sorting. However, for N > P we can parallelize the linear sequential algorithm of Blum et al. [1] as follows.

Algorithm. If $N \leq P$ sort the items; the k-th smallest item is the k-th item in W. If N > P perform the following four steps:

(1) Partition the items into P groups of size (essentially) N/P. Assign the i-th PE to the i-th group and use the sequential fast median algorithm to find the median item in each group.

(2) Sort these medians to find M, the median of the local medians.

(3) Let S, E, and B be the sets of items smaller than M, equal to M, and bigger than M, respectively. Use unordered summing to determine |S| and |E| (the cardinalities of the sets S and E).

(4) Perform one of the following three steps:

(a) $k \leq |S|$: Pack S using unordered packing and then recursively apply this (generalized-median finding) algorithm to S, still searching for the k-th smallest item.

(b) $|S| < k \leq |S| + |E|$: The k-th smallest item is M.

(c) $|S| + |E| < k$: Pack B using unordered packing and then recursively apply this (generalized-median finding) algorithm to B, but now searching for the $k - |S| - |E|$ smallest item.

Analysis. The important property of this algorithm is that at each recursive application at least a quarter of the remaining items are elim-

inated from consideration. After $\log_{4/3}(N/P)$ recursions, the number of items remaining is no more than

$$N(3/4)^{\log_{4/3}(N/P)} = P$$

at which point we apply the sorting algorithm. The complexity of step (1) is $\Theta(N/P)$, of step (2) is $\Theta(\frac{\log P \log\log P}{\log\log\log P})$, of step (3) is $\Theta(N/P)$, and of step (4) is $\Theta(N/P + T_P(3N/4))$ where $T_P(N)$ is the complexity of the entire algorithm. Thus, the complexity $T_P(N)$ is

$$\Theta(\frac{\log P \log\log P}{\log\log\log P}) \qquad \text{if } N \leq P$$

and

$$\Theta(\frac{\log P \log\log P}{\log\log\log P} + N/P + T_P(3N/4)) \qquad \text{if } N > P$$

$$= \Theta(N/P + (\log_2(N/P) + 1)\frac{\log P \log\log P}{\log\log\log P})$$

Hence the algorithm is completely parallelizable for $N = \Omega(\frac{P \log P (\log\log P)^2}{\log\log\log P})$.

IV. Conclusion

We have presented algorithms for solving several basic problems on a replace-add-based paracomputer. All of the problems discussed are completely parallelizable, at least for large enough problems. In fact, for none of the problems does the problem size have to be significantly larger than the number of PEs in order to attain maximal speedup.

While a paracomputer not enhanced with the replace-add will also attain maximal speedup for solving the above problems, such a machine sometimes requires slightly larger problems to attain this goal. What is perhaps more significant is that a machine without the replace-add is more difficult to program and sometimes requires exorbitant overhead in order to allocate the PEs to their tasks. This manifests itself in the case of merging and therefore sorting: The Borodin-Hopcroft [2] technique for solving the PE allocation problem on the weaker model is not only unobvious but requires at each iteration several steps to reallocate the PEs. In contrast, on a replace-add-based machine it is extremely easy to solve the PE allocation problem and the resulting algorithm has low overhead.

In summary, the replace-add-based paracomputer performances for solving the above problems are, in our opinion, quite impressive. Adding to this, their ability (as noted earlier) to realize highly concurrent operating system primitives, makes replace-add-based paracomputers an architecture worth striving for. While fan-in and other limitations prevent their physical realization, they can be reasonably approximated by machines using a multistage interconnection network [3]. The "Ultracomputer group" at the Courant Institute of New York University is presently designing a prototype of such a machine and believes that a full scale version containing thousands of PEs will be constructible by the end of the decade.

V. References

[1] Manuel Blum, Robert W. Floyd, Vaughn Pratt, Ronald L. Rivest, and Robert E. Tarjan, "Time Bounds for Selection", J. Comp. and System Sciences, (Aug. 1972), pp. 448-461.

[2] A. Borodin and J. E. Hopcroft, "Routing, Merging and Sorting on Parallel Models of Computation" Proc. of ACM 14th Ann. Symp. on Theory of Computing, (May 1982), pp. 338-344.

[3] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, Marc Snir, "The NYU Ultracomputer -- Designing an MIMD, Shared-Memory Parallel Machine", Proc. 9th Ann. Symp. on Computer Architecture, (Apr. 1982), pp. 27-42, IEEE Trans. on Comp., to appear.

[4] Allan Gottlieb and Clyde P. Kruskal, "Coordinating Parallel Processors: A Partial Unification", Architecture News, (Oct. 1981), pp. 16-24.

[5] Allan Gottlieb, Boris Lubachevsky, and Larry Rudolph, "Coordinating Large Numbers of Processors", Intl. Conf. on Parallel Processing, (Aug. 1981) pp. 341-349.

[6] Malvin Kalos, "Scientific Calculations on the Ultracomputer", Ultracomputer Note #30, Courant Institute, NYU, 1981.

[7] Donald E. Knuth, "Big Omicron and Big Omega and Big Theta", SIGACT News, (Apr.-June 1976), pp. 18-24.

[8] Clyde P. Kruskal, "Results in Parallel Searching, Merging, and Sorting", Intl. Conf. on Parallel Processing, (Aug. 1982).

[9] Robson, "The Height of Binary Search Trees", Australian Computer Journal, (Nov. 1979), pp. 151-153.

[10] Larry Rudolph, "Software Structures for Ultraparallel Computing", Ph.D. Thesis, Courant Institute, NYU, (Feb. 1982).

[11] J. T. Schwartz, "Ultracomputers", ACM TOPLAS, (Oct. 1980), pp. 484-521.

[12] Yossi Shiloach and Uzi Vishkin, "Finding the Maximum, Merging and Sorting in a Parallel Computation Model", Journal of Algorithms, (Mar. 1981) pp. 88-102.

[13] Leslie G. Valiant, "Parallelism in Comparison Problems", SIAM Journal on Computing, (Sept. 1975), pp. 348-355.

CONSTRUCTING PARALLEL PROGRAMS AND THEIR TERMINATION PROOF

J.P. BANATRE, M. BANATRE, P. QUINTON
I.R.I.S.A.
Campus de Beaulieu
35042 RENNES Cédex - France

Abstract -- This note considers the construction of parallel programs and the production of their termination proof. In the first part, an original scheme for describing process cooperation is presented and it is shown how this scheme may be used for the production of termination proofs. Basically, the approach consists of mapping a system of processes into a multiset which is repeatedly decreased throughout the computation. Using properties of a well-founded ordering on finite multisets, we derive termination proofs. In the second part, an example illustrates the method and other applications are suggested.

1. Introduction

The subject of construction of parallel (or distributed) programs gains more and more interest, as microprocessor technology is goind ahead.

Several languages have been proposed which allow the description of parallel programs for example [3]. Some of the theoretical aspects involved in the semantics of these programs have been investigated in several groups. Another field of interest concerns the proof of strong correctness for distributed programs. Recent progresses are reported in [4]. It appears from the above enumeration that several kinds of investigations are going on "collaterally", but the problem of constructing parallel programs which surely terminates is never addressed globally. This is the topic of the present study.

Concerning the termination of his repetitive construct, Dijkstra states in [1], p. 41 : "The basic theorem for the repetitive construct asserts for a condition P that kept invariantly true that

(P and wp(DO,T) => wp(DO,p and non BB))

Here the term wp(DO,T) is the weakest precondition such that the repetitive construct will terminate. Given an arbitrary construct DO it is in general very hard -if not impossible- to determine wp(DO,T). I therefore suggest to design our repetitive constructs with the requirement of termination consciously in mind, i.e., to choose an appropriate proof for termination and to make the program in such a way that it satisfies the assumptions of the proof". Then he proposes to map the DO construct variables into a well-founded set, chosen to be the natural numbers under the > ordering. This idea provides a straighforward method for proving loop termination.

Our proposal applies the same type of idea to the construction of parallel programs. Constructs that we propose for expressing parallel programs are designed with the requirement of termination clearly in mind.

An original scheme for describing process cooperation is presented in section 2 and section 3 shows how this scheme may be used for termination proofs. Application of these tools in the programming of an example is demonstrated in section 4. Section 5 contains a brief review and discussion.

2. A scheme for process cooperation

Consider a system S of active processes $p_1,\ldots,p_n$. Each p_i is provided with a "weight" w_i. Cooperation between any couple (p_i,p_j) of S is governed by a condition $R(w_i,w_j)$ which has to be met before any communication between p_i and p_j occurs. Processes p_i and p_j are said to be neighbours when their weights verify condition R, otherwise they are "isolated". This neighbourhood relationship is dynamic since after cooperation, processes p_i and p_j may change their respective weights in such a way that $R(w_i,w_j)$ does not hold anymore (but $R(w_i,w_k)$ and $R(w_j,w_r)$ may hold ...).

The overall system S becomes "steady" when all its component processes become isolated.

The following program fragment gives an informal description of the functioning of process p_i :

```
pi : weight (wi) exchange δi with δj
     begin
       do
         wait(#coupling with a process pj or
                steady state#) ;
         if # steady state # then # exit do loop #
                           else bi(wi,δj)
         fi
       od ;
         γi
     end
```

Fig. 1

Process p_i possesses a weight w_i and when coupled with a process p_j "receives" information δ_j from p_j and "sends" information δ_i to p_j. Only after this information exchange, processing $b_i(w_i,\delta_j)$ takes place. If system S becomes steady, process p_i executes its postlude γ_i and terminates.

3. Termination proof

A usual tool for proving the termination of program is the well founded set : a set of elements and an ordering > defined on these elements, such that there can be non-infinite descending sequences of elements. The idea for proving termination of a process is to find a termination function that

0190-3918/82/0000/0224$00.75 © 1982 IEEE

maps process variables into a well-founded set -the value of the termination function being successively decreased through out the computation. Natural number under the $\geq$ ordering are often used for proving termination of loops [1]. In [2], multiset ordering is shown to be well-founded and is used for proving termination of production systems. Multisets are like-sets, but may contain multiple occurrences of identical elements. Consider two multisets of natural numbers M_1 and M_2, the relationship $M_1 >> M_2$ holds if M_2 can be obtained from M_1 by replacing one or more elements of M_1 by any finite sequence of natural numbers, each of which being smaller than the replaced one (more details in [2]).

Our idea consists of applying the multiset ordering for proving termination of our parallel programs. To each process p_i is associated a termination function f_i (corresponding roughly to b_i of fig.1) which maps the weights into the set of natural numbers under the usual ordering. Each application of the funtion reduces the weight through the computation. So w_i will take successively the following values $\{w_{i1}, w_{i2}, \ldots, w_{ik} \ldots\}$ such that, $\forall u,v\ u>v = w_{iu} > w_{iv}$. Assume now, that every process p_i is provided with such a function then the initial state of the global processing (involving $p_1, \ldots, p_n$) may be described by the multiset $\{w_{i1}, w_{i2}, \ldots, w_{in}\} = W_1$, any subsequent state $\{w_{i1}, w_{i2}, \ldots, w_{in}\} = W_i$ will be such that $W_i << W_1$ and any state derived from W_i will be such that $W_j << W_i$. Thus we have a simple means for proving termination of parallel programs built according to our scheme.

4. A short example

4.1. The problem and its solution

Consider the problem of sorting a set S of n (different) integers in ascending order. The following algorithm may be imagined :

A process is associated to each number and initially a weight n is attached to each process So there are n processes and $W_1 = \{\underbrace{n, \ldots, n}_{n}\}$.

The condition R between two processes p_i and p_j is defined as $R(w_i,w_j) \equiv (w_i - w_j = 0)$ i.e., two processes may cooperate iff their respective weights are identical.

Consider two processes p_i and p_j such that $R(w_i,w_j)$ is true. The actual processing performed by p_i consists in comparing value v_i (number to which p_i is associated) with v_j. If $v_i > v_j$ then decrease w_i by one, otherwise w_i remains unchanged. p_j performs the symmetric processing.

The function f_i attached to p_i is the following :

$$\underline{\text{function}}\ f_i = \underline{\text{if}}\ v_i < v_j\ \underline{\text{then}}\ w_i := w_i - 1\ \underline{\text{fi}}$$

This function possesses the property required from termination functions, proof of termination of our algorithm is then straighforward.

4.2. Functioning of the algorithm

Let S be {7,4,2,3,1}, and each process p_i be represented by a couple (v_i,w_i) where v_i represents the value to p_i and w_i the weight of p_i.

A possible processing leading to the solution is the following :

(7,5)	(4,5)	(2,5)	(3,5)	(1,5)
(7,5)	(4,4)	(2,4)	(3,5)	(1,5)
(7,5)	(4,4)	(2,3)	(3,5)	(1,4)
(7,5)	(4,4)	(2,3)	(3,4)	(1,3)
(7,5)	(4,4)	(2,3)	(3,3)	(1,2)
(7,5)	(4,4)	(2,2)	(3,3)	(1,2)
(7,5)	(4,4)	(2,2)	(3,3)	(1,1)

Two communicating processes are linked by an horizontal line.

Of course this is one among the possible paths leading to the solution. This algorithm is non-deterministic as it does not indicates how cooperating couples are selected.

When the system reaches its steady state, weight w_i of process p_i represents the position of v_i in the sequence s ; w_i may then be printed together with v_i by the γ_i part of process p_i.

4.3. Proof of termination

$W_1 = \{n, \ldots, n\}$, then given any configuration W_j derived from W_1 (by application of functions f_i's), we have $W_j << W_1$, and $\forall W_k$ derived from W_j; $W_k << W_j$. Configurations W_i have a lower bound, $W_{1b} = \{1,2,3,4,5\}$. The termination proof is then straightforward.

Remark. If the numbers are not assumed to be different, the condition R becomes $w_i = w_j \wedge v_i \neq v_j$. Thus the weight is a couple of integers (w,v). Termination proof is identical.

5. Review and discussion

Appropriate language constructs have been designed in order to describe processes and conditions. This cooperation scheme has been applied to the solution of a variety of problems : parallel pretty printer, parallel compile-time symbol resolution, implementation of unvariant properties in distributed systems (these properties are related to logical time, weights w_i are timestamps associated to each process) ...

References

[1] Dijkstra, E.W.,A discipline of programming. Prentice Hall (1976).

[2] Dershowitz, N., Manna, Z., Proving termination with multiset ordering. CACM, 22,8 (Aug. 1979), pp. 465-476.

[3] Hoare, C.A.R., Communicating Sequential Processes. CACM, 21,8 (Aug. 1978), pp. 302-321.

[4] Francez, N. Distributed termination. ACM TOPLAS 2, 1 (Jan. 1980), pp. 42-45.

[5] Sintzoff, M. Approximated Synchronization for distributed control. Note (June 1980).

MULTIPLE PIPELINE SCHEDULING IN VECTOR SUPERCOMPUTERS

Shun-Piao Su and Kai Hwang
School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

Abstract -- A Parallel task scheduling model is proposed for multi-pipeline vector processors. This model can be applied to explore maximal concurrency in vector computers, like the CRAY-1, CYBER-205, STAR-100, TI-ASC, and IBM 3838. The optimization problem of simultaneously scheduling multiple pipelines with vector tasks is shown to be NP-complete. Thus, we have developed several heuristic scheduling algorithms,which can be easily implemented in vector processors with low system overhead and high throughput performance.

Introduction

High-performance vector computers are demanded in numerical weather forecasting, structural analysis, seismic data processing, simulation of nuclear reactors, aerodynamics simulation, and among many other large-scale scientific/engineering computing applications. In contrast to a scalar processor that processes one data element at a time, a vector computer has vector instructions applied to groups of data elements, called vectors. Vector instructions have inherent advantages over equivalent scalar instructions embeded in DO-loops. A vector instruction saves repeated instruction fetches in a DO-loop and eliminates index and branch instructions for loop control. Vector computers appear as array processors or pipeline computers [8]. The array approach uses replicated processing elements (PEs) to explore spatial parallelism, such as 64 PEs in Illiac IV and 16 PEs in Burroughs Scientific Processor. Notable pipeline computers include Texas Instruments ASC system, CDC STAR-100, IBM 3838, Cray Research CRAY-1 [15], and CDC CYBER-205 [3]. Pipelined computers have been widely adopted in commercial computer systems. Array processors appear only in a few research computers [8, 10].

In this paper, we develop new methods to promote parallel execution of vector instructions in a pipelined computer. Concurrencies in programs should be exploited by multiple pipelines in a vector processor. Each task system contains a set of vector instructions (tasks) with precedence relation determined only by data dependencies. Each pipeline processor is assumed to be multifunctional, that is, capable of executing different functions at different times, but only one static function at a time.

For parallel vector processing, multiple pipelines are used to reduce the execution time of all instructions in a given task system. The problem of scheduling multiple scalar tasks on multiple pipelines has been studied by Ramamoorthy and Li [14], and Brune, et al [2]. Their results indicate that some optimal scheduling algirithms can be obtained for only very restricted classes of task systems. Li [11] studied the scheduling problem for restricted vector loops. We are interested in using several pipelines simultaneously processing a long vector task. A long vector task is partitioned into many subvectors to be processed by several pipelines simultaneously. Lloyd [12] suggested the use of several processors for a single task. The number of pipelines required to process a vector task is determined by the chosen scheduling algorithm.

In a multi-pipeline computer, significant overhead time is required to execute a vector instruction due to start-up and pipeline flushing delays [10]. This overhead time may reduce the performance of the pipeline system. Li neglected overhead in order to simplify the scheduling model for vector tasks [11,14] . Bruno and Downey [1] discussed the complexity of task sequencing including set-up time. We consider system overhead for scheduling vector tasks in multiple pipelines. We prove that the multi-pipeline scheduling problem is NP-complete, even for restricted task classes. Heuristic scheduling algorithms are developed to enable parallel vector processing. Performance bounds are derived for these heuristic algorithms. Several example task systems are used to illustrate the proposed vector scheduling methodology.

The Vector Task Scheduling Model

A functional block diagram of a typical multiple-pipeline vector computer is shown in Fig. 1. Main memory is often interleaved to minimize the acces time of vector operands. Instructions and data may appear in either vector or scalar forms. The Instruction Processing Unit (IPU) fetches and decodes both scalar and vector instructions. All scalar instructions are dispatched to the Scalar Processor for execution. The Scalar Processor contains multiple scalar pipelines. After a vectortor instruction is recognized by the IPU, the Vector Controller takes over in supervising its execution. The functions of this controller include decoding vector instructions, calculating effective vector-operand addresses, setting up the Vector Access Controller and the Vector Pipelines, and monitoring the execution of vector instructions. The Vector Access Controller is responsible for fetching vector operands by a series of main memory accesses. The Vector Buffer acts as a cache to close up the speed gap between the vector

*This research was supported in part by the U.S.A. National Science Foundation under grant ECS-80-16580, and in part by Academia Sinica, Rep. of China.

0190-3918/82/0000/0226$00.75 © 1982 IEEE

Access Controller and Vector Pipelines. We assume m identical Vector Pipelines, each of which is static and multifunctional.

We concentrated our study on vector tasks exclusively. The vector Controller is capable of scheduling several vector instructions simultaneously. The time required to complete the execution of a single vector instruction (vector task) is measured by (Kogge [10]).

$$t = t_o + t_\ell \cdot L = t_o + \tau \qquad (1)$$

where t_o is the overhead time due to start-up and flushing delays, t_ℓ is the average latency between two successive operand pairs, and L is the vector length (the number of component operands in a vector). The start-up time is measured from the initiation of the vector instruction to the entrance of the first operand pair into the pipeline. The flush time is measured from the entrance of the last operand pair to the completion of that vector instruction. The average latency is measured between two successive operand pairs entering the pipeline. The parameter $\tau = t_\ell \cdot L$ is called the productive time. Parameters t_o and t_ℓ vary with different vector instructions. The overhead time t_0 may require several hundreds of pipeline cycles. The average latency t_ℓ is usually one, two or a few pipeline cycles.

Given a task system, we wish to schedule the vector tasks among m identical pipelines such that the total execution time is minimized. For simplicity, we assume equal overhead time t_0 for all vector tasks. A vector task system can be characterized by a four-tuple $[\Pi, <, t_o, \tau]$, where

(1). $\Pi = \{T_1, T_2, \ldots, T_n\}$ is a set of n vector tasks.
(2). $<$ is a partial ordering relation, specifying the precedence relationships among the tasks in set Π.
(3). t_o is the overhead time of each vector task.
(4). $\tau : \Pi \to R$ is a time function defining the productive time $\tau(T_i)$ of each task T_i.

A parallel schedule for a vector task system $[\Pi, <, t_o, \tau]$ is a total function f, mapping each task $T \in \Pi$ into a finite subset of interval-pipeline pairs, where an interval-pipeline pair $([x,y], P_i)$ represents the event that a subtask of T is being processed by pipeline P_i during time interval $[x,y]$. If $f(T) = \{([x_1,y_1], B_1), ([x_2,y_2], B_2), \ldots, ([x_k,y_k], B_k)\}$, then the following conditions must be met in order to smooth the pipeline operations.

(1). $y_i - x_i > t_0$ and $\sum_{i=1}^{k} (y_i - x_i - t_o) = \tau(T)$, for all $x_i, y_i \in R$, $i = 1,2,\ldots,k$.
(2). $B_i \in \{P_1, P_2, \ldots, P_m\}$ for $i = 1,2,\ldots,k$. If $B_i = B_j$, then $(x_i,y_i) \cap (x_j,y_j) = \phi$.
(3). At time t, vector task T is executed by a subset of k pipelines $\{B_i : x_i \leq t \leq y_i, i = 1,2,\ldots,k\}$.
(4). The start time is $S(T) = \text{Min}\{x_1, x_2, \ldots, x_k\}$ and finish time is $F(T) = \text{Max}\{y_1, y_2, \ldots, y_k\}$.

Morever, a parallel schedule for a given task system must satisfy the following two properties:

(a). Different vector tasks cannot be processed by the same pipeline at the same time because of using only static pipes.
(b). Whenever $T_i < T_j$, then $S(T_j) \geq F(T_i)$ as governed by the precedence relation among tasks.

The finish time ω of a parallel schedule for n tasks is defined by $\omega = \text{Max}\{F(T_1), F(T_2), \ldots, F(T_n)\}$. An optimal parallel schedule has the minimal finish time ω_o among all parallel schedules for the given task system. Our objective is to find an "optimal" or "suboptimal" parallel schedule for any given vector task system. The following example will clarify the problem environment.

Example 1.

Given a vector task system $[\Pi, <, t_o, \tau]$ as specified in Fig.2(a), where $\Pi = \{T_1, T_2, T_3, T_4\}$, $t_o = 1$, $\tau(T_1) = 10$, $\tau(T_2) = 2$, $\tau(T_3) = 6$, and $\tau(T_3) = 6$, and $\tau(T_4) = 2$. We want to schedule the four tasks on two (m=2) pipelines. Using the shorthand notation $\tau_i = \tau(T_i)$ for $1 \leq i \leq 4$, a parallel schedule f_1 is obtained in Fig.2(b), where the shaded area shows the idle period of the pipelines. The vector task T_1 is partitioned into two subtasks, T_{11} and T_{12}, with $\tau_{11} = 7$ and $\tau_{12} = 3$. Similarly, the vector task T_3 is partitioned into two subtasks, T_{31} and T_{32}, with $\tau_{31} = 4$, $\tau_{32} = 2$. The parallel schedule f_1 is specified by the following mappings.

$f(T_1) = \{([0,8], P_1), ([3,7], P_2)\}$, with $S(T_1)=0$ and $F(T_1) = 8$.
$f(T_2) = \{([8,11], P_2)\}$, with $S(T_2) = 8$ and $F(T_2) = 11$.
$f(T_3) = \{([8,13], P_1), ([11,14], P_2)\}$, with $S(T_3) = 8$ and $F(T_3) = 14$.
$f(T_4) = \{([0,3], P_2)\}$, with $S(T_4) = 0$ and $F(T_4) = 3$.

The finish time of the parallel schedule f_1 is thus $\omega = F(T_3) = 14$ as revealed in Fig.2.

The NP Completeness of Pipeline Scheduling Problem

NP-complete problems have received much attention in recent years [6]. Ullman [17] proved that the general preemptive scheduling problem is NP-complete. Ramamoorthy and Li [14] studied the scheduling problem for shared-resource pipeline systems. They considered only scheduling scalar task systems. We consider here the scheduling of vector tasks. The multiple-pipeline scheduling problem can be stated as a feasibility problem: Given a vector task system $[\Pi, <, t_o, \tau]$, a vector computer with m identical pipelines, and a deadline D. Does there exist a parallel schedule f with finish time ω such that $\omega \leqq D$?

We shall consider two partial ordering relations over a given task set Π. An empty relation, ε, corresponds to the set of all independent tasks. A tree relation, θ, is a precedence relation in which all tasks are related by a single-rooted tree. Proofs of all theorems can be found in reference [16].

Theorem 1:

The feasibility problem for scheduling a task system of independent vector instructions over multiple pipelines is NP-complete.

Theorem 2:

The feasibility problem for scheduling a tree system of vector instructions over multiple pipelines is NP-complete.

If all vector tasks in an independent task system have equal productive time, i.e. τ_i is a constant for $i = 1,2,\ldots,n$, then it is possible to solve the feasibility problem in polynomial time. This suggests that, with additional restrictions on the scheduling problem, one may expect a polynomial-time scheduling algorithm. In this paper, we schedule vector tasks with different productive times. The NP-completeness of the above two feasibility problems indicates that the multiple-pipeline scheduling problem is indeed very hard to solve. Due to this computational intractability, heuristic scheduling algorithms are desired in real-life system designs, even though heuristic schedules may not be necessarily optimal.

Scheduling Independent Vector Tasks

The scheduling algorithm for independent tasks is specified with an input and a task system, $[\Pi, <, t_o, \tau]$, for m identical vector pipelines, where $\Pi = \{T_1,\ldots,T_n\}$, $\tau(T_i) = \tau_i$ for $i = 1,\ldots,n$. The output is a parallel schedule, f, for the given task system of independent tasks.

Let t_j be the time span of using pipeline P_j in the execution of a given task system. The overhead time and productive time are both included in t_j. Let k be the total number of partitions for all vector tasks in a parallel schedule. If no vector task has ever been partitioned, the average time span is computed by $t_a = (\sum_{i=1}^{n} \tau_i + n \cdot \tau_o)/m$. If a parallel schedule has k partitions, then the average time span becomes $t_k = (\sum_{i=1}^{n} \tau_i + n \cdot \tau_o + k \cdot \tau_o)$. The criterion for developing the heuristic algorithm is to make t_j, $j = 1,2,\ldots,m$, as close to the average value t_a or t_k as possible. We assume the condition that $t_a > t_o/2$ for any practical task system.

ALGORITHM A (For scheduling independent tasks):

Step 1. /Initialize parameters/

$i \leftarrow 1$; $j \leftarrow 1$; $t_j \leftarrow 0$ foe $j = 1,\ldots,m$; $f(T_i) \leftarrow \phi$ for $i = 1,\ldots,n$;

Step 2. /Assign task T_i to pipeline P_j and then check if the scheduling process is complete/

$t' \leftarrow t_j$; $t_j \leftarrow t' + t_o + \tau_i$;
If $i = n$ and $j = m$, then assign the task by $f(T_i) \leftarrow f(T_j) \cup \{([t', t_j], P_j)\}$; and terminate the process.

Step 3. /Check if the time span of pipeline P_j is within a given bound/

If $|t_j - t_a| \leqq t_o/2$ then assign task T_i to pipeline P_j with $f(T_i) \leftarrow f(T_i) \cup \{([t', t_j], P_j)\}$; increment the indices $j \leftarrow j + 1$ and $i \leftarrow i + 1$; and go to Step 2.

Step 4. /Compare the time span of pipeline P_j with the allowable bound/

If $t_j > t_a$, then go to Step 5, else assign thr task T_i to pipeline P_j with $f(T_i) \leftarrow f(T_i) \cup \{([t', t_j], P_j)\}$; increment $i \leftarrow i + 1$; and go to Step 2.

Step 5. /One subtask of T_i is being processed by pipeline P_j. Update the average time span. Assign a subtask of T_i to pipeline P_{j+1}/

Set $f(T_i) \leftarrow f(T_i) \cup \{([t', t_a + t_o/2], P_j)\}$; $o \leftarrow t_j - (t_a + t_o/2)$; $t_j \leftarrow t_a + t_o/2$; $t_a \leftarrow t_a + t_o/m$; $j \leftarrow j + 1$; $\tau_i \leftarrow \tau_o$; $t' \leftarrow t_j$; $t_j \leftarrow t' + t_o + \tau_i$; and go to Step 2.

The parallel schedule generated from Algorithm

A is denoted as f_A. Algorithm A adopts a bin packing approach [4] by assigning all possible tasks to pipeline P_1 before considering pipeline P_2 and so on. This approach has been successfully used by McNaughton [13] to construct the shortest preemptive schedule for independent tasks on $m \geq 2$ identical processors. The complexity of Algorithm A is O(n). We have used the time span criterion $|t_j - t_a| \leq t_o/2$ to decide when to partition a vector task and update the average time span t_a. The maximum number of partitions for a given task system is m - 1.

The finish time ω_o of an optimal schedule f_o for a task system of independent tasks is lower bounded by the average time span t_a. This lower bound occurs when no task is being partitioned.

$$\omega_o \geq t_a \tag{2}$$

Theorem 3:

Applying Algorithm A to the independent task system $[\Pi, \varepsilon, t_o, \tau]$ over m pipelines, we obtain on the following upper bound on the finish time ω_A of the schedule f_A.

$$\omega_A \leq t_a + (m - 1)t_o/2 \tag{3}$$

We have performed a series of simulation experiments in order to compare our results with three known scheduling algorithms: First Come First Serve (FCFS), Randomly Choose (RC), Longest Process First (LPF). We consider m = 4 pipelines with overhead time $t_o = 1$. The productive time of any task system is a random variable, uniformly distributed in the range [1,999]. We examined 100 task systems each with n independent tasks for $4 \leq n \leq 20$. Schedules for each heuristic algorithm and their average finish time are generated. Let ω_i be the average finish time of a schedule and ω_ℓ be the lower bound of the average finish time for an optimal schedule. The performance ratio (ω_i/ω_ℓ) for each scheduling algorithm is plotted in Fig.5. Algorithm A is shown superior to all three knownschduling heuristics.

Example 2:

Given a task system $[\Pi, \varepsilon, t_o, \tau]$ of independent vector tasks, where $\Pi = \{T_1, T_2, T_3, T_4, T_5\}$, and $\tau(T_1) = 13$, $\tau(T_2) = 8$, $\tau(T_3) = 7$, $\tau(T_4) = 11$, $\tau(T_5) = 3$.

A parallel schedule f_A for this task system is shown in Fig.6(a). Task T_1 is partitioned into two subtasks with $\tau_{11} = 11.25$ and $\tau_{12} = 1.75$. Similar partitioning is done for T_4 with $\tau_{41} = 3.5$ and $\tau_{42} = 7.5$. Vector tasks T_2, T_3 and T_5 are scheduled without partitioning. Such a schedule f_A is defined by

$$f_A(T_1) = \{([0,12.25],P_1),([0,2.75],P_2)\}$$
$$f_A(T_2) = \{([2.75,11.75],P_2)\}$$
$$f_A(T_3) = \{([0,8],P_3)\}$$
$$f_A(T_4) = \{([8,12.5],P_3),([0,8.5],P_4)\}$$
$$f_A(T_5) = \{([8.5,12.5],P_4)\}$$

The time spans of the four pipelines are $t_1 = 12.25$, $t_2 = 11.75$, $t_3 = 12.5$ and $t_4 = 12.5$. The finish time of f_A is $\omega_A = 12.5$, which is slightly higher than the optimal schedule with $\omega_o = 12$ as shown in Fig.6(b).

Scheduling A Tree Task System

The heuristic we developed for a tree task system (Π, θ, t_o, τ) is based on a tagged scheduling policy. First, we mark each vector task by a tag λ. If a vector task T_j has no immediate predecessors, then set tag $\lambda(T_j) \leftarrow 1$. If T_j has not been assigned a tag value and all the immediate predecessors of T_j, namely $T_{j1}, T_{j2}, \ldots, T_{jk}$, have tag values, then assign $\lambda(T_i) \leftarrow \max\{\lambda(T_{j1}), \ldots, \lambda(T_{jk})\} + 1$. After tagging, we form a group of subsets $E_1, E_2, \ldots, E_\ell$, where $E_i = \{T_j \mid \lambda(T_j) = i, T_j \in \Pi\}$, and ℓ is the largest tag value (tree height). Each E_i consists of independent tasks which canbe processed concurrently. Obviously, $\bigcup_{i=1}^{\ell} E_i = \Pi$, and $E_i \cap E_j = \phi$ if $i \neq j$. Once we obtain $E_1, E_2, \ldots, E_\ell$, we can apply Algorithm A, for each E_i, to obtain a parallel schedule for the tree system of n tasks.

Algorithm B (for scheduling tree tasks):

Step 1. Generate $E_1, E_2, \ldots, E_\ell$.

Step 2. For i = 1 to ℓ step 1 do

begin

If $|E_i| \geq 2$, call Algorithm A with the independent task system $(E_i, \varepsilon, t_o, \tau)$ as input. If $|E_i| = 1$, perfor \equal partitioning.
/The start time of the schedule for $(E_i, \varepsilon, t_o, \tau)$ equals the finish time of the schedule for $(E_{i-1}, \varepsilon, t_o, \tau)$/.

end

In Algorithm B, the time needed in Step 1 is of order O(n) as proved in [4]. For each independent task system $(E_i, \varepsilon, t_o, \tau)$, the run time using

Algorithm A has order $O(k_i)$, where k_i is the number of task in E_i. At most m - 1 partitions could occur in the schedule f_A. Thus, the complexity of Algorithm B has order O(n) and at most $\ell(m-1)$ partitions can be made in the schedule f_B.

Theorem 4:

Applying Algorithm B to a tree task system $[\Pi,\theta,t_o,\tau]$ over m pipelines, we obtain the following upper bound on the finish time ω_B, where ω_o is the finish time of an optimal schedule for the same tree task system and ℓ is the teee height.

$$\omega_B \leq \left(1 + \frac{\ell(m+1)}{2} \frac{t_o}{t_a}\right) \cdot \omega_o \qquad (4)$$

Example 3:

Given a tree task system $[\Pi,\theta,t_o,\tau]$ where $\Pi = \{T_1,\ldots,T_9\}$ follows the tree relationship shown in Fig.7(a). Suppose $t_o = 1$, $\tau_1 = 2$, $\tau_2 = 4$, $\tau_3 = 6$, $\tau_4 = 8$, $\tau_5 = 8$, $\tau_6 = 2$, $\tau_7 = 6$, $\tau_8 = 4$, $\tau_9 = 4$. We want to schedule this tree task system on m = 4 identical pipelines. Using Algorithm B, we obtain $E_1 = \{T_1,T_2,T_3,T_4\}$, $E_2 = \{T_5,T_6,T_8\}$, $E_3 = \{T_7\}$, $E_4 = \{T_9\}$ at step 1.

At Step 2, a parallel schedule f_B is generated as depicted in Fig.7(b). Shaded areas indicate the idle times of pipelines. Tasks T_2,T_3,T_4,T_5,T_7 and T_9 have been partitioned into subtasks. The schedule f_B is specified by the following mappings:

$$f_B(T_1) = \{([0,3],P_1)\}$$
$$f_B(T_2) = \{([3,6.5],P_1),([0,2.5],P_2)\}$$
$$f_B(T_3) = \{([2.5,6.75],P_2),([0,3.75],P_3)\}$$
$$f_B(T_4) = \{([3.75,7],P_3),([0,6.75],P_4)\}$$
$$f_B(T_5) = \{([7,11.75],P_1),([7,12],P_2),([7,8.25],P_3)\}$$
$$f_B(T_6) = \{([8.25,11.25],P_3)\}$$
$$f_B(T_8) = \{([7,12],P_4)\}$$
$$f_B(T_7) = \{([12,14.5],P_i): 1 \leq i \leq 4\}$$
$$f_B(T_9) = \{([14.5,16.5],P_i): 1 \leq i \leq 4\}$$

The finish time of f_B is $\omega_B = 16.5$, within the same order of magnitude as the finish time ω_o of an optimal schedule which has a lower bound $\omega_o \geq 13.25$.

Conclusions

Scheduling vector tasks in a multi-pipeline vector processor is done in parallel in the proposed scheduling algorithms. Concurrent processing allows a vector to be partitioned into several subvectors for simultaneous execution by parallel pipelines. We have considered the overhead time associated with the pipelined execution of vector instructions. The parallel pipeline scheduling problem is shown NP-complete, which preludes us from insisting on optimal scheduling algorithms. Heuristic algorithms are thus developed for independent and tree task systems. If the average time span without partitioning is longer than the overhead time, high performance is expected in these heuristic algorithms. Our study can be extended to schedule vector task systems other than independent or tree tasks: The partitioning of a vector by time units can be also converted to partitioning by vector lengths. The proposed pipeline scheduling methodology should be very useful to those who are involved in the design and evaluation of supercomputers for parallel vector processing.

References

[1] Bruno, J. and Downey, P., "Complexity of Task Sequencing with Deadlines, Set-up Times and Changeover Costs," SIAM J. Computing, Nov., 1978, pp.393-404.

[2] Bruno, J., Jones, J. W., III, and So, K., "Deterministic Scheduling with Pipelined Processor," IEEE Trans. Computers, April 1980, pp.308-316.

[3] Control Data Corp., CDC CYBER 200/Model 205 Technical Description, St. Paul, Minn., 1980.

[4] Coffman, E. G., Ed, Computer and Job Shop Scheduling Theory, New York, Wiley 1976.

[5] Deane, R. H. and White, E. R., "Balancing Workloads and Minimizing Set-Up Costs in the Parallel Processing Shop," Opl. Res. Q., Vol. 26(I), 1975, pp.45-53.

[6] Garey, M. R. and Johnson, D. S., Computers and Intractability — A Guide to the Theory of NP-Completeness, W. H. Freeman and Company, 1979, 338 pp.

[7] Hu, T. C., "Parallel Sequencing and Assembly Line Problems," Oper. Res., Vol.9, 1961, pp. 841-848.

[8] Hwang, K., Su, S. P., and Ni, L. M., "Vector Computer Architecture and Processing Technique," Advances in Computers, (M. Yovits, ed.) Vol.20, 1981, pp.115-197.

[9] Karp, R., "Reducibility Among Combinatorial Problems," Complexity of Computer Computations, (Miller, R. and Thatcher, J., eds.) Plenum Press, New York, 1972, pp.85-103.

[10] Kogge, P. M., The Architecture of Pipelined Computers," McGraw-Hill Book Company, New York, New York, Chapts. 4,5, 1981.

[11] Li, H. F., "Scheduling Trees in Parallel Pipelined Processing Environments," IEEE Trans. Computers, Nov. 1977, pp.1101-1112.

[12] Lloyd, E. L., "Scheduling Task Systems with Resources," Technical Report MIT/LCS/R-236, Laboratory for Computer Science, MIT, 1980.

[13] McNaughton, R., "Scheduling with Deadlines and Loss Functions," Management Sciences, Oct. 1959, pp.1-12.

[14] Ramamoorthy, C. V. and Li, H. F., "Sequencing Control in Multifunctional Pipeline Systems," Sagamore Computer Conference on Parallel Processing, 1975, pp.79-89.

[15] Russell, R. M., "The CRAY-1 Computer System," Comm. of Ass. of Computing Mach., Jan. 1978, pp.63-72.

[16] Su, S. P. and Hwang, K., "Multiple Pipeline Scheduling for Parallel Vector Processing," TR-EE-81-17, School of Elec. Eng., Purdue University, W. Lafayette, Indiana, April 1981.

[17] Ullman, J. D., "NP-Complete Scheduling Problems," Journal of Computer and System Sciences, Vol.10, 1975, pp.384-393.

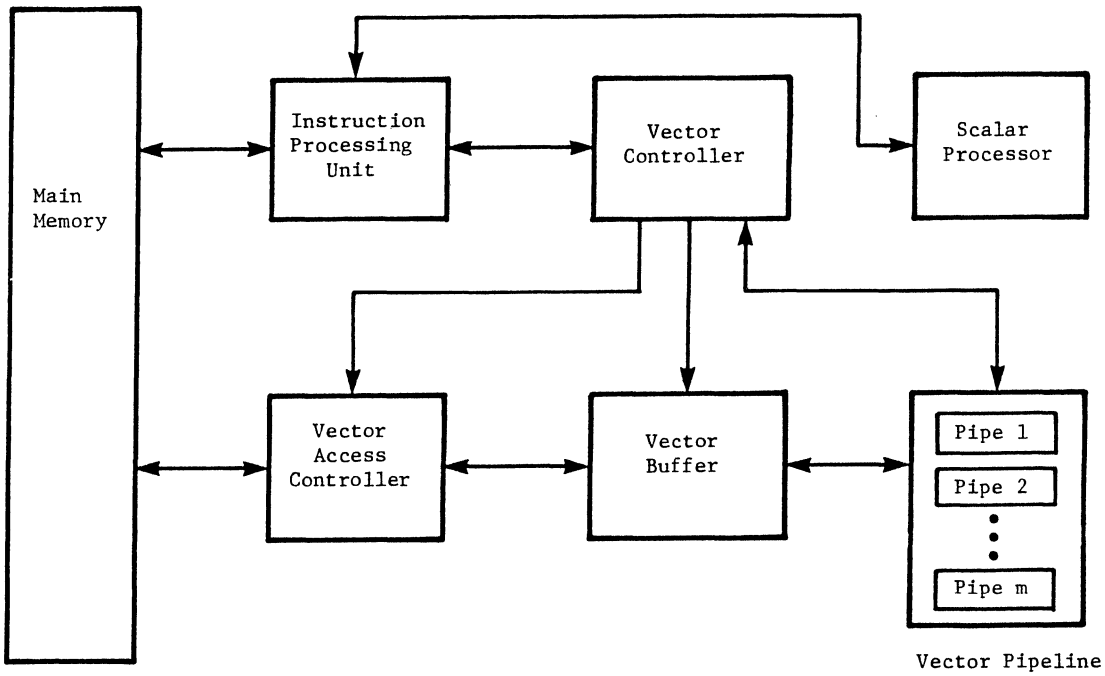
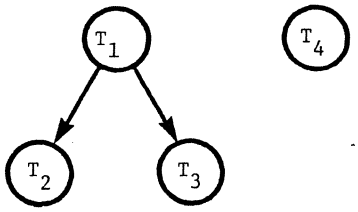
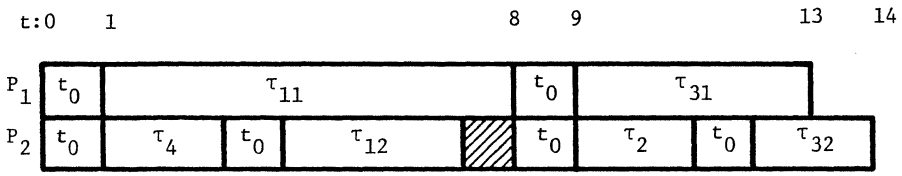


Fig. 1 The functional block diagram of a multiple-pipeline vector computer.



(a) The precedence graph of a vector task system.



(b) A parallel schedule f_1 for the task system in (a).

Fig. 2 The parallel scheduling of a task system of vector instructions.

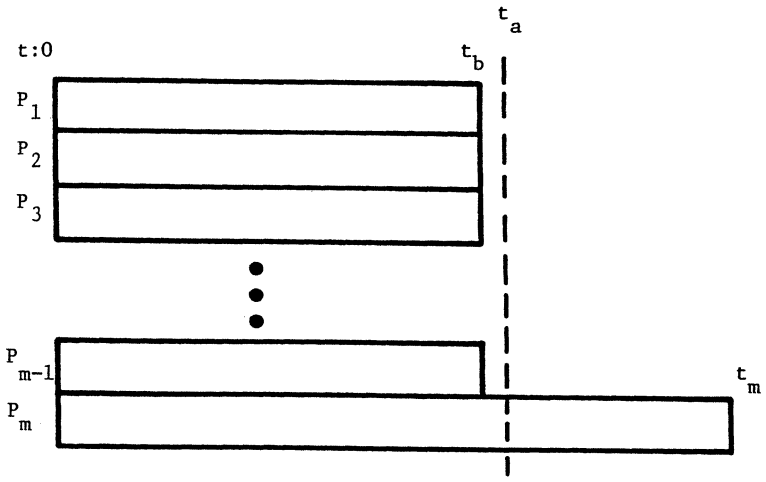


Fig. 3 A worst-case example showing the schedule f_A without partitioning in the proof of case 1 in Theorem 3.

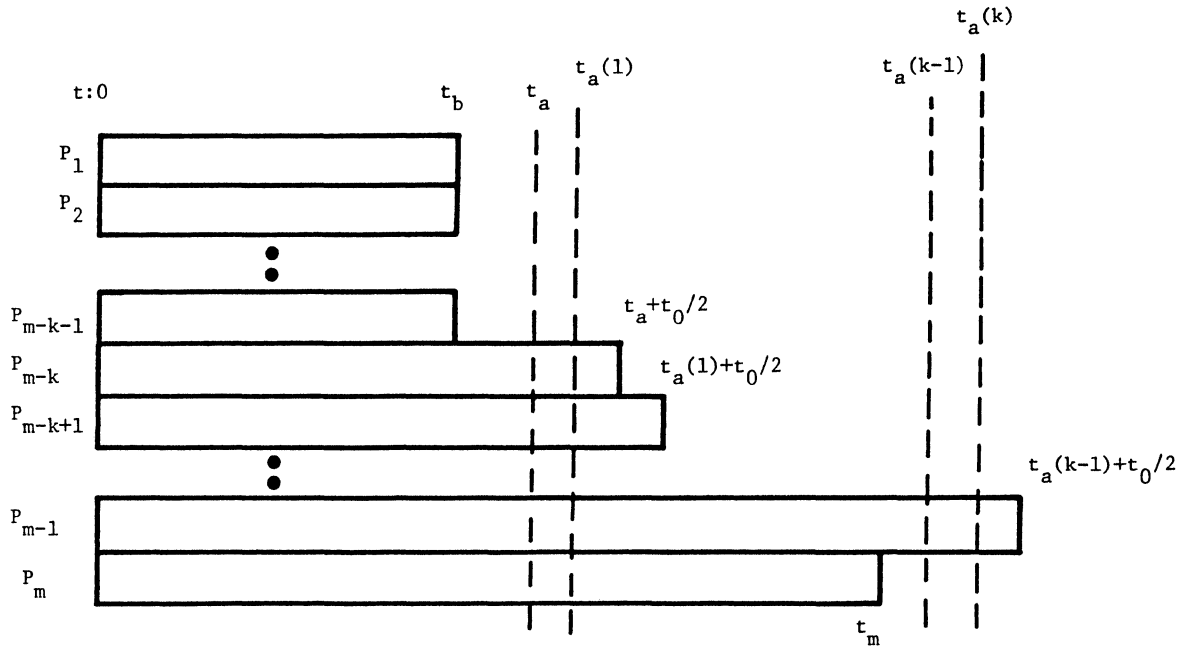


Fig. 4 A worst-case example showing the schedule f_A with k partitions in the proof of case 2 in Theorem 3.

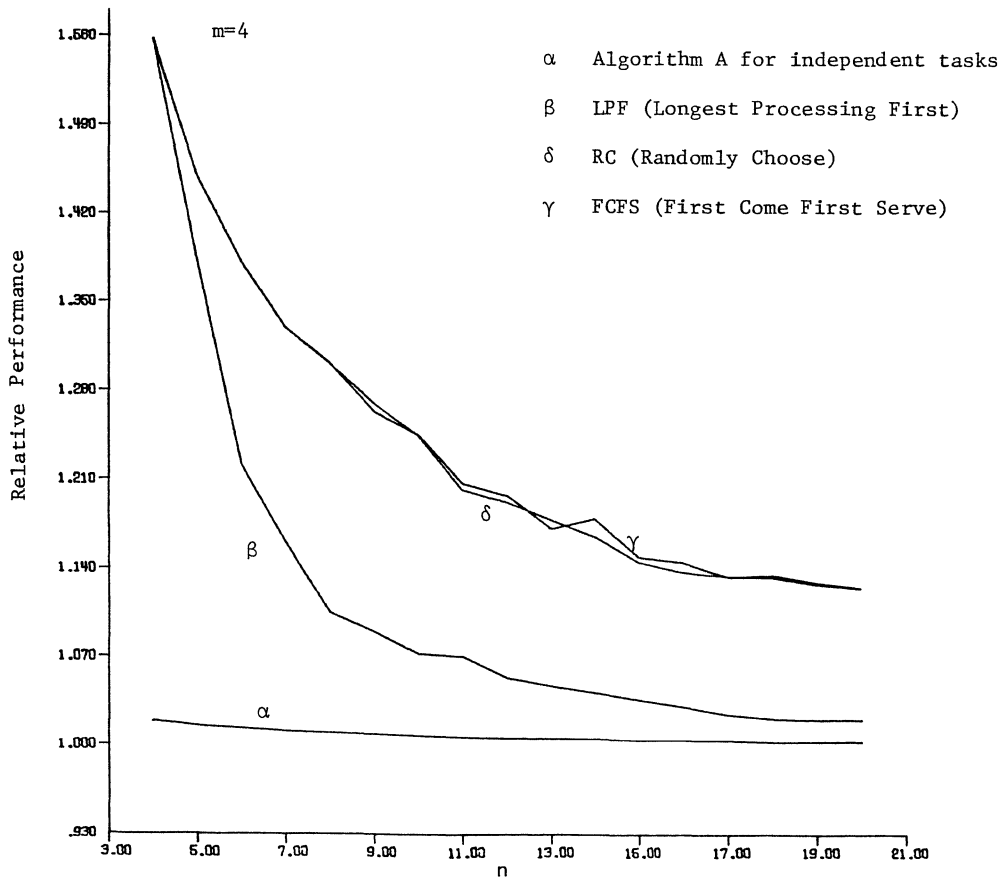
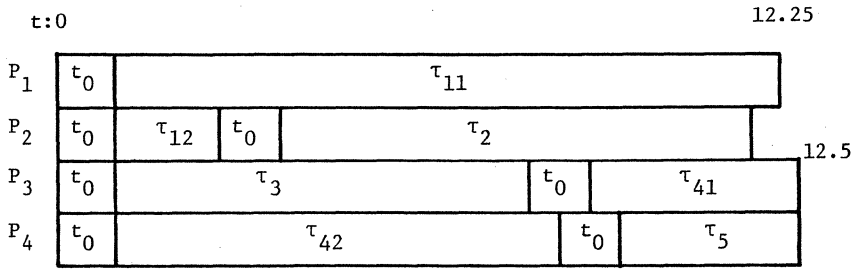
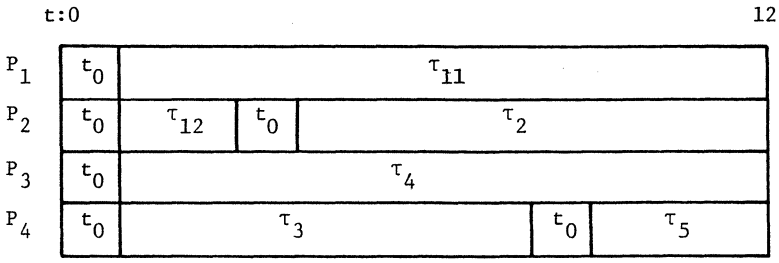


Fig. 5 Performance comparison of Algorithm A and three known scheduling algorithms.

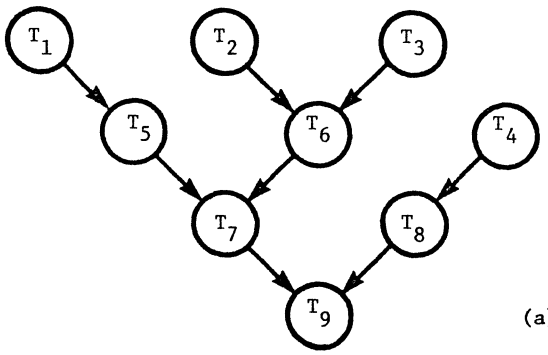


(a) A parallel schedule f_A for the task system in Example 2.

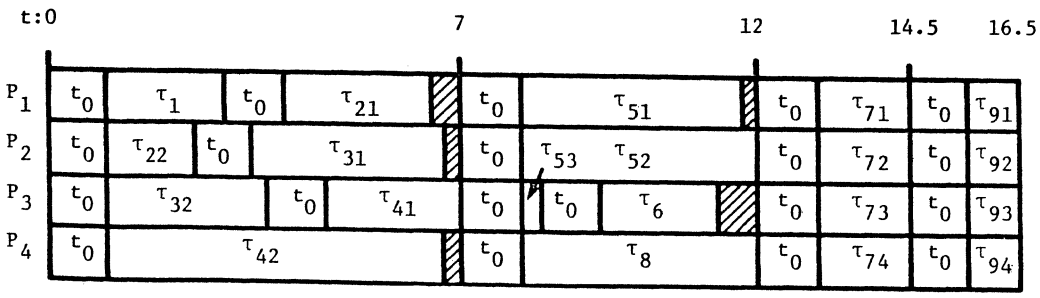


(b) A known optimal schedule for the task system in Example 2.

Fig. 6 Two parallel schedules for the task system in Example 2.



(a) A tree task system



(b) A parallel schedule f_B obtained from Algorithm B

Fig. 7 A tree task system and a parallel schedule

PERFORMANCE EVALUATION OF THREE AUTOMATIC VECTORIZER PACKAGES

Clifford N. Arnold
Research and Advanced Design Laboratory
Control Data Corporation
St. Paul, Minnesota 55112

Abstract - Eighteen kernels were used as a benchmark for three automatic vectorizer packages. The resultant code was timed on Control Data CYBER 203 and CYBER 205 systems to assess the performance improvement produced by such automated techniques. Of the 18 kernels, 16 were significantly transformed by at least one software package; thirteen ran faster on the CYBER 205 than highly optimized scalar code, and of those, eleven ran faster by at least a factor of three. A rough estimate of the programming effort to do the same vector transformation by hand showed that the automated software can speed the translation process by a factor of ten.

1. Introduction

The potential of very fast computational rates for vector and array processors necessitates code rewriting from scalar constructs to array or vector constructs. Array data structures can then be manipulated in parallel in multiple arithmetic units or streamed through pipeline units (which essentially eliminates the load/store time). The Control Data CYBER 205 and the Cray Research Cray-1S computers possess vector rates which range from two to ten times their respective scalar speeds, and typically greater than five times the speed of a CDC 7600. Experience tells us that for a given code these high rates are approached only if the large majority of the work is done in vector mode. Otherwise the scalar time and thus the scalar rate dominates. Software tools that simplify the process of generating good vector code would help users make better utilization of these machines.

Several compiler and preprocessor projects are in progress across the country to develop and perfect techniques to map scalar code into vector code. This function is called automatic vectorization. It should not be confused with scalar optimization which is a highly developed and mature technique.

Several vectorizing compilers exist in the field, but they are generally considered early versions of an underdeveloped artform. Good examples are the compilers for the Texas Instruments ASC, the Cray-1, and the CDC CYBER 200 Series. Research on automatic vectorization has gone far beyond these. For a review of such research see [1] and [2] and references therein.

Two software packages that aid the programmer and compiler in generating vector code have come to my attention. One is from Kuck and Associates, Inc. and is called the Kuck Analyzer Package (KAP) [3]. The other is from Pacific Sierra Research [4][5] and is called the Vector and Array Syntax Translator (VAST). This paper reports on an experiment to assess potential performance improvements due to each of three automatic vectorizers: KAP, VAST, and a CDC CYBER 200 FORTRAN Compiler (hereafter referred to as CYBER 200 FTN). Comments on usability and potential for productivity improvements are included.

2. Procedure

The test base consisted of 18 kernels of FORTRAN from the Lawrence Livermore Laboratory [6][7]. This benchmark, commonly called the Livermore Loops, was run through KAP and VAST. Both analyzers output FORTRAN source code with vector extensions. KAP output is not CYBER 200 compatible, so the syntax of the KAP output was manually converted to CYBER 200 FORTRAN. Care was taken not to add vector constructs unless KAP indicated the need and vector constructs were always added where indicated by KAP. The conversion was intended to be a simple translation by a process that could easily be automated. The only semantic change was to declare three arrays ROWWISE. This change was indicated by KAP (though a straightforward syntax change could also vectorize this construct). The conversion process was very time consuming and prone to errors. Most of the time was spent tracking down hard to discover typographical errors. I also needed to keep track of array declarations for new scratch vectors and had to spend time learning the CYBER 200 vector extensions that an automatic source code generator would have available to it. The process in all required many runs and six to eight person weeks.

VAST, on the other hand, produces CYBER 200 FORTRAN source code directly. The input source code was run through VAST and then through compilation and execution. A few simple user directives were added to the source code which appear as comments to the compiler, but are recognized by the VAST product as permission to collapse loops if possible. Four runs, over a period of three days were required to obtain best results.

0190-3918/82/0000/0235$00.75 © 1982 IEEE

The original code was adjusted to execute three times, varying the loop trip counts each time. For vector kernels, these counts turn into vector lengths showing the characteristics of vector performance for those loops. The ranges covered in the counts were designed to be broad and also bracket those used in the Livermore Magnetic Fusion Energy Procurement (1978).

The actual experiment was based on repetitions of seven different runs; four on the CYBER 203 and three on the CYBER 205. The original source code was compiled by CYBER 200 FTN with only scalar optimization and with both scalar optimization and vectorization. This source code was also processed through VAST followed by CYBER 200 FTN. The converted KAP output was compiled by CYBER 200 FTN.

3. Results

VAST and CYBER 200 FTN take a comparable amount of time to execute the vectorization analysis and code translation, and take considerably less time than KAP. Since the packages are written in different languages, and KAP ran on a different computer, absolute comparison is difficult. Estimates show KAP would take about one decimal order of magnitude longer to do its analysis than VAST or CYBER 200 FTN would if all were run on the same computer.

Computational performance of the kernels for the seven execution scenarios are summarized in Table 1. Table entries are in MFLOPS from the second pass of each run with trip counts as noted in Figure 4. Errors in the timings of the kernels were determined from repetitions of the runs. Typical uncertainties for a given kernel was 4 and 0.5 microseconds for the CYBER 203 and CYBER 205 respectively. The Expected Errors (MFLOPS) in Table 1 are based on the speed of the fastest execution scenario for each kernel.

When automatic vectorization is applied, all kernels except 16 and 17 show significant performance changes relative to the unvectorized run. Kernels 5, 6, 13, and 14 show the smallest variation. All execution scenarios for all variations of trip counts for these four kernels register performance within 56 percent of the unvectorized run. Of the remaining 12 kernels, only one, kernel 8, did not improve over the unvectorized performance for any scenario at some trip count. For the CYBER 205 runs, a majority of the kernels show impressive gains due to automatic vectorization. Kernels 2, 4, 5, 6, 8, 11, 13, 14, 15, and 18 discriminate these vectorizers' capability to uncover vector constructs.

Below I summarize for each kernel the automatic vectorization that has occurred. Note also the difference between the CYBER 203

TABLE 1. TIMINGS FOR THE LIVERMORE LOOPS

	CYBER 203 TIMINGS				CYBER 205 TIMINGS			
Kernel	Unvectorized	CYBER 200 FTN	VAST	KAP	CYBER 200 FTN	VAST	KAP	Expected Error
1	9.6	22.4	23.0	22.7	121.5	121.3	120.2	1.5
2	12.3	12.3	3.9	15.6	12.5	16.2	76.9	1.0
3	5.9	15.7	15.7	15.8	78.8	73.9	76.6	2.9
4	3.3	3.3	3.3	2.8	3.3	3.3	17.0	0.1
5	7.9	7.9	7.9	3.5	7.9	7.9	5.7	0.1
6	5.2	5.2	5.2	4.2	5.2	5.2	6.1	0.1
7	17.0	24.4	24.5	24.4	146.0	146.9	144.7	2.1
8	22.4	22.4	22.4	11.8	22.4	22.4	15.9	<0.05
9	13.0	18.1	17.6	18.1	80.7	81.0	81.6	0.6
10	8.6	11.9	12.0	10.2	29.6	29.8	30.7	0.3
11	1.7	7.5	8.5	7.4	8.4	8.5	8.2	0.4
12	2.9	36.8	35.3	35.6	73.0	77.4	76.0	7.9
13	3.1	3.1	3.0	2.6	3.1	3.0	4.4	<0.05
14	5.5	5.6	5.5	4.8	6.9	6.9	5.3	0.1
15	3.4	3.3	3.3	3.7	3.4	3.4	19.3	<0.05
16	0.6	0.6	0.6	0.6	0.6	0.6	0.6	<0.05
17	4.9	4.9	4.8	4.9	4.9	4.9	4.9	0.2
18	8.3	1.0	14.9	17.2	4.2	42.4	50.1	0.3

Notes:

1) Entries are in million floating point operations per second (MFLOPS).

2) Results are from the second pass with trip counts noted in Figure 4.

and CYBER 205 performance. Since these two machines have identical scalar units, the timing differences are due to the vector calculations, and therefore help to point out how much of each kernel's performance is due to vector manipulation.

Kernel 1: 1-D Hydrodynamics Excerpt
This is a simple loop which all three vectorizers succeeded in vectorizing. Two scalar broadcasts make this loop run impressively on the CYBER 205.

Kernel 2: Unrolled Inner Product
This DOT PRODUCT is camouflaged by being unrolled into a loop summing five partial products. CYBER 200 FTN generates scalar code. VAST generates five vector temporaries using gather instructions and then does five vector additions and multiplications and lastly the vector sum. KAP recognizes the DOT PRODUCT and generates the single CYBER 200 macro instruction.

Kernel 3: Inner Product
All three vectorizers recognize this as a vector DOT PRODUCT and generate the CYBER 200 macro instruction.

Kernel 4: Banded Linear System
This loop is nested to level 2, but is neither tightly nested nor trivially collapsible. VAST leaves the loop alone and CYBER 200 FTN generates scalar code. KAP inverts the loops and discovers a vector DOT PRODUCT for which it needs to gather array "A". KAP leaves the loop on "J" as a scalar loop and finishes by gathering array "X", doing the final vector multiply, and scattering array "X".

Kernel 5: Tri-Diagonal Elimination, Below Diagonal
This loop has a scalar recurrence on array "X", and therefore has been unrolled in the original code for improved scalar performance. VAST leaves the loop alone and CYBER 200 FTN schedules efficient scalar code. KAP rerolls the loop, recognizes the first order recurrence and generates a macro instruction, for which CYBER 200 FTN generates a STACKLIB call. KAP also factors a vector multiplication out of the loop.

Kernel 6: Tri-Diagonal Elimination, Above Diagonal
Aside from the backward loop counter, this kernel is nearly identical to kernel 5. VAST leaves the loop alone and CYBER 200 FTN schedules efficient scalar code. KAP rerolls the loop, recognizes the first order recurrence, and generates a macro instruction (again, a CYBER 200 STACKLIB call). The multiplication that could be vectorized and factored out of the loop is instead used in the macro instruction.

Kernel 7: Equation of State Excerpt
This loop is vectorized by all three vectorizers. Eight broadcast triads for the seventeen operands make this a very impressive loop for the CYBER 205.

Kernel 8: P.D.E. Integration
This is a good example of a loop with possible linear recurrences and many calculations that are not coupled to the recurrences. These later mentioned calculations can be factored into their own loop and thereby transformed into vector instructions. Neither VAST nor CYBER 200 FTN does this and instead scalar code is generated. Because the loop is computationally dense with temporaries, effective scheduling of the 256 word register file permits very fast scalar execution. KAP factors many, though not all, allowable calculations out of the loop. This leaves about half of the calculations in the scalar loop.

Kernel 9: Integrate Predictors
This loop steps along the rows of a two dimensional array, instead of the columnwise ordering to which CYBER 200 FORTRAN defaults. VAST and CYBER 200 FTN generate gather instructions, the vector computation, and then scatter instructions. KAP explicitly notes the parallel structure of the computation, implies the ROWWISE construct, but cannot legally implement it on the CYBER 200 without unpredictable side effects. Here I make the choice of using ROWWISE in the original code because I can rule out the side effects. Then all three vectorizers discover the same solution as a simple vector expression.

Kernel 10: Difference Predictors
This kernel has the same rowwise characteristics of kernel 9. It is vectorized by all three vectorizers in the same way when the ROWWISE statement is used. Again, KAP implies the ROWWISE usage whereas VAST and CYBER 200 FTN do not.

Kernel 11: First Sum
This loop is recognized by both CYBER 200 FTN and KAP as a linear recurrence, and the appropriate instruction macro (STACKLIB call) is generated. VAST leaves this code alone.

Kernel 12: First Difference
All three vectorizers recognize this loop as vector subtraction.

Kernel 13: 2-D Particle Pusher
This kernel contains array references in which the array elements are loaded and

stored in data dependent mappings, that is specifically not by fixed strides. VAST does not change the code. CYBER 200 FTN generates scalar code. KAP gathers and scatters by index lists for all data dependent mappings except array "H" which is left in a scalar loop. Thus, most of the arithmetic is done in vector mode. Note that array "H" can be "gathered/scattered" and calculated in vector mode only if it is a permutation list, that is, a list with no duplicate index references. Such a case is not guaranteed here.

Kernel 14: 1-D Particle Pusher

This loop is analogous to kernel 13 in having data dependent referencing of arrays. Again CYBER 200 FTN and VAST perform no vectorization. KAP vectorizes a smaller percentage of this kernel than of kernel 13. The speed reduction of the vector version relative to the scalar version suggests that the register file cannot be scheduled as effectively as when the entire loop is left as scalar.

Kernel 15: Casual FORTRAN

VAST and CYBER 200 FTN find no vectorization in this kernel. KAP replaces the IF statements with logical tests to generate control bit vectors. The two dimensional arithmetic expressions in the kernel are converted to two dimensional vector expressions with control stores. Figures 1 and 2 show the original FORTRAN and vector FORTRAN versions of the KAP output respectively. Note the discussion in Section 4 below.

```
C*******************************************************************
C*** KERNEL 15     CASUAL FORTRAN.  DEVELOPMENT VERSION.
C
C       CASUAL ORDERING OF SCALAR OPERATIONS IS TYPICAL PRACTICE.
C       THIS EXAMPLE DEMONSTRATES THE NON-TRIVIAL TRANSFORMATION
C       REQUIRED TO MAP INTO AN EFFICIENT MACHINE IMPLEMENTATION.
C
        SAVE=RTC(SAVDMY)
        DO 99915 IGLM=1,ITIMES
        CALL CLRSTK
              NR= 7
              NZ= 25
              AR= 5./3.
              BR= 7./5.
   15   DO 45  J = 2,NR
        DO 45  K = 2,NZ
               IF( J-NR)31,30,30
   30     VY(K,J)= 0.0
                 GO TO 45
   31          IF( VH(K,J+1) -VH(K,J))33,33,32
   32           T= AR
                 GO TO 34
   33           T= BR
   34          IF( VF(K,J) -VF(K-1,J))35,36,36
   35           R= AMAX1( VH(K-1,J), VH(K-1,J+1))
                S= VF(K-1,J)
                 GO TO 37
   36           R= AMAX1( VH(K,J),   VH(K,J+1))
                S= VF(K,J)
   37     VY(K,J)= SQRT( VG(K,J)**2 +R*R)*T/S
   38           IF( K-NZ)40,39,39
   39     VS(K,J)= 0.
                 GO TO 45
   40          IF( VF(K,J) -VF(K,J-1))41,42,42
   41           R= AMAX1( VG(K,J-1), VG(K+1,J-1))
                S= VF(K,J-1)
                T= BR
                 GO TO 43
   42           R= AMAX1( VG(K,J),   VG(K+1,J))
                S= VF(K,J)
                T= AR
   43     VS(K,J)= SQRT( VH(K,J)**2 +R*R)*T/S
   45   CONTINUE
99915     CONTINUE
        $DT(15)=RTC(SAVDMY)
```

Figure 1. Kernel 15-Original Code.

```
      SAVEC = RTC(SAVDMY)
      NR = 7
      NZ = 25
      AR = 5./3.
      BR = 7./5.
      DO  IV = 1,ITIMES
       CALL CLRSTK
       JNX(1:6) = SEQ(1,6,1)+1
       FV1X(1:6) = JNX(1:6)-7
       MF1(1:6) = FV1X(1:6).GE.0
       MF2(1:6) = FV1X(1:6).LT.0
       KRX(1:24) = SEQ(1,24,1)+1
       WHERE (MF1Q1(1:   1:6))
     X      VY(2:25,2:7.  - 0.0
       WHERE (MF2Q3(1:24,1:6))
     X      FV2XQ2(1:24,1:6) = VH(2:25,3:8)-VH(2:25,2:7)
       MF3Q4(1:24,1:6) = FV2XQ2(1:24,1:6).GT.OEO.AND.MF2Q3(1:24,1:6)
       MF4Q5(1:24,1:6) = FV2XQ2(1:24,1:6).LE.OEO.AND.MF2Q3(1:24,1:6)
       WHERE (MF3Q4(1:24,1:6))
     X      TBXQ6(1:24,1:6) = AR
       WHERE (MF4Q5(1:24,1:6))
     X      TBXQ6(1:24,1:6) = BR
       WHERE (MF2Q3(1:24,1:6))
     X      FV3XQ7(1:24,1:6) = VF(2:25,2:7)-VF(1:24,2:7)
       MF5Q8(1:24,1:6) = FV3XQ7(1:24,1:6).LT.OEO.AND.MF2Q3(1:24,1:6)
       MF6Q9(1:24,1:6) = FV3XQ7(1:24,1:6).GE.OEO.AND.MF2Q3(1:24,1:6)
       WHERE(MF5Q8(1:24,1:6)) DO
        RBXQ10(1:24,1:6) = AMAX1(VH(1:24,2:7),VH(1:24,3:8))
        SCXQ11(1:24,1:6) = VF(1:24,2:7)
        END WHERE
       WHERE(MF6Q9(1:24,1:6)) DO
        RBXQ10(1:24,1:6) = AMAX1(VH(2:25,2:7),VH(2:25,3:8))
        SCXQ11(1:24,1:6) = VF(2:25,2:7)
        END WHERE
       WHERE(MF2Q3(1:24,1:6)) DO
        VY(2:25,2:7) = SQRT(VG(2:25,2:7)**2+RBXQ10(1:24,1:6)*RBXQ10(1:24
     X       ,1:6))*TBXQ6(1:24,1:6)/SCXQ11(1:24,1:6)
        FV4Q12(1:24,1:6) = KRXQ13(1:24,1:6)-25
        END WHERE
   60  MF7Q14(1:24,1:6) = FV4Q12(1:24,1:6).GE.0.AND.MF2Q3(1:24,1:6)
       MF8Q15(1:24,1:6) = FV4Q12(1:24,1:6).LT.0.AND.MF2Q3(1:24,1:6)
       WHERE (MF7Q14(1:24,1:6))
     X      VS(2:25,2:7) = 0.
       WHERE (MF8Q15(1:24,1:6))
     X      FV5Q16(1:24,1:6) = VF(2:25,2:7)-VF(2:25,1:6)
       MF9Q17(1:24,1:6) = FV5Q16(1:24,1:6).LT.OEO.AND.MF8Q15(1:24,1:6)
       MF1Q18(1:24,1:6) = FV5Q16(1:24,1:6).GE.OEO.AND.MF8Q15(1:24,1:6)
       WHERE(MF9Q17(1:24,1:6)) DO
        RAXQ19(1:24,1:6) = AMAX1(VG(2:25,1:6),VG(3:26,1:6))
        SBXQ20(1:24,1:6) = VF(2:25,1:6)
        TAXQ21(1:24,1:6) = BR
        END WHERE
       WHERE(MF1Q18(1:24,1:6)) DO
        RAXQ19(1:24,1:6) = AMAX1(VG(2:25,2:7),VG(3:26,2:7))
        SBXQ20(1:24,1:6) = VF(2:25,2:7)
        TAXQ21(1:24,1:6) = AR
        END WHERE
   62  WHERE (MF8Q15(1:24,1:6))
     X      VS(2:25,2:7) = SQRT(VH(2:25,2:7)**2+RAXQ19(1:24,1:6)*RAXQ19
     X      (1:24,1:6))*TAXQ21(1:24,1:6)/SBXQ20(1:24,1:6)
   65  ENDDO
       SDT(15) = RTC(SAVDMY)
       ARRAY MF1Q1(I = 24,J = 10) = MF1(J)
       ARRAY FV2XQ2(I = 10,J = 10) = FV2X(J,I)
       ARRAY MF2Q3(I = 24,J = 10) = MF2(J)
       ARRAY MF3Q4(I = 10,J = 10) = MF3(J,I)
       ARRAY MF4Q5(I = 10,J = 10) = MF4(J,I)
       ARRAY TBXQ6(I = 10,J = 10) = TBX(J,I)
       ARRAY FV3XQ7(I = 10,J = 10) = FV3X(J,I)
       ARRAY MF5Q8(I = 10,J = 10) = MF5(J,I)
       ARRAY MF6Q9(I = 10,J = 10) = MF6(J,I)
       ARRAY RBXQ10(I = 10,J = 10) = RBX(J,I)
       ARRAY SCXQ11(I = 10,J = 10) = SCX(J,I)
       ARRAY FV4Q12(I = 10,J = 10) = FV4X(J,I)
       ARRAY KRXQ13(I = 10,J = 6) = KRX(I)
       ARRAY MF7Q14(I = 10,J = 10) = MF7(J,I)
       ARRAY MF8Q15(I = 10,J = 10) = MF8(J,I)
       ARRAY FV5Q16(I = 10,J = 10) = FV5X(J,I)
       ARRAY MF9Q17(I = 10,J = 10) = MF9(J,I)
       ARRAY MF1Q18(I = 10,J = 10) = MF10(J,I)
       ARRAY RAXQ19(I = 10,J = 10) = RAX(J,I)
       ARRAY SBXQ20(I = 10,J = 10) = SBX(J,I)
       ARRAY TAXQ21(I = 10,J = 10) = TAX(J,I)
```

Figure 2. Kernel 15-KAP Vector FORTRAN.

Kernel 16: MONTE CARLO Search Loop

No vector manipulations are done by any of the three vectorizers.

Kernel 17: Implicit Conditional Computation

No vector manipulations are done by any of the three vectorizers.

Kernel 18: 2-D Hydrodynamics Fragment

This kernel has three loops nested to level 2. It is a classic example of a "picture-in-frame" computation. The mesh points on the frame are boundary conditions to be skipped in this computation. Calculations are to take place for all the points in the picture. CYBER 200 FTN will not collapse this

construct into 1-dimensional vectors because it will not generate the control stores to avoid the frame. CYBER 200 FTN instead vectorizes the inner loops which in this case yield vectors of length 5. Both VAST and KAP generate a collapsed version of the two dimensional loops and the bit vectors to control the stores.

4. Discussion

State-of-the-Art Automatic Vectorization

Even a casual glance at the descriptive summaries listed above shows clearly that KAP did the most vectorization of the three tools studied. KAP, in fact, vectorized a superset of that done by VAST and CYBER 200 FTN combined. A look at the techniques used by each should predict this point. VAST and CYBER 200 FTN rely on pattern recognition of source code within an individual Do Loop to detect constructs for vector conversion. KAP does most of its pattern recognition within vector dependence graphs. Such a representation traces the data dependence of individual array elements and the flow dependence of sequences of source code. Parallelism is not restricted to Do Loops, and Do Loop analysis is more general. The resulting representation is closer to the sense of the flow of the calculations than the original source code in all but the most carefully designed FORTRAN programs.

In the Livermore Benchmark, kernels 4, 8, 13, 14, 15, 16, 17, and 18 represent significant tests for state-of-the-art vectorizers, circa 1981. The remaining kernels are essentially "one-liners". Kernels 16 and 17 require knowledge of the input data before vectorization transformation is reasonable. Of the significant tests remaining, KAP solved the problem optimally for all, save slight improvements on kernels 8 and 15. This last comment is made relative to the best manual vectorization effort. Kernel 15 is a good example of taking very stylized, though not uncommon code, working out the data dependencies, and generating an entirely vectorized kernel from something that did not initially look like vector code at all. I have extracted the two forms of KAP output, Figures 1 and 2 so the reader can see how the transformation is done.

From discussions I have had with compiler writers, vectorization experts, supercomputer users, and the CYBER 200 Compiler Development group, there is a general consensus that KAP represents the state-of-the-art in automatic vectorization technique at this time.

Usability

Both the CYBER 200 FORTRAN Compiler and VAST are eminently usable. Simply add one command line for either into an execution procedure and the rest is automatic. VAST allows the user to add some hints in the form of "directives" which aid the tool in its analysis. The VAST package also has easy to read output which can help the user in several ways. Figure 3 shows the kernel 5 excerpt from the VAST output. It describes clearly what problems it had converting the loop. Often the user can then recognize quickly what could be improved and what should be left alone. VAST was designed to be used in this type of interaction, with the user and tool sharing their expertise in an iterative mode to optimize code. The second part of this kernel, which is outside of the timing loop, shows the user the appropriate CYBER 200 FORTRAN vector syntax for converted code. This is a good learning device for the novice vector programmer. VAST shows that a vectorizing preprocessor can be incorporated invisibly, and that the output can be an aid to the experienced or inexperienced programmer.

The effort required to do a manual conversion of scalar code to vector code is estimated conservatively by the time I used to convert KAP output to CYBER 200 Vector FORTRAN. VAST demonstrates that the speed up in terms of programmer effort for automatically generated code is, in this case, at least a ten to one ratio over the manual effort. Thus, where KAP shows the vectorization state of the art, VAST shows how much programmer time can be saved in the automatic conversion of code.

```
   367.   C*****************************************************************
   368.   C*** KERNEL 5      TRI-DIAGONAL ELIMINATION, BELOW DIAGONAL
   369.                  IJK=LOOPS(5)-2
   370.          CALL C8WJTIME(ISAVE)
   371.          DO 9995 IGLM=1,ITIMES
   372.          CALL VINIT
***** TRANSLATION DIAGNOSTIC -- LINE   372:
      ONLY ASSIGNMENT STATEMENTS CAN BE TRANSLATED TO ARRAY SYNTAX
   373.             DO  5 I=2,IJK,3
   374.             X(I  ) = A(I  )*(Y(I  )-X(I-1))
   375.             X(I+1) = A(I+1)*(Y(I+1)-X(I  ))
   376.       5     X(I+2) = A(I+2)*(Y(I+2)-X(I+1))
*** DATA DEPENDENCY CONFLICT -- LINE   376:
     THIS LEFT HAND SIDE USE PREVENTS SAFE TRANSLATION
**** ARRAY -- X                RIGHT HAND SIDE USE ON LINE     374
**** LINE   376: TRANSLATION NOT POSSIBLE FOR THE DO LOOP
**** BEGINNING LINE --   373    LOOP LABEL --     5    LOOP INDEX -- I
   377.   C          DO A2 I=I,M        $$$  REMAINDER KERNEL
   378.   C  A2      X(I  ) = A(I  )*(Y(I  )-X(I-1))
   379.    9995 CONTINUE
   380.          CALL C8RJTIME(,,JSAVE)
   381.          SDT(5) = FLOAT(ISAVE-JSAVE)-SIII
   382.          WRITE (6,1002) SDT(5),SIII
   383.   C*******************
   384.          KKK=5
   385.          CKSUM=0.
   386.          DO 3051 I=2,IJK,3
   387.    3051 CKSUM=CKSUM+I*(X(I)+X(I+1)+X(I+2))

      NVAST=(IJK-2)/3+1
      VVAST(1;NVAST)=C8VGATHP(X(2;NVAST),3,NVAST;VVAST(1;NVAST))
      VVAST(NVAST+1;NVAST)=C8VGATHP(X(3;NVAST),3,NVAST;VVAST(NVAS
     . T+1;NVAST))
      VVAST(2*NVAST+1;NVAST)=C8VGATHP(X(4;NVAST),3,NVAST;VVAST(2*
     . NVAST+1;NVAST))
      VVAST(3*NVAST+1;NVAST)=C8VINTL(2.,3.;VVAST(3*NVAST+1;NVAST)
     . )
      VVAST(NVAST+1;NVAST)=VVAST(1;NVAST)+VVAST(NVAST+1;NVAST)
      VVAST(1;NVAST)=VVAST(NVAST+1;NVAST)+VVAST(2*NVAST+1;NVAST)
      SVAST(1)=C8SDOT(VVAST(3*NVAST+1;NVAST),VVAST(1;NVAST))
      CKSUM=CKSUM+SVAST(1)
      I=NVAST*3+2
3051  CONTINUE

**** LINE   387: DO LOOP SUCCESSFULLY TRANSLATED TO ARRAY SYNTAX
**** BEGINNING LINE --   386    LOOP LABEL --  3051    LOOP INDEX -- I
   388.          WRITE (6,910)KKK,CKSUM
   389.   C*****************************************************************
```

Figure 3. VAST Output for Kernel 5.

Performance Considerations

Throughout this paper the motivation for vectorization is to harness the potential performance improvements of the vector and array processors. Does automatic vectorization in fact do this? The answer is not a simple yes or no. Figure 4 shows the performance profiles for the eighteen kernels with varying trip counts. The profiles represent for each kernel the fastest vector version of that code as choosen from the three vector solutions. Kernels with flat profiles (kernels 5, 6, 8, 11, 13, 14, 16, and 17) are dominated by scalar computation. Except for kernel 8 their characteristic computation rate is less than 10 MFLOPS. The remaining curves have profiles which have increasing performance with increasing trip count and an asymptotic behavior for large trip counts. This is the characteristic shape of vector code executing on the CYBER 205. The computation rate of these kernels typically exceeds 50 MFLOPS, and in all cases exceeds 10 MFLOPS.

Vectorization did not produce the fastest computation rates for all the kernels. The fastest solution on the CYBER 205 for kernels 5, 8, and 14 was by CYBER 200 FTN which did not detect any vectors for these kernels. The timings in Table 3 for the CYBER 203 show

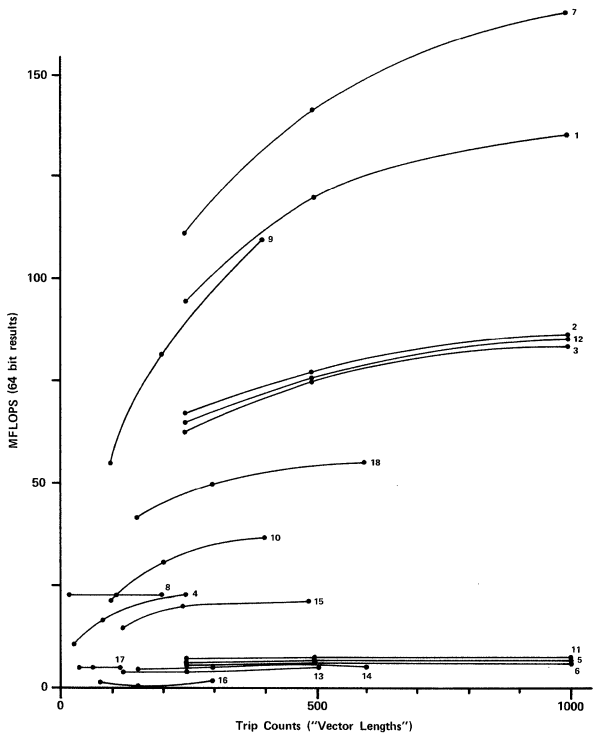


Figure 4. CYBER 205 Performance Profiles for the Livermore Kernels (Best Automatic Vectorization)

additional dramatic examples of vectorization slowing code down. These occurrences depend both on the source code and on a machine's relative vector versus scalar performance capability. For example, when vectorization adds a substantial vector set-up penalty it can negate the advantage of vector computation. In such cases the compiler should generate scalar object code. This process is called vector optimization and is machine dependent and sometimes data dependent. It should not be confused with vectorization. Note Table 1 for ranges in performance due to the three vectorizer packages.

The goal of vector optimization is to generate the best object code for a given vector computation. Using the CYBER 205 as an example, vector execution speed depends strongly on

1. vector length,
2. vector store density for bit vector control stores
3. vector set-up due to
   a. gather/scatter
   b. compress/mask-merge
4. generation of index and control bit vectors.

The optimized object code must reflect the expected execution time which as shown above may depend on many parameters. Such vector optimization must be addressed directly in future vectorizers to guarantee good performance.

Figure 5 shows the comparison for best vector speed (due to automatic vectorization) versus the best scalar speed. Because of the maturity of scalar optimization relative to automatic vectorization, the scalar times are near the best. The vector ranges on the other hand represent for each kernel the fastest vector version for that code as in Figure 4. These ranges are not guaranteed to be the best vector speed because the decision on how much to vectorize a given loop is not based on performance considerations. Figure 5 shows impressive performance improvements due to vectorization of very different types of codes.

Lastly, the following questions should be asked. Is not the effective speed of the computer determined by the code in which the computer spends the most time, which is often the slow scalar code? Has vectorization improved anything if scalar code remains and dominates? The answer is a definite yes. The code may still be scalar dominant, and yes, we have also improved things. If a computer has a very fast vector unit, relative to its scalar unit, then the percentage improvement is equal to the percentage of scalar computations converted to vector computations, whether by a compiler, or human, or preprocessor. If the scalar code can be

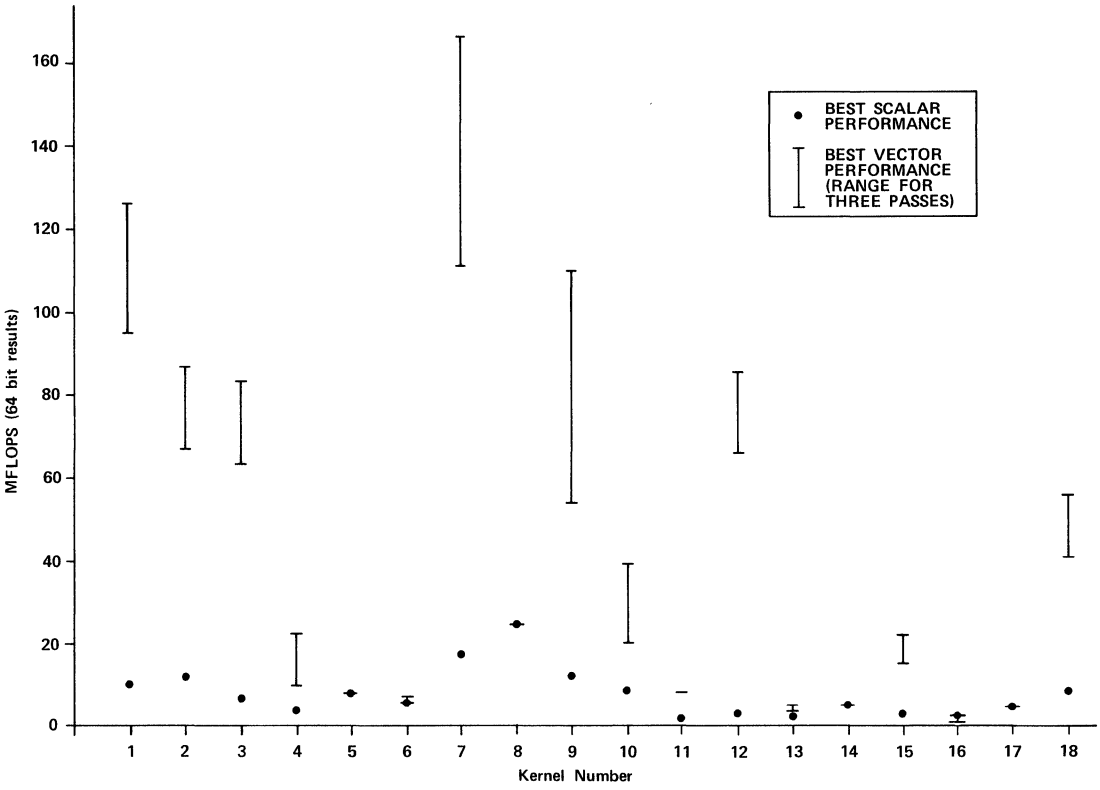


Figure 5. Scalar and Vector Performance Summary for the Livermore Kernels

reduced so that it no longer dominates the computation, then the average computation rate represents the impressive vector rate. Though this latter goal may be unlikely for a computer center workload as a whole, many users will enjoy the benefit. So the more an automatic tool does for you, the more the speed up you get. In the worst case, the improvement is linear with the vectorization success.

5. Summary: A Case for Automatic Vectorization

I have studied three automatic vectorization packages and the performance enhancement each brings to the Livermore Loops Benchmark. Performance speed-up for several kernels are very impressive, essentially achieving the theoretical speed-up. Out of the 18 kernels, automatic vectorization improved the CYBER 205 times by the following indicated factors over scalar performance:

3 kernels (1, 3, 12):	> 10 * scalar
7 kernels (2, 4, 7, 9, 11, 15, 18):	> 5 * scalar
1 kernel (10):	> 3 * scalar
2 kernels (6, 13):	>1.1 * scalar

If these 18 kernels are used as a system workload, each kernel equally weighted, automatic vectorization would increase the total throughput by 70 percent. Note that the scalar time still dominates the execution time (60 percent scalar, 40 percent vector). Kernel 16 dominates this time (59 percent of total) because it is so much slower than the other 17 kernels. If this kernel were eliminated from the sample, automatic vectorization would increase the total throughput by 170 percent, or a factor of 2.7 times in speed. In this case the scalar time is less than half the total (37 percent scalar, 63 percent vector; comparable to 11 of 17 kernels vectorized). Thus, as an approximate figure, automatic vectorization will give about 50 percent vectorization by computations count, a speedup of 100 percent in time, for code like the Livermore Benchmark. Note that in the best case when a code is full of loops analogous to kernels 1, 2, 3, 4, 7, 9, 10, 12, 15, and 18, automatic vectorization will bring a manyfold improvement over the scalar execution.

The KAP in all cases did equal or more vectorization than the CYBER 200 FTN or VAST, and is generally agreed to represent the state of the art in analysis technique. Of the 170

percent speed-up quoted above, 66 percent can be attributed to CYBER 200 FTN vectorization, an additional 11 percent to VAST vectorization, and an additional 93 percent to KAP. This assumes that as information pertaining to the vectorizability of a kernel is increased the kernel will at worst run at the same speed and may run faster; that is, the best vector solution is deterministic. This point has yet to be proven.

The amount of programmer time and effort that an easy to use vectorizer can save over an equivalent manual solution is demonstrated by comparing the time to use VAST as opposed to KAP. Translating manually from KAP vector output to CYBER 200 FORTRAN took ten to fifteen times longer than the 2 1/2 days to get a best effort from VAST. The time period for the KAP work was not an attribute of KAP, but of the manual process for translating and debugging code. VAST shows that there is nothing particularly difficult in generating CYBER 200 FORTRAN automatically. There is a lesson to be learned when I spend 6 weeks on something that need only take 2 1/2 days.

Thus, as a programmer productivity aid, automatic vectorization has high potential. Presently a tool does not exist that has the vectorization potential of KAP and the usability of VAST.

The most salient criticism of automatic vectorization is that it does not address the total problem as stated by those users who need the best conceivable vector solution. That is, it does not rewrite code on a global scale, nor reorganize the mathematical approach. In simple terms, those who need the best solution find they need to rework the problem from scratch. There are a lot of human factors involved in such an effort. The required imagination and thinking cannot be replaced. Interactive vectorization tools could help program development. Such tools would have to be able to respond to queries on the vectorizability of a kernel and the resulting side effects to other routines. It should be able to predict both the static and dynamic memory requirements of program changes. Graphical response showing how the calculations proceed through the grid space would be helpful to the user who interacts well with chalk board tactics.

Past experience shows that such tools will encourage the user's imagination and insight, and thus truly help the code development effort. There is nothing in this scenario that appears to be beyond the technical ability of near future vectorizers. As vector and array processors become more pervasive, the demand for such software will rapidly increase. Therefore, I see a promising future for automatic vectorization software and the users who must rely on it.

Acknowledgement - The author thanks Jim Emery and Eric Rowe of Control Data Corporation for their helpful comments during the preparation of this text.

References

[1] D. Kuck, R. Kuhn, B. Leasure, M. Wolfe, "The Structure of an Advanced Vectorizer for Pipelined Processors," Proc. of COMPSAC 80, the 4th Int'l Computer Software and Applications Conference, Chicago, Ill., October, 1980, pps. 709-715.

[2] D. Kuck, R. Kuhn, D. Padua, B. Leasure, M. Wolfe, "Dependence Graphs and Compiler Optimizations," Proc. of the 8th ACM Symposium on Principles of Programming Languages, Williamsburg, Va., January, 1981, pps. 207-218.

[3] M. Wolfe, B. Leasure, Understanding KAP Output Listings, Kuck and Associates Inc., September, 1981.

[4] B. Brode, "Precompilation of Fortran Programs to Facilitate Array Processing," Computer, Vol. 14, Number 9, September, 1981, pps. 46-51.

[5] B. Brode, VAST User's Guide (CYBER 205 Output Option), Pacific Sierra Research Corporation, Publication No. N-355-A, September 1981.

[6] F. McMahon, Unpublished. (Available from C. N. Arnold on request.)

[7] F. McMahon, To be Published as a Livermore Report.

RESULTS OF PARALLEL PROCESSING A LARGE SCIENTIFIC PROBLEM ON A COMMERCIALLY AVAILABLE MULTIPLE-PROCESSOR COMPUTER SYSTEM

Robert Hiromoto
Computing Division
Los Alamos National Laboratory
Los Alamos, New Mexico 87545

Abstract

Presented is a summary of a parallel-processing experiment designed to study the feasibility of doing large-scale scientific calculations on multiple-processor architectures. This particular experiment was performed on a UNIVAC 1100/80 computer system, whose architecture (configured about a common memory) eliminates the need for data transmission between processors. The algorithm used in the experiment is a particle-in-cell (PIC) method; it was selected because of its large, independent computational tasks that are adaptable to this particular parallel-processing architecture. Timing results for the parallel-processing version of this algorithm using one, two, three, and four identical processors are given and are shown to have promising speedup times when compared to the overall run times measured for a single processor version of the algorithm.

Summary

This paper presents the results of an investigation concerning the feasibility of parallel processing a significant scientific problem on a commercially available multiple-processor system. Of particular interest is the computational speed-up as a function of the number of processors employed. The algorithm used in this experiment is a particle-in-cell (PIC) method for simulating the electrostatic interactions of a collisionless plasma [1]. This particular problem represents a large scientific calculation of interest to the Los Alamos National Laboratory as well as a class of algorithms exhibiting limited vector capabilities. Figure 1 illustrates the multiple/single-thread PIC algorithm implemented in our experiment with threads 1 and 4 parallel processed.

An initialization stage of the algorithm sets up the aggregate of plasma particles positions, velocities, and corresponding charge distribution. For each discretized time step (Δt), the main computational loop advances the particle's position and velocity through the effects of the electrostatic interactions arising between particles and a uniform, background electric/magnetic field. Throughout this loop, a particle-in-cell method is employed to decompose a region of space into a collection of cells [2]. These cells are then used to track particle movement, and assist in the evaluation of the total charge distribution (C), the electrostatic potential (ϕ), and the electric field (E) under which all particles are accelerated (pushed).

Our experiment was successfully implemented and timed on a UNIVAC 1100/80 multiple-processor computer system at Sperry UNIVAC, Roseville, Minnesota. A simplified diagram of the UNIVAC 100/80 is given in Fig. 2. A principal feature of this system is the ability of all processors to execute a single instruction stream in parallel upon data in common memory. This feature is supported by the Cobol compiler but not by the Fortran compiler. Software designed by David Hammer (a UNIVAC consultant with Sandia National Laboratories, Albuquerque, New Mexico) enabled a single copy of the PIC code written in Fortran to be implemented in a parallel-processing mode. The multiple-processor computer system may be configured with one, two, three, or four processors. A further characteristic of the UNIVAC 1100/80 architecture is that no one physical processor may always have exclusive access to the execution of a given activity. On the contrary, depending upon the length of the task itself, all the physical processors may have time-shared portions of the activity's entire execution stream. A distinction, therefore, is made between activities and processors.

Because of software addressing limitations, the PIC code was restricted to a maximum of 262,000 decimal words of total memory. For each particle, five data quantities (two for position and three for velocity) were required. Three mesh quantities, constituting a 34 X 34 mesh size, were required and duplicated for a maximum of eight particle-push activities. A total of 37,000 particles were initiated for processing, requiring 213,000 words of memory (particle plus mesh data). An additional 47,000 words of memory were used for the instruction stream, address mapping and activity synchronization scheme.

Table I gives the results of the experiment. The speedup values are the ratios of the overall execution time of a single-thread version of PIC running on one processor to the overall execution time of a multithread PIC code running on two, three, and four processors. We found that a maximum speedup of three was attained when using four processors with four activities spawned for each multiple task.

Because the multithread PIC was not totally parallel (see Figure 1), the speedup for four processors may not indicate the full potential of the PIC algorithm. The times recorded and used for the parallel-processing speedup calculations were based on wall clock times, with timing runs made in a dedicated mode. Because of time constraints and limited resources, actual CPU times were not measured.

0190-3918/82/0000/0243$00.75 © 1982 IEEE

We conclude that significant computational speedups are strongly suggested by our results for multiple-processor environments similar to the UNIVAC 1100/80 computer system. We further note and caution that our results are highly coupled to the particular algorithm and the multiple-processor architecture selected.

References

[1] Morse, R. L., and C. W. Nielson. "One-, Two-, and Three-Dimensional Numerical Simulation of Two Beam Plasmas." Physical Review Letters 23 (10 November 1969), 19, pp. 1087-1090.

[2] Morse, R. L., and C. W. Nielson. "Numerical Simulation of Warm Two Beam Plasma." The Physics of Fluids 12 (November 1969), 11, pp. 2418-2425.

Acknowledgments

I want to thank the staff and management of the Computational Division of the Sandia National Laboratories, Albuquerque, for the use of their facilities and their assistance during the initial implementation and testing of the PIC code. In particular, I appreciate the help of David Hammer during this period. I also want to thank the Sperry UNIVAC group in Roseville, Minnesota, for their generous cooperation and for the use of their 1100/80 computer system upon which our results are based.

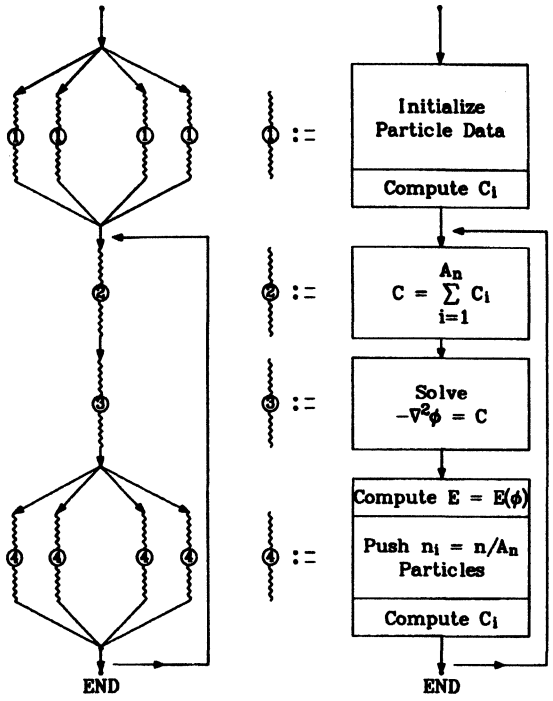


Figure 1. A multithread version of PIC as implemented on a UNIVAC System 1100/80 with two parallel-processing tasks (1 and 4), where A_n = total number of parallel activities (multithread), n = total number of particles, n_i = number of particles for activity i, C = total charge (distribution), and C_i = charge computed for activity i.

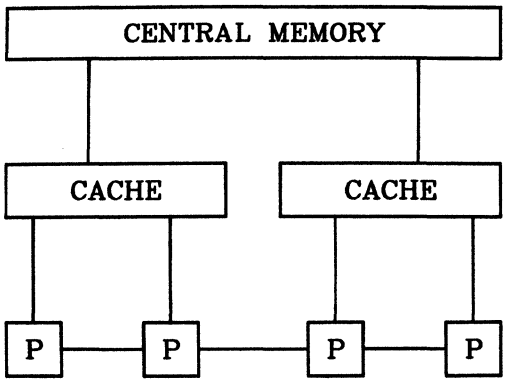


Figure 2. A simplified diagram of the UNIVAC 1100/80 with four processors P (designated 1100/84).

Number of Activities Per Parallel Task	Number of Processors	Average run Time (millisecond)	Speedup
1	1	102631	1
2	2	57110	1.80
3	3	42214	2.43
4	4	33263	3.09

Table I. Run times and speedups as a function of number of processors and number of activities for each parallel task spawned.

KERNEL-CONTROL TAILORING OF SEQUENTIAL PROGRAMS FOR PARALLEL EXECUTION

Mark Furtney
Babcock and Wilcox, Inc.
Lynchburg, Virginia 24505

Terrence W. Pratt
Department of Applied Mathematics and Computer Science
University of Virginia
Charlottesville, Virginia 22901

Abstract

Kernel-control tailoring is a method of preprocessing an ordinary sequential program for parallel execution. The preprocessing is intended to remove a substantial amount of the control dependence between operations in the program through deletion of conditional branches and unrolling of loops. The method is applicable to existing programs of practical size. Preliminary results from tailoring of a sample of FORTRAN programs are reported.

Introduction

There are three classes of dependence between operations that inhibit parallel execution: data dependence, in which one operation must wait for an operand to be computed as the result of another earlier operation; resource dependence, in which operations must wait on the availability of memory, functional units, or other resources; and control dependence, in which an operation must wait until it is known to be on the actual path of program execution before being executed. Various studies have indicated that the effect of control dependence in inhibiting parallel execution is a central problem. Riseman and Foster [1] isolated the problem in a classic study. For a sample of real programs run on an idealized machine (no resource dependence) they showed that programs with an average potential speedup of 51 to 1 (parallel over sequential execution), when only data dependence is considered, in fact could realize only an average speedup of less than 2 to 1, due to the effect of control dependence. The goal of this study is to obtain more extensive data on the magnitude of this control dependence in large scientific FORTRAN programs, and then to investigate a novel solution to the removal of some of this control dependence. The new method is called kernel-control tailoring and is based on the use of kernel-control decomposition (Pratt [2]) to preprocess the program to determine the control path, leaving a simplified program with less control dependence to be executed on the parallel computer.

This work was supported in part by NSF Grant MCS78-00763 and NASA Contract NAS1-16394 while the second author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, Virginia.

Control Dependence in FORTRAN Programs

The initial question of this study was to obtain better data on the amount of control dependence in a realistic sample of FORTRAN programs. The usual notion of basic block is useful. A basic block is a sequence of straight line code that includes no conditional branching, looping or subprogram calls. Within a basic block, execution of operations can be performed in parallel, inhibited only by data dependencies. Within a basic block, every operation essentially has the same control signal input, which is the control signal sent when the branch point preceding entry to the block is passed during execution. Large basic blocks in a program indicate relatively little control dependence (since all operations in a basic block are dependent on the same control signal); small basic blocks indicate much control dependence.

A sample of 44 production FORTRAN programs was analyzed, ranging in size from 2500 to 125,000 lines of code, all written for applications in nuclear and structural engineering. About 920,000 lines of code were analyzed; after deletion of comments, declarations, etc., about 390,000 FORTRAN statements remained to form the basis for the study. The size of the basic blocks was used as an appropriate indicator of the amount of control dependence. Each program was scanned and basic block sizes were tabulated (among a wealth of other statistics). The average basic block size was found to be 3.5 statements or 23 operations (using some rough approximations for operation counts). These figures substantiate that control dependence is indeed a serious problem, since not much useful parallelism can be expected within a block of only 23 operations. In essence, these results support what is clear from a visual analysis of typical sequential programs in almost any language: control structure, that is branching, looping, and subprogram calls, almost always breaks up a program into small basic blocks. This fragmentation makes most operations dependent on a control signal computed only slightly before the operation itself is to fire, regardless of the fact that its data operands may have been computed much earlier.

Various solutions have been proposed in the literature to remove control dependence in programs: prefetching and executing both arms of conditionals (see Riseman and Foster [1] and Magid, Tjaden, and Messinger [4]), the use of boolean variables to turn control dependence into

0190-3918/82/0000/0245$00.75 © 1982 IEEE

data dependence (see Padua, et al, [3]), and various loop transformations (Padua [3]).

Kernel-control decomposition

An alternative approach to removing control dependence is suggested by the theory of kernel-control decomposition (Pratt [2]). We first sketch the main theoretical results, then indicate why the theoretical potential is unrealizable in practice, and show an alternative approach that appears to bypass some of the practical difficulties.

Briefly summarized, the theory of kernel-control decomposition shows that any program may be decomposed into a control part, which is concerned only with determining the control path to be taken by the program, and a kernel part, which is concerned only with computing the output results of the program. The surprising result is that, in principle, control dependence can be completely eliminated from the actual computation of the program (the kernel) by performing this decomposition and then executing the control part separately first to determine the control path.

Practical Constraints on K-C Decomposition

The theoretical result of complete removal of control dependence is tempered by severe practical difficulties. The first problem is that in real programs, very few variables and statements are pure kernel or pure control; most participate in both kernel and control computations. In the decomposition, these variables and statements must be copied into both kernel and control parts. The second major difficulty lies in the identification of the kernel and control components of a program. To identify each component involves a process of backtracing control paths from output statements (to identify the kernel part) and from conditional branch points (to identify the control part). This is a nontrivial task in large practical programs. The result of studying the decomposition potential of several medium sized FORTRAN programs is that complete decomposition is impractical, primarily for these two reasons.

A Practical Approach

Analysis of the sample of FORTRAN programs suggests that an alternative to complete kernel-control decomposition might be practical. We observe that in these practical programs, there are in fact a subset of the variables that are global control variables whose values are set directly from input data, and that thereafter remain constant during execution. These variables represent the problem size parameters, output option choices, and various other important parameters during a run of the program. Actually these are not pure control variables in the theoretical sense, because inevitably their values are printed out during the course of the run, but ignoring this output (which serves only for documentation of the run) they serve only control purposes within the program.

In a typical FORTRAN program, these global control variables are part of a COMMON block. At the start of the program, their values are initialized from data provided by the user in setting the problem size parameters and options to be used. Subsequently these control values are tested repeatedly during program execution to control branching and also to control the number of iterations of loops, but their values do not enter directly into the computation of output results except in minor ways. Another important practical observation is that for many large production programs, such as nuclear reactor simulations, these global control variables are used to set problem size parameters and options that remain constant over many runs of the same program, e.g., for many runs of a large simulation.

We use the concepts from kernel-control decomposition to decompose the program, but only for a partial decomposition based solely on these global control variables. The method is termed kernel-control tailoring (or K-C tailoring) and may be outlined as follows:

1. Identify the global control variables of interest. This can be done automatically by scanning the source program, identifying "candidate" variables in COMMON blocks, and then deleting those variables that are assigned modified values in any subroutine, to get the final list of variables (making due allowance for possible coding tricks in FORTRAN that mask such assignments).

2. Identify the input values for these control variables (extract them from the input data).

3. Preprocess each routine in the program to find the conditional branching and looping that is controlled by these control variables only.

4. For each conditional branch found, determine the direction of the branch, using the known values for the control variables. Since the control values are invariant during execution, the direction of such a branch is always the same during execution. Thus the code down the branch not taken is dead code and can be deleted. Also delete the conditional branch statement itself (and replace it by a GOTO to the proper branch if necessary).

5. For each loop found, determine the number of iterations of the loop, using the known values for the control variables. The loop may now be unrolled completely to become straight line code with no testing for termination, or it may be partially unrolled in various ways. Nested loops can be transformed to unroll the loop with the smallest iteration count, etc.

6. Take the preprocessed program, which now has a substantial part of the control sequencing removed (ideally), and which also has a substantial amount of dead code eliminated (ideally), and process it for parallel processing in the usual way.

7. The preprocessed program may now be executed repeatedly for different data sets, provided the control variable input values remain

unchanged. Thus for a large production program, the cost of the preprocessing is potentially spread over more than a single execution of the program.

K-C tailoring is applicable to any ordinary sequential program (in FORTRAN or any other language) and produces a simplified program in the same language. The simplified program may then be optimized for parallel (or sequential) execution by the standard language processors, vectorizers, etc.

Effectiveness of the Approach

To determine the potential effectiveness of this technique, the data base of FORTRAN programs was used again. A typical program was taken from the sample, and the number of control variables determined. The amount of control sequencing determined by these control variables was analyzed. The program consisted of 51 subprograms that contained 1736 variables that were used for control purposes. Of these 133 were identified as the global control variables of interest. The program was instrumented to determine the use of these global control variables. About 20% of the conditional branching and 40% of the looping was found to be determined only by the values of these control variables. A subsequent analysis of a set of benchmark FORTRAN programs showed a large variation among programs both in the number of global control variables and in the amount of control sequencing affected by these variables. We have observed counts of 30% of the conditional branching and 50% of the looping controlled by global control variables in some programs, but ranging down to some programs with very few global control variables controlling less than 1% of the total number of run-time control decisions. A more refined set of measurements are currently being implemented. These preliminary results indicate that there do exist a substantial class of production programs for which K-C tailoring affects a nontrivial portion of the run-time control decisions.

Inner loops of critical routines are known to consume a large proportion of the computing time used by many large production programs. We are particularly interested, therefore, in the effect of K-C tailoring on the performance of inner loops. In the same reactor simulation program, a single routine was measured to use about 40% of the computing time in a typical run. The critical part of this routine is a triply nested DO loop. The loop parameters for each loop are global control variables (representing the size of the reactor core in three dimensions). Thus these critical loops are directly affected by K-C tailoring. To observe the possible effects from various forms of K-C tailoring that involve substitution of known control variable values followed by loop unrolling, we manually performed these transformations on this routine and measured the speedup. When the inner loop alone was unrolled, a speedup of 1.3 to 1 was measured. When the two outer loops were unrolled (leaving multiple copies of the small inner loop), a speedup of 2.1 to 1 was observed. Both these transformations may easily be made automatically during K-C tailoring. Finally a more complex transformation was performed that changed the nested I-J-K loops into a K-I-J nest, followed by unrolling of the two inner loops (leaving one large outer loop). This transformation led to a measured speedup of the loop alone of 3.2 to 1, but since some array transposition was required, the overall speedup was 2.4 to 1. Complete unrolling of the entire loop nest was considered impractical due to the code expansion involved. Dongarra and Hinds [5] show similar results from studies of the performance improvements due to loop unrolling in FORTRAN programs.

Conclusion

Kernel-control tailoring of a sequential program can remove a substantial amount of control dependence by using a simple preprocessing of static code. After the K-C tailoring is complete, the program may be more effectively transformed and optimized by existing methods, such as those of Kuck, et al. [3] for parallel and vector machines and those of ordinary global optimizers for sequential computers. Thus K-C tailoring appears promising as an adjunct to existing methods of optimization.

The theory of K-C decomposition suggests that there is a range of options available in removal of control dependence through these techniques, ranging from the straightforward static preprocessing proposed here, through levels of increasingly sophisticated (and costly) separations of control and kernel parts based on dynamic execution of a partial control part, to complete decomposition. A deeper understanding of this range of options may lead to additional practical methods for K-C tailoring.

References

[1] Riseman, E. and Foster, C., "The inhibition of potential parallelism by conditional jumps," IEEE Trans. Comput, C-21, No. 12, December 1972, pp. 1405-1411.

[2] Pratt, T., "Program analysis and optimization through kernel-control decomposition," Acta Inform, Vol. 9, No. 3, 1978, pp. 195-216.

[3] Padua, D., Kuck, D. and Lawrie, D., "High-speed multiprocessors and compilation techniques," IEEE Trans. Comput, C-29, No. 9, September 1980., pp. 763-776.

[4] Magid, N., Tjaden, G. and Messinger, H., "Exploitation of concurrency by virtual elimination of branch instructions," Int. Conf. on Par. Proc., 1981, pp. 164-165.

[5] Dongarra, J.J. and Hinds, A.R., "Unrolling loops in FORTRAN," Software - Practice and Experience, 9, (1979), pp. 219-226.

A PERFORMANCE MODEL FOR INSTRUCTION PREFETCH IN PIPELINED INSTRUCTION UNITS

Gregory F. Grohoski
IBM Corporation
T. J. Watson Research Center
Yorktown Heights, NY 10598

Janak H. Patel
Coordinated Science Laboratory
University of Illinois
Urbana, IL 61801

Abstract

This paper presents two models of an instruction execution pipeline incorporating an instruction prefetching strategy. One model ignores operand accessing and thus models a computer system with separate program and data memories. The second model accounts for operand accessing. The throughput and memory traffic of the prefetch strategy are analyzed based upon run statistics derived from trace tapes of programs compiled for the IBM 360/370 architecture.

1. Introduction

Pipelined processors frequently utilize instruction prefetching to supply instructions to the pipeline without delay. Little work has been published which analyzes instruction prefetching at the instruction word level in pipelined computers. Yet an understanding of instruction prefetching is crucial to the design of high performance computers when it is not economical to use a large, fast memory, or when available memory technology is not fast enough to support the rate of instruction execution desired.

A prefetching strategy can be stated as follows. Instruction words ahead of the one currently being decoded are fetched from the memory before the instruction decoding unit requests them. Thus, the memory access time of an instruction word is masked by the execution of previously fetched instructions. If the instruction stream being executed was purely sequential, by fetching instructions early enough, no instructions past the first one executed would see any delay, and the memory would experience no more requests for instruction words than if no prefetching was being performed.

Successful branches disturb the sequentiality of the instruction stream. A program consists of unconditional branch instructions, conditional branch instructions, and non-branch instructions. Conditional branch instructions result in a transfer of control only if the condition they are testing is true, so conditional branch instructions may be successful or unsuccessful. Unconditional branch, or jump, instructions always result in a transfer of control. Therefore, they are always successful.

One of the first analyses of instruction prefetching was done by Rau[1,2]. His analysis does not account for the effect of operand accessing upon instruction prefetching.

Acknowledgements: This research was supported by the Naval Electronics Systems Command under VHSIC contract N00039-80-C-0556.

2. Analytical Model

The prefetch strategy of the model requires the hardware diagrammed in Figure 1. It is assumed that the memory is interleaved and can accept requests at a maximum rate of one request per cycle. A cycle is simply the smallest unit of time which the model is aware of and corresponds to a hardware clock cycle. All requests require T cycles to return from memory. There are two prefetch buffers, of sizes s and t instruction words. The s-size buffer holds instructions fetched during the sequential part of a run. When a branch is successful, the entire buffer is invalidated. The other buffer holds instructions fetched from the target of a conditional branch. Similarly, when a conditional branch is resolved and determined to be unsuccessful, the contents of this buffer are invalidated. The non-pipelined instruction decoding unit requests instruction words at a maximum rate of one word per r cycles. If the instruction requested by the decoder is available in the sequential buffer for sequential instructions, or is in the target buffer if a conditional branch has just been resolved, and is successful, it enters the decoder with zero delay. Otherwise, the decoder is idle until the instruction returns from memory. After r cycles, the instruction type is known.

The program is assumed to begin with a jump instruction. For the first 1+s cycles, 1 memory request for an instruction is issued each cycle, to fill up the sequential prefetch buffer. Thereafter, one request is issued every r cycles until a conditional branch or jump instruction is decoded.

Except for jump instructions, all decoded instructions enter the execution pipeline, where E units are required to complete execution. If the decoded instruction is an unconditional branch the instruction word at the target of the jump is requested immediately by the decoder, and decoding ceases until the target instruction returns from the memory. The pipeline will see the full memory latency time, T, since there was no opportunity for target prefetching.

If the decoded instruction is a conditional branch, sequential prefetching is suspended during the E cycles it is being executed. The instruction simultaneously enters the execution pipeline, but no more instructions are decoded until the branch is resolved at the end of E units. Instructions are prefetched from the target memory address of the conditional branch instruction. Requests for t target instructions are issued at the rate of one per cycle. Once the branch is resolved, target prefetching becomes unnecessary.

If the branch is successful, the target instruction stream becomes the sequential stream, and instructions are requested every r units from

0190-3918/82/0000/0248$00.75 © 1982 IEEE

this sequential stream. Execution of this new stream begins when the target of the branch returns from memory, or whenever E units have elapsed, whichever is later.

If the branch is unsuccessful, instruction requests are initiated every r units of time following the branch resolution and continue until the next branch or jump is decoded.

A list of the parameters necessary to describe the model appears in Table 1. All intervals on the following time diagrams are assumed to be closed on the left and open on the right.

A run consists of all instructions executed following the decoding of a successful branch instruction, and terminating with the decoding of another successful branch instruction. There are only two run types: those that begin with an unconditional branch, and those that begin with a successful conditional branch. Figure 2 illustrates the two run types and details the instructions and the execution times that are counted as part of the runs. A run whose first instruction is the target instruction of an unconditional branch will be referred to as a u-run, while a run whose first instruction is the target of a successful conditional branch will be referred to as a cs-run.

Figure 3a is a time diagram of the execution of a u-run. An unsuccessful conditional branch occurs during the run. Instants when memory requests are submitted for sequential instructions are marked with an asterisk, while instants when target requests are submitted to the memory are marked with a '+'. Figure 3b diagrams the execution of a cs-run.

The performance measures used to evaluate the prefetch strategy are the throughput of the execution pipeline and the memory traffic the strategy generates. The throughput is defined to be the number of instructions executed divided by the execution time of the program, and the memory traffic is defined to be the number of memory requests generated divided by the execution time of the program. The throughput is bounded by 1/r; memory traffic is bounded by one.

2.1 Analysis of the non-operand fetching case

At the beginning of every run the condition of the prefetch buffers is known. Therefore, the prefetching and execution behavior of a program can be reconstructed by breaking the instruction stream up into runs, and analyzing the performance of the prefetch scheme for each run case. For arbitrary values of the parameters the analysis becomes complex and amounts to simulation of the run. By restricting the parameter space, analytical results are obtained. We do not present the lengthy derivations of the results; for detailed derivations the interested reader is referred to [3].

Let the interbranch distance be defined as the number of instructions from one branch instruction to the next, excluding the first branch, but including the second. Let the ith such interval in a run be denoted by l_i. Thus, referring to Figure 4, $l_1 = 2$ is the number of instructions from the beginning of the run to the first branch, which is an unsuccessful conditional branch.

An unsuccessful conditional branch removes memory cycles from sequential prefetching. Therefore, after the branch is resolved, the number of instructions in the sequential buffer may not be sufficient to support the maximum rate of execution, and the decoder may have to wait for instructions to return from the memory. Thus an unsuccessful conditional branch, after its resolution, adds a delay to the execution time of a run.

Let z_i be the number of instructions in the sequential buffer after the resolution of the ith unsuccessful conditional branch in a run, and let d_i be the delay added to the execution time of a run due to the ith unsuccessful conditional branch, that is, due to an insufficient number of instructions in the sequential prefetch buffer. For convenience we define z_0 to be the number of instructions in the sequential prefetch buffer at the start of the run and d_0 to be the delay incurred at the start of the run. Then, for a run with k intervening unsuccessful conditional branches, we have the following:

$$\text{Number of instructions executed in any run} = \sum_{i=1}^{k} l_i$$

Non-operand fetching results; $0 < s < T < E + 1$

Unconditional Run:

$$z_0 = 0;\ d_0 = T;\ z_i = s;\ 1 \le i \le k; \tag{1}$$

$$d_i = \begin{cases} 0, & l_{i+1} \le z_i \\ \max(0, T - r z_i), & l_{i+1} > z_i \end{cases} \quad 1 \le i \le k \tag{2}$$

$$\text{execution time of run} = kE + \sum_{i=1}^{k+1} r l_i + \sum_{i=1}^{k} d_i \tag{3}$$

$$\text{number of memory requests} = kt + \sum_{i=1}^{k+1} l_i + \sum_{i=1}^{k} \left\lceil \frac{d_i}{r} \right\rceil + s + \left\lceil \frac{T-s}{r} \right\rceil \tag{4}$$

Conditional successful run:

$$z_0 = t; \tag{5}$$

$$d_0 = \max(0, T - E) + \max(0, E + T - tr - \max(T, E)) \tag{6}$$

$$z_i = \min\left(s, z_{i-1} + \left\lceil \frac{d_{i-1}}{r} \right\rceil\right) \tag{7}$$

d_i = same as eq. (2)
execution time of run = E + eq. (3) (8)

$$\text{number of memory requests} = (k+1)t + \sum_{i=1}^{k+1} l_i + \sum_{i=0}^{k} \left\lceil \frac{d_i}{r} \right\rceil \tag{9}$$

2.2 Effect of operand accessing

Operand accesses interfere with instruction prefetching when program and data memory are not separate. To prevent execution pipeline delays, operand fetches must preempt any instruction fetches which would otherwise occur. Since memory cycles are stolen from the prefetch mechanism, instructions might not be prefetched in time to avoid decoding delays. The terms requiring an operand and operand accessing refer to both operand fetching and storing.

We assume that an instruction which requires an operand may require only one operand and may not

be a branch instruction. After r units the decoder will realize that the instruction requires an operand. At this point, any instruction prefetching which would have occurred is preempted and a memory request is issued for the operand. Simultaneously, the instruction enters the execution pipeline. For the instruction to make use of the operand while it is in the pipeline, the operand must return prior to E units after it entered the pipeline. We therefore assume

$$0 \le s < T \le E - 1$$

for the remainder of Section 2.2.

We can model operand accessing by introducing p, the probability that an instruction requires an operand, given that it is not a branch or jump instruction. This probability can be estimated from program trace tapes. Furthermore, we will set r=1, since this presents the most reasonable case. Equations for u-runs and cs-runs are presented.

Unconditional Run:

$$z_0 = 0;\ d_i = \text{same as equs. (1) and (2)} \tag{10}$$

$$z_i = \min\,(s, z_{i-1} + d_{i-1} + (1-p)(l_i - 1) + 1 - l_i + p^{*}max(0, l_i - 1 - T));\quad 1 \le i \le k \tag{11}$$

execution time of run =

$$kE + \sum_{i=1}^{k+1} [l_i + d_{i-1} + p^{*}max(0, l_i - 1 - T)] \tag{12}$$

number of instruction requests =

$$kt + \sum_{i=1}^{k+1} [d_{i-1} + 1 + (1-p)(l_i - 1) + p^{*}max(0, l_i - 1 - T)] \tag{13}$$

$$\text{number of operand requests} = p\sum_{i=1}^{k+1} (l_i - 1) \tag{14}$$

Conditional successful run:

$$z_0 = t;\ z_i = \text{same as eq. (11)} \tag{15}$$

$$d_i = \text{same as eqs. (1) and (2)} \tag{16}$$

$$\text{execution time of run} = E + \text{eq.(12)} \tag{17}$$

$$\text{number of instruction requests} = t + \text{eq.(13)} \tag{18}$$

$$\text{number of operand requests} = t + \text{eq.(14)} \tag{19}$$

3. Results

Since our analysis is not exact and also since it is not valid for certain ranges of parameters, a trace-driven simulator was used to verify the analytical results. Program traces were broken up into the two run types and runs were further classified by the number and spacing of intervening unsuccessful conditional branches. The frequency of occurrence of each run was used to weigh the execution time and memory request equations presented in Sections 2.1 and 2.2.

Two program traces were analyzed. These programs were compiled for the IBM 360/370 architecture. One program, GAUSS, is a FORTRAN execution of a Gaussian elimination program. The other program, SLIST, is a trace of a PL/I list processing program. Table 2 lists some of the important statistics of each program.

For the non-operand fetching case for T <= E + 1, simulation results differed from analytical results by at most 2% for GAUSS and at most 4% for SLIST. The analytical model is only approximately valid for T > E + 1. For this T for GAUSS, the simulation results were still within 2% of the analytical model; however, for SLIST, the error was less than 9%. The error results from the treatment of unsuccessful conditional branches in our derivation. Note in Table 2 that the number of of unsuccessful conditional branches in SLIST is about 4 times that of GAUSS. We have included the performance of the non-prefetching case. For this case we assume that an instruction fetch request is issued to memory after r units for non-branch and jump instructions and after r+E units for conditional branches. Figure 5a shows the throughput as a function of target prefetch buffer size. Throughput saturates when t=3. For the parameters shown, this is to be expected. A maximum delay of T cycles may occur following the resolution of a successful conditional branch. To overcome this delay, enough instructions must be in the buffer or must have been requested such that the decoder can run without delay for T cycles. Loosely, then, t should be chosen such that rt=T to eliminate the delay cycles. This means that choosing t=3 will give the maximum throughput for a given s. Increasing the sequential buffer size effectively scales the throughput, and, as for t, s>3 does not enhance the performance. This is similar to target prefetch buffer size saturation.

Figure 5b plots the memory traffic versus t for the same trace, SLIST, of Figure 5a. The traffic is not nearly as sensitive to s as the throughput is. This occurs because increasing s with t fixed has two compensatory effects. On the one hand, increasing s increases the number of sequential requests generated immediately following the decoding of an unconditional branch. On the other hand, increasing s reduces the d-delays immediately following the resolution of unsuccessful conditional branch instructions. Since sequential requests are generated during these delay cycles, increasing s decreases the number of these requests which are generated.

Increasing t with s fixed also has two compensatory effects. As t increases, the number of requests for instructions at the target address of all conditional branch instructions increases. As t increases, the execution time of cs-runs decrease, because increasing t reduces the delay which follows the resolution of a successful conditional branch instruction. For both SLIST and GAUSS, the number of conditional branches is sufficiently greater than the number of successful conditional branches to cause the memory traffic to increase as t increases with s fixed.

Results are not presented for GAUSS, since the effect of s>1 on throughput and traffic is negligible. This is due to the predominance of successful conditional branches (11.9%) combined with the relative lack of unsuccessful conditional branches (2.1%). Since the frequency of successful conditional branches in SLIST and GAUSS is about the same (10.2% and 11.9%, respectively), the effect of t is similar in both cases.

Finally, we present the effect of operand accessing on performance. Figure 6a plots throughput versus the degree of target prefetch for s=1 and s=3, for both the model and simulation for SLIST. The p used for the model was 0.159, which is derived from the trace of SLIST. Thus, a meaningful comparison can be made between the

simulation and the model. The operand accessing model is slightly optimistic. Figure 6b plots the traffic for the simulator and the model. For this combination of parameters, the operand fetching model yields results within 8% of the simulation.

4. Concluding Remarks

Unsuccessful conditional branches cause delays resulting from an insufficient sequential buffer size to appear. They also enable operand accesses to cause delays which can not be overcome by increasing the size of the sequential prefetch buffer. Programs which have few unsuccessful conditional branches will not suffer from these problems.

While prefetching from the target of conditional branch instructions reduces delays for successful conditional branches, it substantially increases the memory traffic. Increasing the size of the sequential buffer may increase or decrease the traffic, depending upon the parameters and program under consideration.

Since conditional branches are much more frequent than unconditional branches, significant improvements in performance require a predictive prefetch strategy. In particular, based upon the detailed run analyses performed to evaluate the equations of the model, some of Smith's[4] strategies may work well.

References

[1] B. R. Rau and G. Rossman, "The Effect of Instruction Fetch Strategies Upon the Performance of Pipelined Instruction Units," Proceedings of the Fourth Annual Symposium on Computer Architecture, 1977, pp. 80-89.

[2] B. R. Rau, "Sequential Prefetch Strategies for Instructions and Data," Stanford Electronics Laboratory Technical Report No. 131, Stanford University, January, 1977.

[3] G. F. Grohoski, "An Instruction Prefetch Model for Pipelined Execution Units," Tech. Rept. R-913, Coordinated Science Laboratory, University of Illinois, Urbana, Aug. 1981.

[4] J. E. Smith, "A Study of Branch Prediction Strategies," Proceedings of the Eighth Annual Symposium on Computer Architecture, 1981, pp. 135-148.

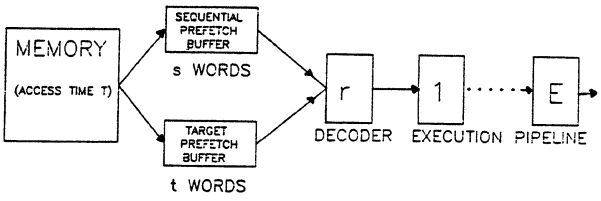


Figure 1. Instruction Pipeline

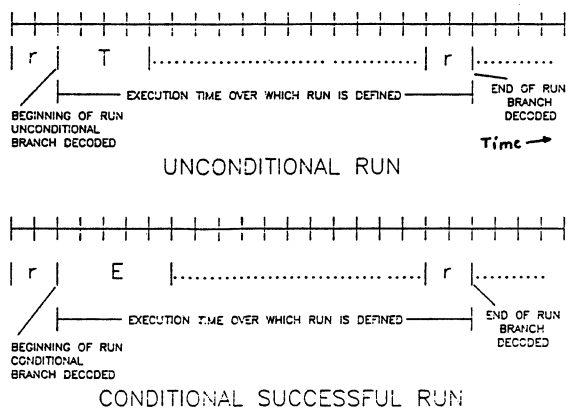


Figure 2. Possible Run Types.

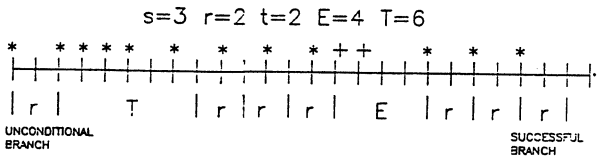


Figure 3a. Unconditional Run.

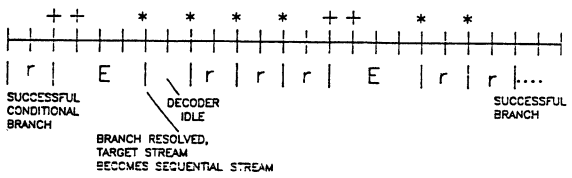


Figure 3b. Conditional Successful Run.

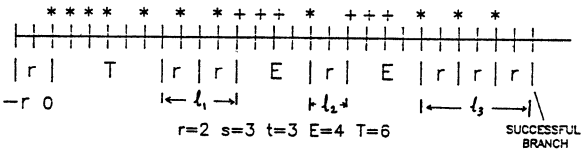


Figure 4. U-run with two intervening unsuccessful conditional branches.

Table 1. Model Parameters

Parameter	Description	Range
r	Instruction decode time	>=1 cycles
s	Size of sequential prefetch buffer	>=0 words
t	Size of target prefetch buffer	>=0 words
E	Execution pipeline length	>=1 segments
T	Memory access time	>=1 cycles
p	Probability an instruction requires an operand	

	# instr.	# unc.	%	# cs.	#	# cu.	%	# op.	p
SLIST	69935	2007	2.9	7127	10.2	5773	8.3	8765	.1590
GAUSS	63611	33	.05	7535	11.9	1337	2.1	449	.0082

Table 2. Program Statistics.

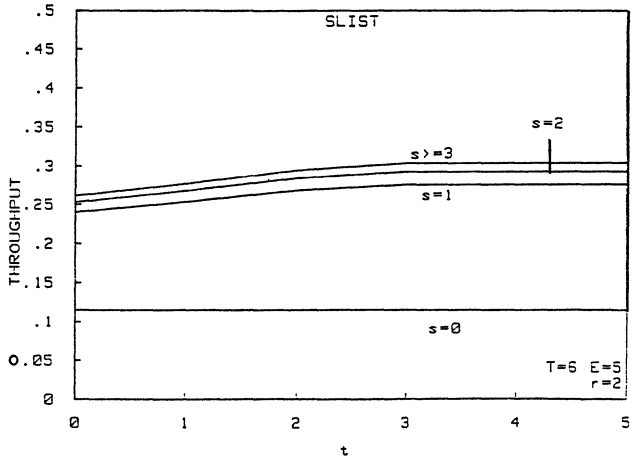


Figure 5a. Throughput vs. target prefetch buffer size.

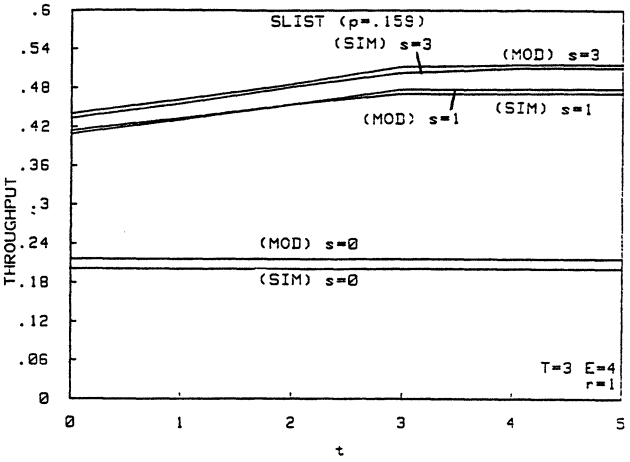


Figure 6a. Throughput comparison of simulation and operand accessing model.

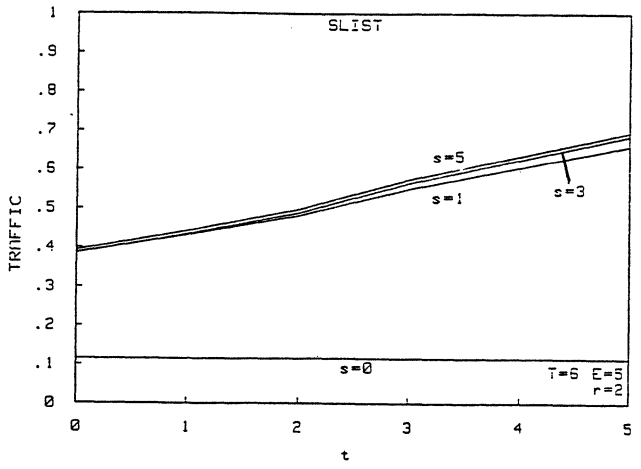


Figure 5b. Memory traffic vs. target prefetch buffer size.

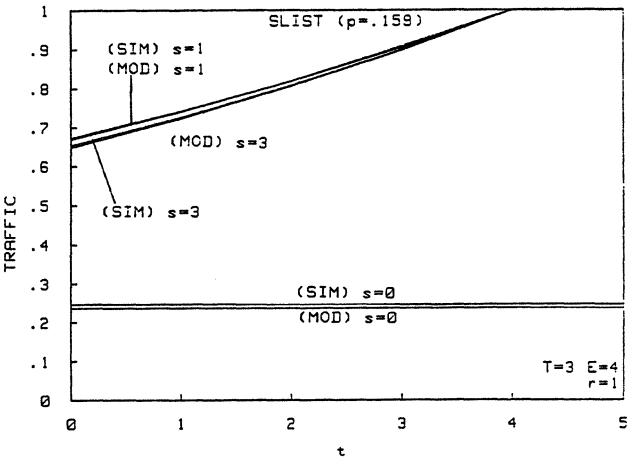


Figure 6b. Traffic comparison of simulation and operand accessing model.

PROGRAMMING TECHNIQUES ON THE LUCAS ASSOCIATIVE ARRAY COMPUTER *

Christer Fernström

Department of
Computer Engineering
Lund University
P.O. Box 725
220 07 LUND, SWEDEN

Abstract

LUCAS (Lund University Content Addressable System) is an associative array computer in the SIMD (Single Instruction - Multiple Data) category. This paper describes the programming of the machine on different levels and introduces the software tools that are used. A high level language that takes full advantage of the architecture and yet allows powerful manipulation of data on an algorithmic level, is presented. Programming examples show the use of the language in signal processing applications and data base management.

1. Introduction

An associative array processor is an associative memory where each memory word contains its own processing element and where a communication network is defined between the words. The organization is of type SIMD (Single Instruction - Multiple Data), meaning that the same instruction is executed in parallel on different data. An important feature with associative array processors is the possibility to search the memory contents in parallel. The result of the search (contents in words satisfies/does not satisfy the search criterion) is marked in an activation register, called the tag register, in each word. Subsequent parallel operations may be limited to the words where the tag register contains the value 'true'.

The interest in associative array computers has been steadily increasing during the last two decades. It has been possible to add the complexity needed without paying the penalty of much higher costs. The regular structure in this kind of machines is very well suited for implementation in VLSI, and as VLSI design techniques become more widespread, the interest will continue to grow.

Since the introduction of the first commercially available associative computer, STARAN in 1973, several general purpose high level languages for programming parallel array computers have been proposed. Some of these can be said to be true general purpose languages in that no special application and no special machine has guided the design [1-4], while others are oriented towards an existing computer such as STARAN [5], DAP [6], and ILLIAC IV [7-9]. Important application areas for associative array computers are image processing and data base management. Special purpose languages for these applications have also been designed [10,11].

LUCAS (Lund University Content Addressable System) is an associative array processor currently under development at the University of Lund, Sweden. The working mode is bit serial and the machine is constructed from off-the-shelf components. This paper describes different programming levels on the machine and includes a description of a high level language defined as an extension to Pascal.

2. The Architecture of LUCAS

LUCAS consists of two parts: a Control Unit and an Associative Array. By a simple interface it is attached to a host computer, at present a PROLOG Z80 microcomputer system with CP/M operating system. The host computer sends instructions to the Control Unit and handles input and output of data (fig 1). Alternatively, input and output can be handled by a dedicated I/O processor that directly communicates with a fast secondary memory. A detailed description of the design is given in [12].

2.1 The Associative Array

The Associative Array (fig 2) consists of 256 words that are interconnected by a powerful interconnection network. The design of one word is depicted in fig 3.

The word length is 4096 bits. Data is accessed one bit at a time, pointed to by a 12-bit address from the Control Unit. A processing element for bit-serial computations is also included in each word.

The processing element comprises an ALU and four one-bit result registers: T (tag), R (result), C (carry) and X (auxiliary). The ALU performs operations on the five one-bit arguments M, T, R, C and X. The M input can recieve data from either the corresponding memory word or from another word via the interconnection network. In this way data can be moved up and down the array or interchanged according to different communication patterns, e.g. the perfect shuffle.

The tag register is used as an activation control for the word since it enables the write signal to the memory. This mechanism is of the highest im-

*This research was in part supported by the National Swedish Board for Technical Development under grants 79-3770 and 81-4606 at the University of Lund.

0190-3918/82/0000/0253$00.75 © 1982 IEEE

portance and is in fact the key to associative array computing. A parallel search selects certain words by setting their tag registers according to the results of the search. Then follows computation in parallel on selected words.

Input and output of data is done via a shift register which is directly accessed from both LUCAS and the host (or the I/O processor).

2.2 The Control Unit

The Control Unit (fig 4) accepts instructions from the host computer and executes these in the form of microprograms. An instruction to LUCAS can include up to four parameters - ususally describing the locations and the lengths of the operands.

An important part of the Control Unit is the Address Processor. It performs fast computations of addresses to the Associative Array (increment, decrement, add constant etc). The Address Processor contains several index registers and a data stack. Furthermore the Control Unit contains a Common Register which is used for parallel searching in the Associative Array and for operations between one dimensional and parallel data (e.g. add the same value to data in every word of the array).

An example shows the interaction between LUCAS and the host:

Given an n x m dimensional matrix M and an m dimensional vector V, calculate the n dimensional vector X = M x V.
The elements of V are stored in the Common Register and the elements of M in the Associative Array. Space is reserved for X in the Associative Array.

Common	V[1]	...	V[m]		
Word 1	M[1,1]	...	M[1,m]	X[1]	s[1]
Word 2	M[2,1]	...	M[2,m]	X[2]	s[2]
...					
Word n	M[n,1]	...	M[n,m]	X[n]	s[n]
	<---------->			<-->	<-->
	area in the Associative Array that is reserved for the matrix M			area reserved for X	scratch pad area

Actions performed on the host.	Actions performed on LUCAS.
1 Specify operation CLEAR FIELD to LUCAS with parameter indicating the X vector	
2	Execute CLEAR FIELD operation
3 Specify operation to select the n first words	
4	Select the n first words by putting a one in the tags
5 set j=1	
6 calculate the address to the j:th elements in V and M	
7 Specify the operation MULTIPLY COMMON with parameters indicating the j:th elements in V and M and the scratch pad area	
8	Execute the MULTIPLY COMMON operation, leaving the result in the scratch pad area
9 Specify operation ADD FIELDS with parameters indicating the X vector and the scratch pad area	
10	Execute ADD FIELDS operation
11 j:=j+1	
12 if j <=m then go to 6	
13 exit	

As can be seen in the example, the host keeps track of the variables in the Associative Array, while all the actual calculations are done in parallel in LUCAS. This is the normal interaction between LUCAS and the host.

The interface is designed so as to let the two actions overlap.

3. The software structure

Great effort has been put into making LUCAS flexible. Mainly intended as a research machine, different aspects of array computing have been and will be exploited. This has led to a design which is programmable on several levels.

3.1 Micro Programming

The LUCAS machine instruction set, which constitutes the software interface to the host computer, is defined as a set of microprograms, executed by the control unit. Prior to the execution of an application program, the microprogram memory can be loaded from the host. Machine instructions may range from single search operations up to compound operations such as matrix multiplications or operations on data base relations.

In each clock cycle a total of 80 control signals are sent from the Control Unit to various parts of LUCAS. The control signals can be divided into five groups: host communication (6), microprogram flow control (32), Associative Array control (16), Address Processor control (26) and one group for user defined auxiliary functions (2).

Fig 3 shows a processing element in the Associative Array. As can be seen in the picture, the ALU has six inputs and four outputs. The function performed in the ALU is controlled by a five bit code. This means that only 32 of the $4*2^{64}$ possi-

ble functions in the ALU can be performed. Since all computations are done bit-serially, it is extremely important that the ALU instruction set is well suited for the kind of calculations done, to avoid unnecessary overhead,

The implementation of the ALU, in the form of a user programmable read only memory, allows optimization of its instruction set for different applications. In signal processing applications, the instruction set would be strongly oriented towards arithmetic calculations, while in data base management applications the transfer of bits between the four registers and the memory, as well as boolean functions such as AND, OR etc would be chosen.

3.2 Interaction with sequential programs on the host

To avoid dependency upon the choice of the host computer, nearly all the system software developed has been written in Pascal. For the same reason it was decided that application programs also should be written in high level language. A library of Pascal procedures that interacts with LUCAS are incorporated in the system library.

It is interesting to note that the penalty for using a high level language, as compared to assembly language, in terms of speed, is remarkably small. Especially in applications with heavy computations. This is due to the fact that most of the sequential operations performed are housekeeping operations and can overlap in time with computations in LUCAS.

Assembly programming is kept to a minimum, and in fact less than 50 bytes of machine code is needed to set up the software interface with LUCAS. This means that all the software developed is completely transportable between different hosts.

4. A high level programming language

The rest of this paper is a description of a high level language that is currently being implemented on LUCAS. We start the presentation with a brief discussion of what we want to obtain followed by an informal description of the language elements. In the subsequent sections each element is described in detail and the use of the language is presented in the form of programming examples. Finally a complete description of the syntax is given.

4.1 Primary Considerations

There are two different approaches to the design of high level languages for parallel computers:

- the parallelism of the computer has a correspondence in the syntax of the language.
- the syntax of the language does not contain any primitives for parallel computations, but the compiler tries to detect inherent parallelism in the sequential program and to generate code for the parallel computer.

Our view is that the parallelism should be apparent in the language. If the parallel structure of the computer has no correspondence in the language, the user will be less motivated to design his algorithms in a way which involves parallel computation. It is unlikely that an algorithm for a sequential computer could easily be transformed to fit into a parallel machine. This not only calls for unnecessary complications in the compiler, but also leads to less efficient use of the parallel computer.

4.2 Guidelines for the design

The following requirements for the language are stated:

- The language should only include constructs that can be efficiently implemented on LUCAS.
- The language should be functionally complete in the sense that all possible algorithms for LUCAS can be expressed in it.
- The language should be suitable for and emphasize the use of structured programming.
- The language should be kept small and be fairly easy to implement.

The use of an existing sequential programming language as a base for the new language would have many advantages:

- The sequential language has a well defined syntax.
- Implementation is simpler since existing compilers can be modified to accept the new language.
- The user needs to learn relatively few new concepts.

4.3 Description of the language

We decided that the language should be an extension to Pascal and it is referred to as Pascal/L. The choice of Pascal meets the two last requirements stated above, provided that the extensions are chosen carefully.

Pascal is a well structured language with strong typing of variables. Its structure allows a large amount of error detection both at compile time and at run time. Compilers for Pascal are relatively simple to implement. The syntax has been chosen so that only one symbol lookahead is needed, enabling the use of a simple parsing technique. To facilitate the code generation and to make the compilers more portable, an implementation scheme with code generation for a stack oriented virtual machine is used. The existence of portable compilers, often written in Pascal, simplifies its implementation on different machines.

The following extensions to Pascal are defined:

- Declaration of variables that will be allocated in the Associative Array. In the following these will be referred to as parallel variables.
- An indexing scheme to access parts of parallel variables.
- Expressions involving parallel variables.
- An extended control structure, allowing the use

of parallel variables as control variables.

- Standard functions for alignment of parallel variables.
- Input and output of parallel variables.

In the text, the words 'scalar' or 'sequential variable' stand for variables that are allocated in the host computer memory, while a 'parallel variable' is allocated in the Associative Array.

Appendix A contains a syntax summary of the language.

4.3.1 Data Declaration

Parallel variables are characterized by their dimension and their range. The dimension of a parallel variable refers to the number of subscripts in the declaration. The range, which can be seen as a measure of the parallelism, is given by the size of the first dimension.

The linear organization of the LUCAS Associative Array makes it especially suited for operations on one- and two-dimensional arrays. In principle, arrays of any dimension can be represented in LUCAS. However the natural storing scheme for one- and two-dimensional arrays, where adjoining array elements also are physical neighbours, will be lost. In this description of Pascal/L we are therefore only concerned with arrays of one and two dimensions, even though the final definition of the language probably will include arrays of higher dimensions.

There are two kinds of parallel variables:

selector and parallel array.

A selector is defined as a boolean bit vector intended to limit the parallelism of the operations in the Associative Array. At execution time this is accomplished by setting the tag registers in those memory words where the corresponding selector element has the value true. When a selector is declared, it can optionally be initialized to any value.

```
<selector type> ::= selector [constant..constant]|
     selector [constant..constant] := <boolean
                                       aggregate>

<boolean aggregate> ::= <choice> => <boolean value>|
     <choice> => <boolean value> , others =>
                                     <boolean value>

<choice> ::= constant | constant..constant |
     constant..constant step constant

<boolean value> ::= true | false
```

Examples:

```
var   a :  selector[0..255];

var  a : selector[0..255]:=(0,1,5=>true, others=
                                        =>false>;

var a : selector[0..199]:=(0..126 step 2=>true,
                                    others=>false>;
```

In the declaration of a parallel array the first dimension specifies the maximum range of parallelism for the variable.

```
var para : parallel array[0..99,0..2] of integer;
```

declares the variable para to be defined in the words 0 to 99 in the Associative Array. Each word contains three elements of para.

```
<parallel array type> ::=
     parallel array [constant..constant]
                              of <parallel component
                                  type> |
     parallel array [constant..constant , constant
                                       ..constant]
                              of <parallel component
                                  type>

<parallel component type> ::= <parallel type>

<parallel type> ::= <parallel type identifier> |
     <parallel standard type> | record <parallel
                                field list> end

<parallel type identifier> ::= <identifier>

<parallel standard type> ::= integer | real | boo-
                             lean | char |
                             string[constant]

<parallel field list> ::= <parallel record section>
                          { ; <parallel record
                              section> }

<parallel record section> ::=
     <field identifier> { , <field identifier> } :
                          <parallel standard
                           type>
```

Example:

```
var parrec : parallel array[0..99] of record
                                     a,b : integer;
                                     c  :  real
                                         end;
```

4.3.2 Indexing

When operating on a parallel variable it is possible to reference several elements along its parallel dimension at the same time. This set of elements is referred to as the range of parallelism for the operation. If the index is omitted, the complete array is referenced.

```
<parallel indexed variable> ::=
     <parallel array variable> [ <first index> ] |
     <parallel array variable> [ <first index>,
                                 <expression> ]

<parallel array variable> ::= <variable>

<first index> ::= <selector expression> | * |
                  constant | constant..constant

<selector expression> ::= <expression>
```

Examples:

para[*,0]	Selects column 0 of para. Para is a two dimensional parallel variable.
para[a,0]	Where a is a selector, selects a subset of column 0 in para.
para[2..4,0]	Selects elements 2,3 and 4 of column 0 in para.

4.3.3 Expressions and Assignment

It is possible to combine sequential and parallel variables in expressions as long as no type conflict occurs. An expression that contains parallel variables always results in a parallel value (except for some standard functions that take a parallel variable as an argument and yield a scalar value). The meaning of expressions such as:

```
4 * para   or   para > 4
```

where para is an array, is that the scalar is combined with each element of the parallel variable.

There are four kinds of assignment statements:

1) Left side and right side are scalars.
   This is the normal Pascal assignment statement.
2) Left side is parallel and right side is scalar.
   All elements within the range of the left side variable are assigned the value of the scalar expression.
3) Left side is scalar and right side is parallel.
   The right hand side of the assignment should be a parallel variable indexed so that only one element is selected.
4) Left side and right side are parallel.
   The elements within the range of the left hand side variable are assigned the corresponding values of the right hand side expression. The range of the expression must be equal to, or overlap, the range of the left hand side variable.

In expressions, all parallel variables must have the same range, otherwise a run time error occurs.

The following program gives an example of different kinds of assignment.

```
Program Assign;
var odd : selector[0..255]:=(1..255 step 2=>
            true, others=>false);
    even, sel : selector[0..255];
    p1,p2 : parallel array[0..255] of integer;
    i : integer;
begin
  even:=not odd;    (* Both sides parallel. The
       .               same range *)
       .
  p1[even]:=p2*2;   (* Both sides parallel. The
                       range of the right side
                       expression overlaps the
                       range of the left side
                       variable. *)
  p1[odd]:=0;       (* Left side parallel. Right
                       side scalar. *)
  i:=p2[5];         (* Left side scalar. Right
                       side parallel. The range
                       of parallelism includes
                       one element. *)
  sel:=p1 > p2;     (* Both sides parallel. The
                       same range. *)
  i:=p2[sel];       (* Left side is scalar. Right
                       is parallel. sel must have
                       one and only one true ele-
                       ment. *)
end.
```

In statements where data is stored in different words in the Associative Array, the movement of data must be explicitly specified using standard functions for data alignment.

```
var p1,p2 : parallel array[0..100] of real;
begin

  (* p1[2]:=p2[3]; is not allowed. Should be
     written: *)
     p1[2]:=shift(p2,-1);

  (* p1[4..84]:=p2[0..80]; is not allowed.
     Should be written: *)
     p1[4..84]:=shift(p2,4);
```

4.3.4 The Control Structure

To control the sequential program flow, Pascal contains five different structured constructs: if, case, while, repeat and for statements. The first two are used to select different paths in the program execution, while the remaining three control repetition of statements. Similar concepts are included in Pascal/L to allow parallel expressions to control selection and repetition.

The construct:

```
if boolean expression then true-statement
                      else false-statement
```

in Pascal selects one of two different paths in the program flow, depending on the value of the boolean expression. In the corresponding parallel statement, the boolean expression yields a selector. Each element in the selector determines what statements will be executed on its corresponding data elements.

In a global perspective this means that both the true statement and the false statement are executed, but on different data. Rather than to extend the if-then-else construct in Pascal, a parallel selection takes the form:

```
where parallel boolean expression do true-
      statement
                  elsewhere false-statement
```

where the elsewhere-part is optional.

Analogous to the Pascal case statement, which is a generalization of the if-then-else construct, where the selection is based on the value, of the expression given at the head of the case statement, Pascal/L defines a parallel form of the case statement. As with the if-then-else construct, the parallel case does not choose *one* execution path but all, each working on different data. The form of the parallel case statement is:

```
case where parallel expression of
                        constant1 : statement;
                        constant2 : statement;
                                  ...
                        constantn : statement;
                        others : statement
end
```

where the others-part is optional. In the imple-

mentation of the parallel case statement, care must be taken so that the code generated assures that only one choice is made for each word in the Associative Array. Since every choice in the list is taken, one after another, it is possible that a variable in the head expression is changed so that a second correspondance would occur, this time with another constant.

In a similar way an extension to the Pascal

```
while boolean expression do repetition statement
```

is defined to control repetition for parallel data:

```
while and where parallel boolean expression
                    do repetition statement
```

Here the repetition statement is repeated as long as the parallel boolean expression takes the value true in *any* element. The repetition statement is only executed on data where the corresponding element in the boolean expression has the value true.

The following example illustrates the use of the while and where construct:

```
v1,v2 : parallel array[0..2] of integer;
begin
  v1[0]:=2; v1[1]:=4; v1[2]:=3; v2[0]:=0;
  v2[1]:=0; v2[2]:=0;
  while and where v1 > 0 do
  begin
        v2:=2*v2+v1;
        v1:=v1-1
  end;
```

loop iteration

```
1       v2[0]<-2*0+2=2
        v2[1]<-2*0+4=4
        v2[2]<-2*0+3=3

2       v2[0]<-2*2+1=5
        v2[1]<-2*4+3=11
        v2[2]<-2*3+2=8

3       v2[0]<-5   unchanged since v1[0]=0
        v2[1]<-2*11+2=24
        v2[2]<-2*8+1=17

4       v2[0]<-5   unchanged since v1[0]=0
        v2[1]<-2*24+1=49
        v2[2]<-17  unchanged since v1[2]=0
```

The loop ends after four iterations since all elements in v1=0.

4.3.5 Standard Functions

A number of standard functions for data alignment are defined. Some of these work on variables with arbitrary range of parallelism, while others are defined for fixed size variables.

```
shift (parallel array ↕ selector, i)
```

The function shifts the parallel variable, i steps, along its first dimension. Zero elements are shifted in from the edge. This corresponds to moving data up or down the Associative Array.

```
rotate (parallel array ↕ selector, i)
```

Similar to the shift function except that the elements that are shifted out at one edge are shifted in at the opposite edge of the parallel variable.

```
propagate (selector, i)
```

The propagate function copies all true elements in the selector to the i following elements.

```
var  s1 : selector[1..10]:=(3,6=>true, others=>
          false);
     s2 : selector[1..10];
begin
  s2:=propagate(s1,2);  (* s2 will be true in
                           elements: (3,4,5,6,
                           7,8) *)
  ...
```

```
exchange (parallel array ↕ selector)
```

The elements of the variable are pairwise interchanged using the Exchange Network. The range of the variable must be even.

```
shuffle (parallel array ↕ selector)
```

The variable is shuffled using the Perfect Shuffle Interconnection Network. The function is only defined for parallel variables which have a range corresponding to the size of the Associative Array.

```
first (selector)
```

This function finds the first true element in a selector and returns a new selector with only this element true.

```
next (selector)
```

The next-function returns the same value as the first-function. The difference is that the first true element in the parameter automatically is assigned the value false.

```
any
```

The any-function returns the value false if a previous call to the first- or the next-function returned an all-false selector, otherwise it returns the value true.

```
some (parallel boolean expression)
```

A call to the some-function evaluates the boolean expression and returns the value true if it contains at least one true element, otherwise it returns the value false.

```
var  par1 : parallel array[0..9] of integer;
     sel1 : selector[0..9];
     su   : integer;
begin
  su:=0;
  sel1:=par1[*] > 10;   (* select elements greater than 10 *)
  while some(sel1)  do
        su:= su + par1[next(sel1)];
```

```
(* su contains the sum of all elements in par1
   whose value > 10 *)
```

```
responders (parallel boolean expression)
```

The responders-function evaluates the boolean expression and returns the number of true elements in the result.

```
range (parallel expression)
```

Returns a selector of range 256 with true elements indicating the range of parallelism for the expression.

4.3.6 Input and Output

The Pascal standard procedures read and write are extended to allow input and output of parallel variables. Either complete parallel arrays or selected subsets can be read and written.

5. Programming examples

The use of Pascal/L in two different applications is presented in the following programming examples.

The first example demonstrates one common operation on a relational data base, namely the PROJECT operation:

```
PROJECT R1 OVER A GIVING R2
```

where R1 and R2 are relations and A an attribute of R1. This operation creates a new relation, R2, from R1 by discarding attributes other than A. After that, all redundant tuples are removed from R2. Each relation has a corresponding mark selector that indicates where tuples are defined. A description of the operation can be found in [13].

```
Program Project;
var  r1mark : selector[0..255]; (* Selects words
                                   containing r1 *)
     r2mark : selector[0..255]; (* Selects words
                                   containing r2 *)
     temp1 : selector[0..255];  (* Marks remaining
                                   tuples in r1 *)
     temp2 : selector[0..255];  (* Marks all du-
                                   plicates of the
                                   tuple that is
                                   under compari-
                                   son *)
     r1 : parallel array[0..255] of record
                                  a,b,c : string[20]
                                   end;
     instance : string[20];

begin
  . . . (* Relation r1 is input and r1mark is
           initiated *)
  temp1:=r1mark;
  instance:=r1[first(r1mark)].a; (* Select first
                                    instance of
                                    attribute a *)
     while any do

     begin
       temp2:=(instance=r1[temp1].a); (*Select du-
                                        plicates *)
       temp1[temp2]:=not (temp1); (* mark as ana-
                                     lyzed *)
       r2mark[first(temp2)]:=true (* the first is
                                     included in
                                     r2 *)
       instance:=r1[first(temp1)].a; ( * get next
                                        distinct
                                        instance
                                        of attri-
                                        bute a *)
     end;
   end.
```

The second example shows how the FFT algorithm can be programmed on LUCAS. For details of the use of the Perfect Shuffle and Exchange Networks in FFT, see [12,14].

```
Program FFT;
const  iterations=8; (* 8 iterations for 256 point
                        FFT *)
type   complex = record  re,im : real  end;
var    omega : parallel array[128..255,1..itera-
               tions] of complex;

       product : parallel array[128...255] of
                 complex;
       samples : parallel array[0..255] of com-
                 plex;
       spectrum : parallel array[0..255] of com-
                  plex;
       iter : integer;
       lower : selector[0..255]:= 0..127=>false,
               others=>true);
       even : selector[0..255]:=
                  (0..254 step 2=>true, others=>
                   false>;
begin
      . . . (* input samples and complex constants
               omega *)
  spectrum:=samples;  (* spectrum after 0 itera-
                         tions *)
  for iter:=1 to iterations do
  begin
    product[lower].im:=spectrum[lower].re *
                          omega[lower].re -
                        spectrum[lower].im *
                           omega[lower].im;
    product[lower].im:=spectrum[lower].re *
                       omega[lower].im+
                       spectrum[lower].im *
                       omega[lower].re;

    where even do
      begin
        spectrum.re:=spectrum.re + shuffle(pro-
                     duct.re);
        spectrum.im:=spectrum.im + shuffle(pro-
                     duct.im);
      end
    elsewhere
      begin
        spectrum.re:=spectrum.re - exchange(shuff-
                     le(product.re));
        spectrum.im:=spectrum.im - exchange(shuff-
                     le(product.im));
      end;
  end;   (* for-loop *)
```

```
  (* FFT spectrum is found in array 'spectrum'
     with bit-reversed index *)
end.
```

6. Summary

The LUCAS associative array processor is intended as a working tool for research in the field of associative processing and some related application areas. In this paper programming aspects are investigated and a high level language, Pascal/L, is proposed.

Pascal/L is defined as a superset of Pascal and includes the following extensions:
- parallel variables that are allocated to the Associative Array
- an indexing scheme to access parts of the parallel variables
- expressions that include parallel variables
- an extended control structure, where parallel expressions are used to control the execution
- standard functions for data alignment of parallel variables
- extended input and output.

Appendix A.
Syntax summary of the extensions to Pascal

Data Declarations

```
<selector type> ::=selector [constant..constant]|
    selector [constant..constant] := <boolean
                                      aggregate>

<boolean aggregate> ::= <choice> => <boolean va-
                                    lue> |
    <choice> => <boolean value> , others => <boo-
                                    lean value>

<choice> ::= constant | constant..constant|
    constant..constant step constant

<boolean value> ::= true | false

<parallel array type> ::=
    parallel array [constant..constant]
                   of <parallel component type>|
    parallel array [constant..constant , constant
                                      ..constant]
                   of <parallel component type>

<parallel component type> ::= <parallel type>

<parallel type> ::= <parallel type identifier> |
    <parallel standard type> | record <parallel
                               field list> end

<parallel type identifier> ::= <identifier>

<parallel standard type> ::= integer | real | boo-
                             lean | char |
                             string[constant]

<parallel field list> ::= <parallel record section>
                          { ; <parallel record
                              section>}

<parallel record section> ::=
    <field identifier> { , <field identifier> } :
                       <parallel standard type>
```

Indexing

```
<parallel indexed variable> ::=
    <parallel array variable> [ <first index> ] |
    <parallel array variable> [ <first index> ,
                                <expression> ]

<parallel array variable> ::= <variable>

<first index> ::= <selector expression>  | * |
                  constant | constant..constant

<selector expression> ::= <expression>
```

Statements

```
<where statement> ::=
    where <parallel boolean expression> do <sta-
                                        tement> |
    where <parallel boolean expression> do <sta-
                                        tement>
                        elsewhere <statement>

<while and where statement> ::=
    while and where <parallel boolean expression>
                        do <statement>

<parallel case statement> ::=
    case where <parallel expression> of
         <case list element> { ; <case list ele-
                                   ment> } end |
    case where <parallel expression> of
         <case list element> { ; <case list ele-
                                   ment> } ;
                  others : <statement> end

<case list element> ::= <case label> { , <case
                        label> } : <statement>
```

References

[1] H.K.Resnick, A.G.Larson, "A COBOL Extension for Associative Array Processors", Proc of the Conference on Programming Languages and Compilers for Parallel and Vector Machines, pub. as SIGPLAN Notices 10,3 (March 1975)

[2] E.B.Allen, A.G.Larson, "FORTRAN Extension Design Concepts for Associative Processing", 1975 Sagamore Computer Conf. on Parallel Processing.

[3] R.H.Perrott, "A Language for Array and Vector Processors", ACM Trans on Prog Languages and Systems, Vol 1, no 2, Oct 1979

[4] A.P.Reeves, J.Bruner, M.Poret, "The Programming Language Parallel Pascal", Internal Purdue Electrical Engineering Report TR-EE 80-32, July 1980

[5] R.G. Lange, "High Level Language for Associative and Parallel Computation with STARAN", Proc of the 1976 Internat. Conference on Parallel Processing.

[6] P.M.Flanders, "DAP-FORTRAN Language", CM39, RADC, ICL 1976

[7] K.Stevens, "CFD - a FORTRAN-like Language for the ILLIAC IV", Sigplan Notices, March 1975

[8] D.H.Lawrie, T.Layman, D.Baer, J.M.Randal, "Glypnir - a Programming Language for ILLIAC IV", Comm ACM 18, March 1975

[9] R.E.Millstein, "Control Structures in ILLIAC IV FORTRAN", Comm ACM 16, Oct 1973

[10] P.T.Mueller Jr, L.J.Siegel, H.J.Siegel, "A Parallel Language for Image and Speech Processing", Proc. of COMPSAC 80, Oct 1980

[11] K.Bratbergseugen, O.Risnes, T.Amble, "ASTRAL - A Structured and Unified Approach to Data Base Design and Manipulation", RUNIT Comp Centre at the University of Trondheim, Norway. Report No STF14.A80003, 1979

[12] C.E.Fernstrom, I.F.Kruzela, B.A.Svensson, "The LUCAS Associative Array Processor", Lund University Dept of Computer Engineering Techn Report 1981

[13] I.F.Kruzela, B.A.Svensson, The LUCAS Architecture and Its Application to Relational Data Base Management", Proc of 6th Workshop Computer Architecture for Non Numeric Processing 1981

[14] H.S.Stone, "Parallel Processing with Perfect Shuffle", IEEE Trans on Computers, vol C-20, Feb 1971

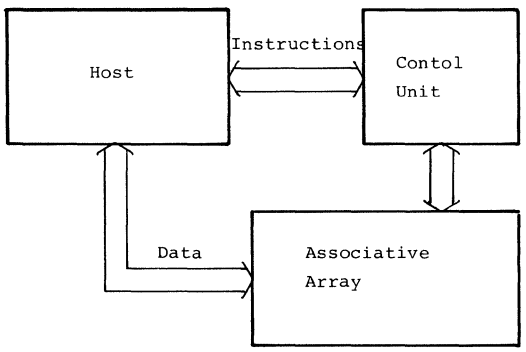


Figure 1.

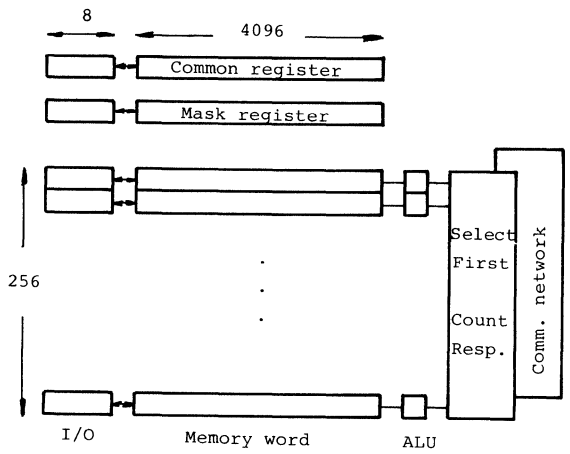


Figure 2. The Associative Array

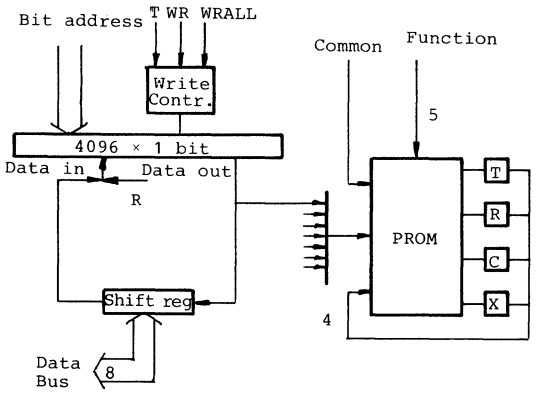


Figure 3. One Memory Word. I/O register and ALU.

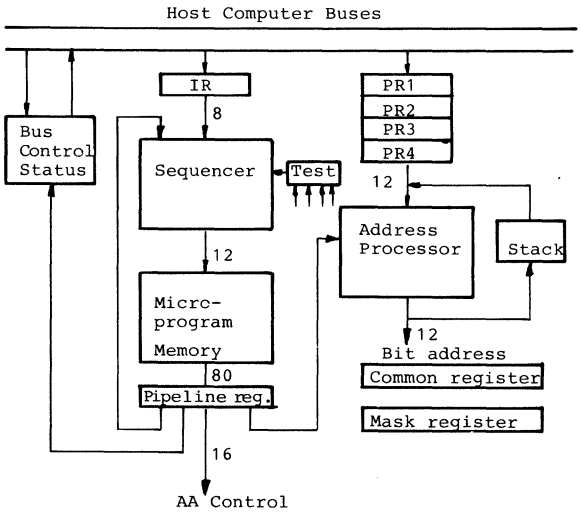


Figure 4. The Control Unit

WAFER SCALE INTEGRATION OF CONFIGURABLE, HIGHLY PARALLEL (CHiP) PROCESSORS[a] -- EXTENDED ABSTRACT --

Kye S. Hedlund[b]
Lawrence Snyder
Department of Computer Science
Purdue University
West Lafayette, IN 47907

Abstract -- A two level decomposition strategy for wafer scale implementation of CHiP processors is presented. With current technology, machines with over 300 processors per wafer can be fabricated. These wafer scale machines will be cheaper, faster and more reliable than their counterparts implemented with single chip components.

Introduction

Many different architectures for parallel processors have been proposed but few large-scale parallel systems have actually been built. One reason is that a large-scale parallel processor consists of a great many components. This introduces severe practical problems of construction, wiring and reliability. If the number of individual components could be decreased, parallel processors would be far easier and cheaper to construct.

The absolute minimum number of components is reached when the entire parallel processor is fabricated on a single silicon wafer. These *wafer scale systems* have greatly reduced cost due to the increased level of integration. Reliability is higher since the connections between processors are implemented in silicon. Furthermore, there is the potential for increased performance since data values passed between processors are not driven off the wafer.

Wafer scale integration (WSI) has been previously attempted by discretionary wiring [1]. Due to the additional masking steps required, this has not proved to be practical. Other researchers are currently investigating laser restructuring [2] and fuse blowing approaches to implementing WSI.

At the center of our approach is the configurable, highly parallel (CHiP) processor [3] family of restructurable architectures. CHiP computers are composed of many simple processing elements (PEs) that are not directly connected together but are inserted at regular intervals into a *switch lattice*. The programmable switches can be set to connect the PEs in a wide variety of interconnection patterns.

We propose wafer scale implementation of CHiP processors. No extra masking steps are required making the approach cost effective. A two level methodology decomposes the problem into *mapping* small CHiP machines into building blocks and then *structuring* the building blocks on the wafer. Although we consider CHiP computers, the concepts presented are entirely general and can be applied to other parallel systems.

[a] Research supported in part by Office of Naval Research under Contract Nos. N00014-80-K-0816 and Contract N00014-81-K-0360.
[b] Author's present address is Dept. of Computer Science, U. of North Carolina, Chapel Hill, NC 27514

Implementing Wafer Scale Integration

A large number of simple PEs can be patterned on a single wafer. But on any given wafer, many of the PEs will contain defects - errors in the circuitry such as broken wires or nonfunctional transistors. These defects are randomly distributed over the wafer surface.

To implement a wafer scale system, all PEs on a wafer are tested, and then the good PEs are connected together. The wafer is *structured* so that the presence of faulty PEs is masked and only functional PEs are used. The switch lattice of CHiP processors provides the interconnection flexibility required to structure the wafer. Switches can be programmed to route around faulty PEs and connect together only the functional PEs. Redundant switches are added to the lattice to perform the structuring.

The structuring problem is made difficult by low PE yield; for any particular PE it is very unlikely that all its four neighbors will also be functional. The positioning of good PEs on the wafer differs from the required connection pattern. Hence considerable wiring may be required to connect a PE to its neighbor in the CHiP lattice. This introduces delays from the intervening switches and increases signal transmission time. System speed is proportionately reduced.

Now suppose that most PEs are functional. The good PEs are distributed in a more regular pattern - one more closely resembling a lattice. This simplifies the structuring problem. Faulty PEs can be eliminated by *column exclusion*, all PEs in a column containing a faulty PE are eliminated, and the columns adjacent to the excluded column are connected together. The only requirement is that we can wire around faulty or unused PEs. This strategy has been used previously in 64K memories and in Batcher's MPP.

For this simple approach to be practical, the wafer must contain very few faulty PEs. But due to the nature of the integrated circuit manufacturing process, high yield is achievable only with very simple circuits - much less complex than a PE.

But suppose the units patterned on the wafer are not individual PEs but *building blocks* of a CHiP machine. Each block is itself a small CHiP processor. By providing sufficient redundancy within each block, a smaller but completely functional CHiP machine (the *virtual lattice)* will exist within almost every building block. With each block contributing a small, fixed size virtual lattice, a large CHiP machine is formed from the blocks.

If enough redundancy is provided within each building block, the percentage of blocks containing a smaller, completely functional virtual lattice will be very high. This allows the use of the column exclusion strategy to eliminate the relatively rare block that is completely dysfunctional.

0190-3918/82/0000/0262$00.75 © 1982 IEEE

To determine the degree of redundancy required, we developed a yield model based on the Price model of the integrated circuit manufacturing process. This model is the basis for the quantitative determination of the effect of redundancy.

The smaller virtual lattice must be *mapped* into the larger building block. This mapping makes the block function as if it were a virtual lattice. An observer of the input/output behavior of the block would be unable to distinguish it from a virtual lattice. The mapping associates each vertex (PE or switch) in the virtual lattice with an image in the block. Furthermore, every datapath in the virtual lattice becomes a path in the block. A path may be a single datapath or a sequence of connected switches.

In summary, we have introduced a two level decomposition of the wafer scale lattice. A very large CHiP lattice is patterned on the wafer. It is logically divided into small building blocks. From almost every building block a small fixed size CHiP processor is extracted, and the blocks are composed to form the wafer scale machine. Faulty blocks are eliminated by column exclusion. Note that the two level decomposition limits the length of paths between PEs to the size of a block. This assures that system performance will not be catastrophically degraded by the occurrence of faulty PEs.

Designing Building Blocks

Each PE has a simple arithmetic-oriented instruction set, an 8-bit ALU and 64 bytes of memory. This is sufficiently powerful to execute a wide variety of systolic algorithms. Implemented with current (2μm) technology, each PE occupies approximately a 1.75mm × 1.75mm region of silicon.

A 2 × 2 virtual lattice is mapped into a building block. From the yield model, the cumulative probability density function of defects (Fig. 2) for relative area A = 1.0 and 2.0 is known. From this we can derive [3] the probability that at least 4 out of N PEs are functional (Table 1). 4 good PEs can be found out of a set of 12 in 99% of the time. Consequently, each building block is chosen to be 4 PE × 3 PE CHiP machine insuring that almost all blocks contain the 4 PEs required for a 2 × 2 virtual lattice. In addition to redundant PEs, each building block also has twice the required number of switches (Fig. 1b). These redundant switches are used to map the 2 × 2 lattice into the block.

A Wafer Scale CHiP Processor

A 9 × 9 grid of building blocks can be patterned on a 4" wafer (Fig. 3). The bonding pads and drivers required to connect the wafer scale CHiP machine to external memory (or other wafer scale machines) are placed around the periphery of the grid. Redundant drivers are used to guarantee the integrity of the external connections. The grid occupies only 68% of the wafer area which leaves sufficient remaining area for 150 pads and drivers per lattice edge. Packaging constraints may place a lower limit on the number of external connections.

A 2 × 2 virtual lattice is recovered from each block 99% of the time, and the occasional faulty block is eliminated by excluding the column containing the fault. Table 2 shows the frequency of different grid sizes after faulty blocks are eliminated. Almost half of the wafers use all 81 blocks. An "average" wafer contains a CHiP processor of 297 PEs. The switches in unused or faulty blocks are used to connect the blocks in the columns adjacent ot a faulty block. Thus the "wire around" requirement for blocks becomes a "wire through" capability via the CHiP switch lattice.

Is this approach efficient? In addition to excluded columns, only 4 of the 12 PEs in each block are used. On the average, there are 74 usable 2 × 2 CHiP lattices in each wafer scale machine (Table 1). Suppose we simply pattern the entire wafer with 2 × 2 lattices. A 4" wafer holds 288 of these. At the predicted 20% yield, only 58 of the lattices are fully functional. Hence *fault tolerant building blocks containing redundant components are area efficient*. The area lost to redundancy is more than made up for by the increased recoverability of the blocks. Moreover, the wafer scale solution is more robust to failures, has better performance and lower cost.

Conclusions

The two level decomposition could be a practical method for implementing wafer scale integration. It is cost effective since no additional processing steps are required. Additionally, the maximum wire length between PEs is limited, and the wafer area is efficiently utilized.

As described above, our methodology benefits from the fact that the mechanism needed for structuring, the switch lattice, is an integral part of the architecture. Although this simplifies our work, it is not necessary. The method is entirely general. It can be applied to other systems composed of uniform parts including parallel processors with fixed interconnection structures.

Practical problems of testing, power consumption, synchronization and clocking, etc. are discussed in detail in [4,5].

Acknowledgments

The authors would like to thank Gerold Neudeck for his assistance in formulating the yield model.

References

[1] R.L. Petritz "Current Status of Large Scale Integration Technology," *IEEE J. Solid-State Circuits* **SC-2** ,4(Dec. 1967), 130-147.

[2] G. Chapmann - private communication, Lincoln Labs.

[3] L. Snyder "Introduction to the Configurable, Highly Parallel Computer," *IEEE Computer* **V-15** ,1 (Jan. 1982), 47-56.

[4] K. Hedlund AND L. Snyder "Wafer Scale Integration of Configurable, Highly Parallel (CHiP) Processors," Tech. Report 407, Comp. Sci. Dept., Purdue U., Aug. 1982.

[5] K. Hedlund "Wafer Scale Integration of Parallel Processors," Ph.D. Thesis, Comp. Sci. Dept., Purdue U., Aug. 1982.

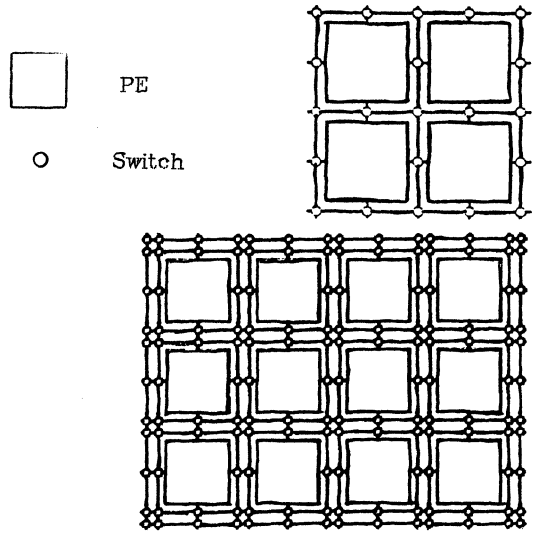


Figure 1 - a 2 × 2 CHiP Lattice (Virtual Lattice) and a 4 × 3 Building Block

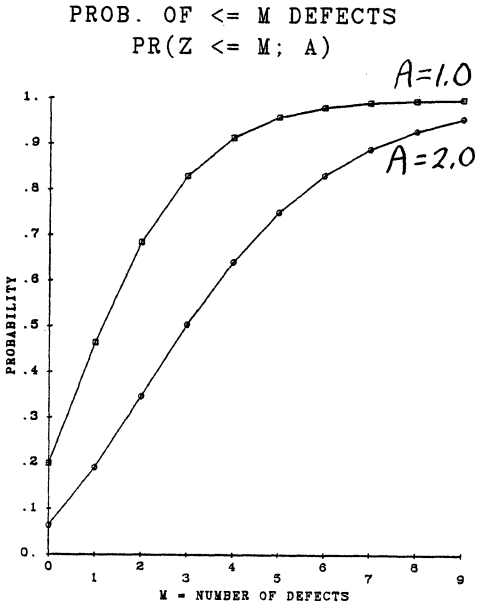


Figure 2 - Defect Distribution for Relative Area = 1.0 and 2.0

Table 1 - Recovery of 4 PEs from N PEs

N = number of PEs	relative area	prob >= 4 good PEs	number of redundant PEs
4	1.00	.200	0
6	1.50	.733	2
6	2.25	.953	5
12	3.00	.990	8

Table 2 - Size of Wafer Scale Processor for 9 × 9 Grid

Lattice Size from a 9 × 9 Grid			
probability	cumulative probability	grid size	size of CHiP processor (PEs)
.443	.443	9 × 9	18 × 18 = 324
.394	.837	9 × 8	18 × 16 = 288
.129	.966	8 × 8	16 × 16 = 256
.0271	.993	8 × 7	16 × 14 = 224
.0069	1.000	< 8 × 7	

Expected Number of Processors = 297

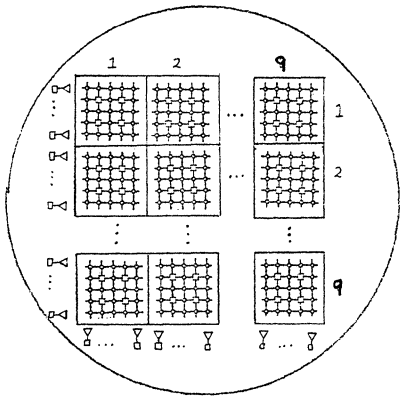


Figure 3 - Layout of a Wafer Scale CHiP Processor

TESTING COORDINATION FOR "HOMOGENEOUS" PARALLEL ALGORITHMS

Janice E. Cuny
Lawrence Snyder
Department of Computer Sciences
Purdue University
West Lafayette, Indiana, 47907

Abstract: A collection of parallel processors is said to be coordinated if each write from one processing element (PE) to another is answered by a read. We report on an efficient algorithm to test coordination for parallel programs in which the code for each PE is a loop. We also test a weaker predicate for parallel algorithms with oblivious PE codes and we show that the general problem is PSPACE-hard.

"Homogeneous parallel algorithms" refers to a large class of parallel computations, reminiscent of cellular arrays, formed from "identical" processing elements (PEs) that often use pipelining and novel interconnection structures. They include algorithms for matrix operations [1], dynamic programming [2], data base operations [3], sorting [4], and signal processing [5]. The algorithms in this class are motivated by the potential for direct VLSI implementation but they are equally well suited for implementation as programs for general purpose parallel architectures [6].

Upon close scrutiny many of these algorithms are anything but "homogeneous". The processing requirements of PEs may differ because of initialization details, termination details, timing and edge effects (i.e. special problems encountered when a PE is on the perimeter of a processing array). There is a benefit in retaining the conceptual simplicity of homogeneity and relegating the differences to the status of implementation details. This is because, at a high level, many processors are identical and their differences on lower levels can largely be inferred from the algorithm's global structure. A key goal, then, in the effort of simplifying parallel algorithm development is:

> To support "homogeneous" simplification by automatically generating *PE variants* when possible and to assist in their development when manual design is required.

We report on progress towards this objective for variants that differ in the timing characteristics of their interprocess communication.

We have reported [7] on the automatic synthesis of PE variants to synchronize interprocess communication. We start with a parallel algorithm which assumes an abstract data flow execution mode and for a limited, but widely practical class of algorithms, we generate the timing necessary for synchronous execution. But what if the algorithm is not in the class or if manual design is required? In this paper, we report on algorithms that assist the designer by testing the compatibility of interprocess communication.

A MODEL OF PARALLEL PROGRAMS

Our abstraction of a homogeneous parallel processor is an *Interprocessor Communication (IC) System*. An IC system is given by a set of m finite state machines, $M_1, M_2, \ldots, M_m$, each describing the input/output behavior of a single PE.† The alphabet of the machine is the power set of ††

$$\{r_{i,\sigma}, w_{i,\sigma} | i \in [m] \wedge \sigma \in \Sigma\}$$

where Σ is a finite set of values, $r_{j,\sigma}$ denotes the reading of the value σ from PE j, $w_{j,\sigma}$ denotes the writing of the value σ to PE j and φ, the empty set, represents any other operation not involved in interprocessor communication including operations that transfer values to and from the external environment. If PE i writes to PE j or PE j reads from PE i, we say that there is a *communication link* from i to j. Notice that the interconnection graph of the processors is implicit in the indexes of the symbols.

We assume that the PEs execute synchronously and that on each step a PE can execute a set of operations simultaneously. Specifically, the execution of an IC system is defined by two sequences, $C^1, C^2, C^3, \ldots$ and $Q^0, Q^1, Q^2, \ldots$. Each element of the first sequence is an an m-vector of symbols, one per PE, describing the operations executed in a single step. Each element Q^k of the second sequence is an $m \times m$ matrix of strings,

This work is part of the Blue CHiP Project. It is supported in part by the Office of Naval Research Contracts N00014-80-K-0816 and N00014-81-K-0360. The latter is Task SRO100.

† IC systems can be defined more generally [8] but for the purposes of this paper, we present only a limited version.

†† $[m]$ denotes the set $\{1,2,\ldots,m\}$.

0190-3918/82/0000/0265$00.75 © 1982 IEEE

where $q_{i,j}^k$ gives the status of the communication link from PE i to PE j. The $q_{i,j}^k$ are all of the form $\alpha\beta$ with $\alpha \in \Sigma^*$ and $\beta \in (\Sigma^{-1})^*$ where Σ^{-1} is the set of inverse symbols of Σ. We interpret $q_{i,j}^k = \alpha\beta$ to mean that the symbols α have been written on the link but they have not yet been read and the symbols β have been requested from the link but they have not yet been written. A symbol σ and its inverse σ^{-1} cancel at the boundary between α and β, i.e. $\sigma \cdot \sigma^{-1} = \lambda$, the empty string.

In general, for $k \geq 1$ and $i \in [m]$, c_i^k is the k-th symbol in a word defined by M_i and

$$c_i^1 \cdot c_i^2 \cdot c_i^3 \cdot \cdots \cdot c_i^k \in L(M_i) \ .$$

Initially, Q^0 is empty, i.e., $q_{i,j}^0 = \lambda$ for all $i,j \in [m]$. Generally, $q_{i,j}^{k+1} = a \cdot q_{i,j} \cdot b$ where

$$a = \begin{cases} \sigma & \text{if } w_{j,\sigma} \in c_i^{k+1} \\ \lambda & \text{otherwise} \end{cases}$$

and

$$b = \begin{cases} \sigma^{-1} & \text{if } r_{i,\sigma} \in c_j^{k+1} \\ \lambda & \text{otherwise} \end{cases}$$

and a sequence $Q^0, Q^1, Q^2, \ldots$ is a ***legal computation*** if and only if

$$(q_{i,j}^k \in \Sigma^*) \vee (q_{i,j}^k \in (\Sigma^{-1})^*) \ .$$

The latter condition enforces our intention that a PE reads the same symbol that was written to it in the corresponding write.

An IC system is said to be ***strongly coordinated*** if for all i, j, and k,

$$q_{i,j}^k = \lambda$$

that is, during synchronous execution corresponding reads and writes occur simultaneously. † If we allow the writes to precede their corresponding reads we say that the system is ***weakly coordinated:*** for all i, j, and k

$$q_{i,j}^k \in \{\lambda\} \cup \Sigma \ \wedge$$
$$((k>0 \wedge q_{i,j}^{k-1} \in \Sigma \wedge q_{i,j} = a \cdot q_{i,j}^{k-1} \cdot b) \Rightarrow a = \lambda) \ .$$

RESULTS

In this work, we consider algorithms to answer the question

> Given an IC system, is it strongly (weakly) coordinated?

for three cases of increasingly complex IC structure. We develop algorithms for testing coordination in the first two cases: ***loop programs*** in which all PEs repeat a single cycle of operations and ***oblivious programs*** in which restrictions on legal computations are removed. In the third case, consisting of general IC systems, we show that testing is computationally intractable.

Loop Programs

We first restrict our attention to loop programs in which each PE executes an initialization sequence and then repeatedly executes a single cycle of instructions. While this restriction seems prohibitive, many highly parallel systems, such as the systolic processors [1], can be characterized in this way.

We have developed an algorithm to test strong coordination on a single communication link of a loop program.†† The algorithm checks the first $MAX + LCM$ execution steps of the machines for coordination errors, where MAX is the length of the longer initialization sequence for the PEs involved and LCM is the least common multiple of their cycle lengths. This test is sufficient because, for $k > MAX + LCM$, each PE executes the same operations in time step k that it does in time step $(k - MAX) \bmod LCM$. The algorithm, with a few modifications can be used to test weak coordination as well. In both cases, it requires $O(n^2)$ time where n is the maximum number of states in the machines involved.

If we assume that a system is composed of a small number of distinct PE types which are interconnected in analogous ways, then it is sufficient to test each link type just once. For a system with t link types, we have

> **Theorem 1.** The coordination of a system of interconnected, loop programs can be tested in $O(c \cdot t)$ where c is a constant dependent on the loop structure of the PE code.

Notice that the bound is independent of the number of PEs and is influenced only by the variety of their communication, which would be small for "homogeneous" algorithms.

Oblivious Programs

Generalizing, we allow arbitrary finite state machines but we remove the restriction on legal computation sequences. In these oblivious programs, it is impossible to branch based on the

† We permit simultaneous reading and writing for technical simplicity, but the more conventional unit time delay between writing and the subsequent reading requires only more complicated, not substantially different, definitions.

†† The complete details of our algorithms and proofs are presented in the full version of this paper [9].

values transmitted between PEs. For such systems, we can test only worst case coordination, answering the question

> Given a communication link, does it have a potential coordination error?

If our algorithm reports NO, then the communication link is coordinated; if our algorithm reports YES, it is possible that the detected error would never occur in any legal computation of the system.

The algorithm first constructs the "cross product" machine for the two finite state machines involved. For strong coordination, the testing question is then reduced to a question of state reachibility in this new machine. The test, therefore, requires $O(q)$ time where q is the number of states in the cross product machine; in terms of the original machines, the algorithm requires $O(n^2)$ time. For weak coordination, we reduce the test to a predicate on the computation tree for the cross product machine. We show that we can determine the value of this predicate in $O(q^3) = O(n^6)$ time. For systems with t interface types, then, we have

> **Theorem 2.** The worst case coordination of an IC system can be tested in $O(d \cdot t)$ where d is a constant dependent on the structure of the PE code.

Again, the result is dependent only on the variety of the PEs not necessarily their number.

General IC Systems

In the most general case, given an IC system with arbitrary structure and data dependent branching, we show

> **Theorem 3:** Testing the coordination of arbitrary IC systems is PSPACE-hard [10].

The proof of this theorem involves reducing the language recognition problem for linear bounded automata to our testing question.

CONCLUSIONS

Although the complexity theory results indicate that coordination testing is a very complicated task, it is important to notice that many recently developed parallel algorithms are covered by Theorem 1. The testing algorithms discussed here are being implemented and we expect that they will be of significant assistance in the development of parallel programs.

ACKNOWLEDGEMENTS

We owe a debt of gratitude to Dennis Gannon for useful discussions concerning coordination, to Cathy Cole for her cheerful assistance with the algorithms, and to other Blue CHiP Project members, including Francine Berman, for their support and suggestions.

REFERENCES

[1] H. T. Kung and C. E. Leiserson, "Systolic Arrays (for VLSI)," Tech. Rep. CS-79-103, Carnegie-Mellon University (1979).

[2] L. J. Guibas, H. T. Kung and C. D. Thompson, "Direct VLSI Implementation of Combinatorial Algorithms," Caltech Conference on VLSI, California Institute of Technology, 1979.

[3] S. W. Song, "A Highly Concurrent Tree Machine for Data Base Applications," *Proc. Int'l Conf. Parallel Processing*, 1980, pp. 259-268.

[4] Sally Browning, "The Tree Machine: A Highly Concurrent Programming Environment," Ph.D. Thesis, California Institute of Technology, Jan. 1980.

[5] Hassan M. Ahmed, Jean-Marc Delosme, and Martin Morf, "Highly Concurrent Computing Structures for Matrix Arithmetic and Signal Processing," *Computer*, Vol. 15, No. 1, pp. 65-82, Jan. 1982.

[6] Lawrence Snyder, "Introduction to the Configurable Highly Parallel Computer," *Computer*, Vol. 15, No. 1, pp. 47-56, Jan. 1982.

[7] Janice E. Cuny, and Lawrence Snyder, "Conversion from Data-Flow to Synchronous Execution in Loop Programs," Tech. Rep. CS-82-392, Purdue University, 1982.

[8] Janice E. Cuny and Lawrence Snyder, "A Model for Analzying Generalized Interprocessor Communication Systems," Tech. Rep CS-82-406, Purdue University, 1982 (in preparation).

[9] Janice E. Cuny, and Lawrence Snyder, "Testing the Coordination Predicate," Tech. Rep. CS-82-391, Purdue University, 1982 (in preparation).

[10] Michael R. Garey and David S. Johnson, *Computers and Intractability*, W. H. Freeman and Co., p. 271, 1979.

MPP VLSI MULTIPROCESSOR INTEGRATED CIRCUIT DESIGN

John Burkley
Digital Technology Department
Goodyear Aerospace Corporation
Akron, Ohio 44315

ABSTRACT

A large scale integrated multiprocessor circuit has been developed for use in the Massively Parallel Processor system (MPP). The chip, built in an HCMOS technology, contains eight bit-serial processing elements (PE's) and is the basic building block for the MPP processing array.

INTRODUCTION

The MPP is a large scale single instruction stream, multiple data stream (SIMD) machine being built by Goodyear Aerospace Corp. for NASA/GSFC. (1,2,3). The system block diagram is shown in Figure 1. The Array Unit (ARU) processes two dimensional arrays of data. Array control is generated by the Array Control Unit (ACU) which executes the user program and performs any sequential processing and scalar arithmetic necessary to support array operations. Array data I/O is through a special Staging Memory which both stores and permutes array data. The Program and Data Management unit serves as an external I/O preprocessor.

The ARU makes the MPP special; it incorporates 16348 PE's organized in a 128 x 128 array and operating at a basic cycle of 100 nsec. Each PE supports boolean and arithmetic operations, is maskable and is capable of routing data to its orthogonal neighbors. Table I shows the speed of typical operations.

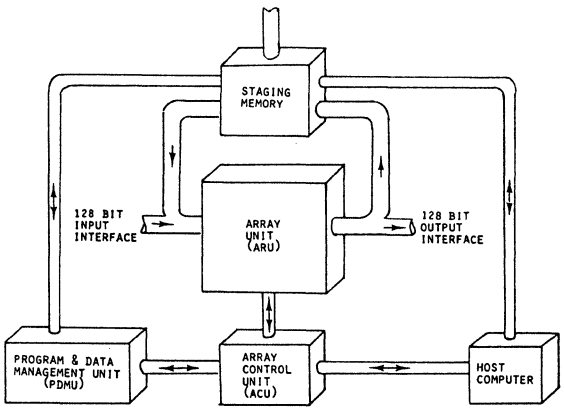


Fig. 1 - MPP Block Diagram

To build an array of this size and speed required the development of a VLSI chip. The chip is partitioned to include eight PE's configured in a 2 x 4 array, an eight bit bi-directional data port with a parity tree and a SUMOR tree, and a disable circuit capable of disconnecting the chip from its east-west neighbors. This last feature facilitates automatic repair of the array using redundant processing elements. The chip replaces some 200 MSI and SSI circuits. The chip will execute ten million operations per second when operating with high speed RAM (45 nsec access). PE memory was not included within the chip for several reasons. First, local memory would have reduced the number of PE's per chip and complicated its design and development. Second, the use of external memory allowed the MPP system to take full advantage of existing memory technology, allowing more memory per PE at a faster access time than is possible in HCMOS. Finally, future systems could expand PE memory without a chip redesign. A total of 2112 chips are required to construct an MPP array. This total includes a spare column of chips (4 columns of PE's) for redundancy.

TABLE I - SPEED OF TYPICAL OPERATIONS

OPERATIONS	EXECUTION SPEED*
ADDITION OF ARRAYS	
8-BIT INTEGERS (9-BIT SUM)	6553
12-BIT INTEGERS (13-BIT SUM)	4428
32-BIT FLOATING-POINT NUMBERS	470
MULTIPLICATION OF ARRAYS (ELEMENT-BY-ELEMENT)	
8-BIT INTEGERS (16-BIT PRODUCT)	1861
12-BIT INTEGERS (24-BIT PRODUCT)	910
32-BIT FLOATING-POINT NUMBERS	291
MULTIPLICATION OF ARRAY BY SCALAR	
8-BIT INTEGERS (16-BIT PRODUCT)	2824
12-BIT INTEGERS (24-BIT PRODUCT)	1489
32-BIT FLOATING-POINT NUMBERS	373

*MILLION OPERATIONS PER SECOND

CHIP DISABLE

A chip disable line is provided which logically disconnects the chip from the array by disabling the SUMOR output and enabling a bypass circuit which routes data directly from the west route and S register inputs to the east route and S register outputs. This logically removes that chip from the array allowing column substitution. Since only a small portion of chip logic must work for the bypass logic to be functional, a failed array could be repaired automatically by substituting a spare column of chips for a failed column without waiting for a maintenance call.

0190-3918/82/0000/0268$00.75 © 1982 IEEE

PE DESIGN

The PE includes six single bit registers (A,B,C,G,P,S), a variable length shift register, a full adder and some combinatorial logic. A PE logic diagram is shown in Figure 2. The chip is controlled by 16 control lines. The PE may be divided into four subunits; logic and routing, arithmetic, I/O, and masking. These subunits have independent control but share a common clock. The subunits are interconnected by a bi-directional data bus which also connects to external PE memory.

LOGIC & ROUTING SUBUNIT

The logic and routing subunit is formed by the P register together with some supporting combinatorial logic. P can be logically combined with the state of the data bus and the result is stored in P. When routing is enabled, one of four inputs to the route multiplexor is selected and latched in P. The multiplexor inputs are the states of the P registers in the north, south, east, and west neighbor PE's.

ARITHMETIC SUBUNIT

The arithmetic subunit consists of a serial-by-bit adder formed by B and C and a variable length shift register whose output may be stored in A. A may also be loaded from the data bus. The adder receives an input from A and P. When enabled by control the adder adds the two input bits to a carry bit stored in C and forms a two bit sum. The least significant bit is stored in B and the most significant bit is stored in C so it becomes the carry bit for the next cycle. C may be initialized to either a one or a zero.

The arithmetic unit also includes a variable length shift register for local storage of partial products. This feature significantly improves multiply and divide operation times. The shift register circulates the output of B back through N stages of delay to the adder input register A. The length of the shift register, N, may be set in steps of 4 from 2 to 30. Since A and B also add two stages of delay, the total shift register length may vary from 4 to 32.

I/O SUBUNIT

The I/O subunit is formed by the S register and a two input multiplexor which selects input from either the data bus or the S register of the PE's west neighbor. S register shifting may go on independent of other PE operations except when data must be stored or loaded from PE memory.

MASKING SUBUNIT

The masking subunit is formed by the G register. Masking is enabled when G is low. Routing and arithmetic operations may be masked separately. In addition, the state of P may be outputted to the data bus selectively negated by G. This allows a masked invert of data in PE memory to be executed in two cycles.

MEMORY INTERFACE

The PE subunits are interconnected by a bi-directional data bus. This bus may be used to exchange data between PE registers or to read and write PE memory. The control lines allow only one bus source at a time. The chip includes an eight bit parity tree which generates parity on memory write operations and checks parity on memory read operations. If bad parity is detected a parity error latch is set. Because of the parity tree delay, memory operations with parity will operate at a 120 nsec cycle. Parity may be ignored for 10MHz operation. The eight memory buses are also sumored to form a single bit output.

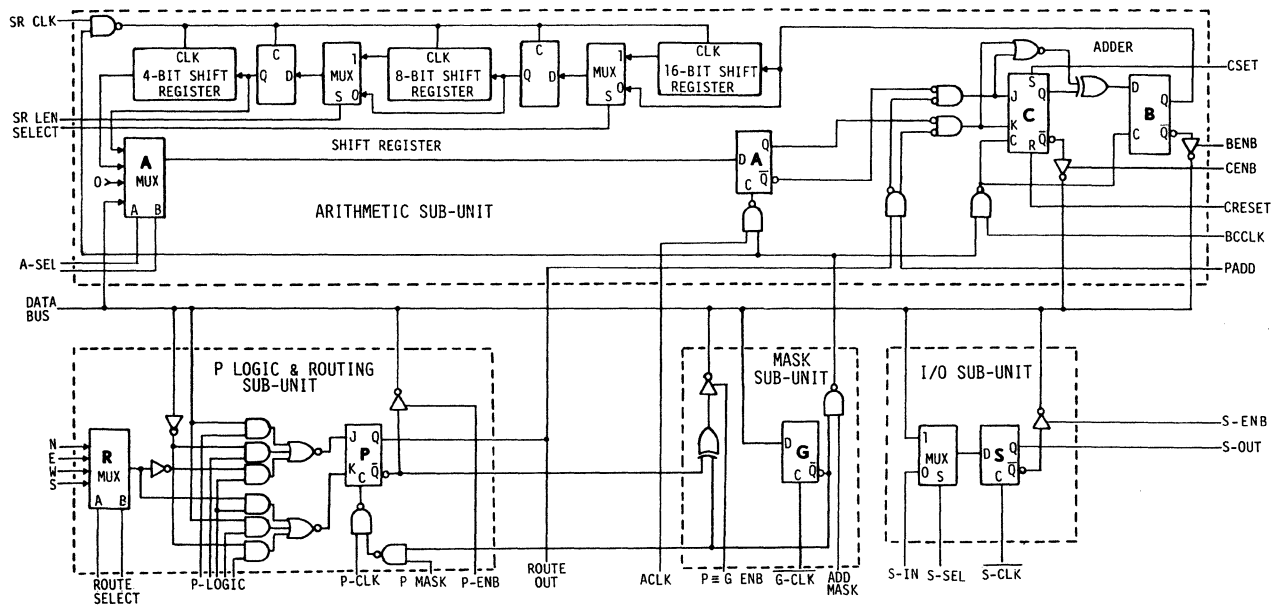


Fig. 2 - MPP Processing Element Logic Diagram

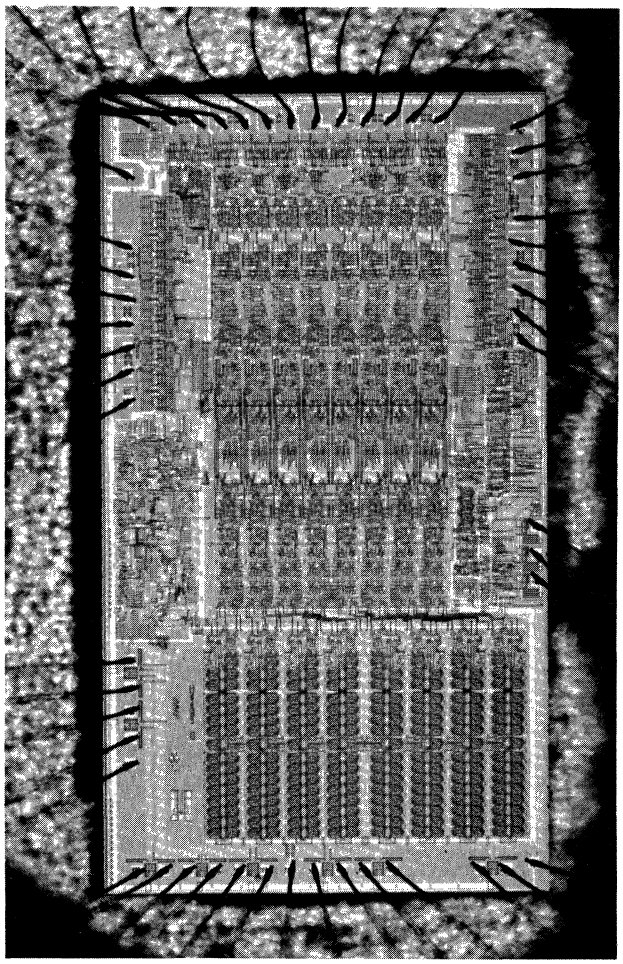


Fig. 3 - MPP Chip Photomicrograph

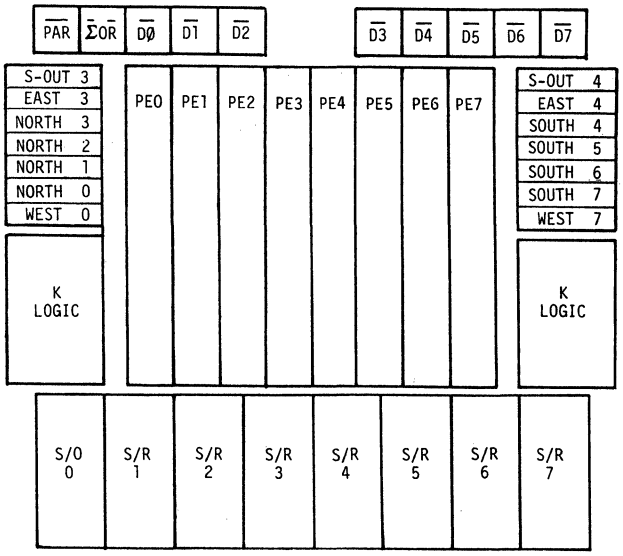


Fig. 4 - MPP Chip Topology Drawing

CHIP FABRICATION

The MPP multiprocessor integrated circuit was fabricated by Solid State Scientific Inc. in an HCMOS technology using 5um design rules. The design was implemented using about 8000 transistors and required a chip size of 235 x 131 mil^2. The chip requires two power supplies. Internal circuitry operates at 7 volts; the output translators require 5 volts. The chip is packaged in a 52 pin flat pack and dissipates 550mw when operating at 10 megahertz.

The chip design includes a high speed bi-directional data bus. This data bus was implemented using NMOS transistor pull-downs and a current mirror biased pull-up transistor. This bus implementation increased chip power dissipation but greatly improved response time.

The chip photomicrograph is shown in Figure 3; its topology drawing is shown in Figure 4. The eight PE's are grouped together in the middle of the chip in long narrow strips. This was done to minimize control line metal runs. The data bus and routing logic are grouped toward the top of the chip. This logic had the most severe timing constraints and was laid out as close to the chip pins as possible to minimize line delays. The shift registers are grouped along the bottom of the chip. The control decode is split and fed in from both sides of the chip.

CONCLUSIONS

An eight PE multiprocessor chip has been developed for use in the MPP. The PE chip design meets all the functional and critical timing specifications first proposed by Batcher (2) in 1979.

REFERENCES

(1) J.P. Strong; D.H.Schaefer; J.R.Fischer; K.R.Wallgren; P.A.Bracken: "The Massively Parallel Processor and Its Applications", 13th Int'l Symposium on Remote Sensing of Environment,April 1979 ERIM, Ann Arbor.

(2) Dr K.E.Batcher: "MPP - A Massively Parallel Processor", 1979 Int'l Conference on Parallel Processing, August 1979, IEEE Catalog No. 79CH1433-2C.

(3) J.Tsoras: "The Massively Parallel Processor (MPP) Innovation in High Speed Processors" AIAA Computers in Aerospace Conference III, October 1981

EFFICIENT PARALLEL ALGORITHMS FOR PROCESSOR ARRAYS*

Kuang-Hua Huang and Jacob A. Abraham

Coordinated Science Laboratory
University of Illinois
Urbana, Illinois 61801

Abstract

With the advent of VLSI technology, it is possible to provide extremely high but inexpensive computational capability with a system consisting of a large number of identical processors organized in a simple, regular structure. In order to exploit the high computation capability of the arrays, however, it is important to employ an efficient parallel algorithm. In this paper a measure is proposed which can calculate the efficiency of an algorithm performed in a processor array. This measure is used to compare several proposed array architectures for a variety of algorithms. Finally, efficient parallel algorithms for recursive filtering problems, matrix-vector multiplication, and matrix multiplication are also proposed.

1 Introduction

Problems such as weather prediction, seismic data analysis, and signal and image processing have to process extremely large amounts of data, but even the fastest existing computer cannot satisfy these demands [1]. A solution to the need for high computational power is the connection of a large number of identical processors or processing elements (PEs). Each PE has limited private storage, and in order to not restrict the number of PEs placed in an array, each PE is only allowed to be connected to some neighboring PEs. Thus, all PEs are arranged in a well organized structure such as a linear array or two-dimensional array. With VLSI technology, the processor arrays can be implemented in one chip or in a number of identical chips, and the hardware cost increases only linearly with the number of processors in the array.

Systems using a large number of PEs include the MPP (massively parallel processor) [2], the CLIP family [3], and systolic arrays [4,5]. We refer to these arrays as processor arrays in this paper. They are usually used as peripheral processors which perform computation intensive tasks. Figure 1 shows a typical processor array employed in a computer system. All the data transferred between the host system and the processor array has to pass through the bus and PEs in the boundary of the processor array; this may cause some of PEs to be idle at some time. However, the data transfer in a processor array can be overlapped with computation.

*This research was supported by the Naval Electronics Systems Command under VHSIC contract N00039-80-C-0556.

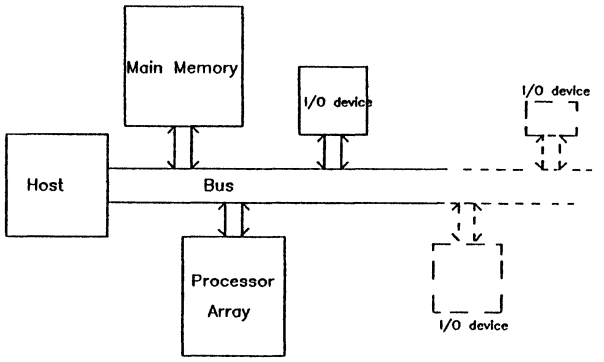


Fig. 1. A processor Array as a Peripheral Processor in a General System

It is therefore important to employ an efficient parallel algorithm to exploit the high computation capability of the arrays and reduce their idle time. In this paper, the efficiency of parallel algorithms for a computation task performed in a processor array is investigated. A measure of efficiency is proposed in section 2. It is used to measure the efficiency of some parallel algorithms in section 3. Section 4 proposes new algorithms and processor arrays to obtain a better performance for the computation tasks which are discussed in section 3.

2 The Criteria for a Measure of Efficiency for Parallel Algorithms

In this section a measure will be given for the efficiency of an algorithm when it performs a computation in a processor array. It is obvious that the number of required PEs (P) and the turnaround time (T) of the computation are two of the factors which affect the efficiency of a parallel algorithm. Another factor is the required data transfer bandwidth of the algorithm to send data into the array; an algorithm requiring a large data transfer bandwidth easily saturates the bus in Figure 1, and reduces the system performance. The data transfer bandwidth, B, is defined as the maximum number of words which have to be transferred through the I/O ports of the boundary PEs in a time unit (a time unit is defined as the period of time a PE performs an operation). Furthermore, the importance of the bandwidth is more obvious when a processor array is applied in a real-time environment with a large volume of data.

Consider a computation task with C operations which requires the transfer of I input and output operands for its execution, in a processor array consisting of P PEs. The turnaround time (T) is

0190-3918/82/0000/0271$00.75 © 1982 IEEE

the time from the beginning of transferring the task to a processor array until the result is sent back to the host. This time should satisfy equation (1) below, since the completion of an operation requires one time unit, and up to P operations can be done by the processor array in each time unit. The turnaround time, T, should also be greater than or equal to the number of words that can be transferred on the bus per time unit, I/B, to complete the data transfer.

$$T \geq C/P \qquad (1)$$

$$T \geq I/B \qquad (2)$$

From (1) and (2) we get,

$$PBT^2 \geq CI. \qquad (3)$$

The product of P, B, and T^2 is the Space-Time-Bandwidth complexity of an algorithm executed in a processor array. Equation (3) shows that product CI is the lower bound of the complexity PBT^2; an optimal algorithm has a value of PBT^2 approaching the lower bound.

The ratio of the complexity, PBT^2, of an algorithm to its lower bound CI, represented as R, is a measure of the efficiency of the algorithm. In the rest of this paper, the complexity PBT^2 and ratio R are used to measure the efficiency of an algorithm. The lower the value of PBT^2, the higher the performance of the algorithm in some sense; also, R=1 implies that the algorithm is optimal, and a large value of R means that the algorithm is inefficient.

3 Measuring the Efficiency of Some Systolic Algorithms

The systolic array architecture [4,5] proposed by Kung is a kind of special purpose processor array. The systolic algorithms provide well organized data flow through the arrays; once a piece of data is sent into a systolic array, it passes through the array and is fully exploited until its associated computations are done. Thus, more PEs can be kept busy and the communication requests between the array and the host system are reduced to a minimum. These are the primary factors which realize a high system performance.

The PE primarily used in systolic arrays is an inner product step processor which consists of three registers: R_a, R_b, and R_c. These registers are used to perform the following multiplication and addition in one time unit: $R_c = R_c + R_a * R_b$. Two different geometries of inner product PEs, which Kung defined and called type-A and type-B, are shown in Figure 2 (a) and (b).

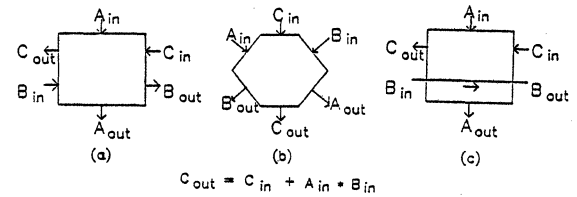


Fig. 2 Three Types of Inner Product Step Processors (a) Type-A, (b) Type-B, (c) Type-C.

In the sections 3.1 through 3.3, the efficiency of the systolic algorithms, for problems such as recursive filtering, matrix-vector multiplication and matrix multiplication, is examined. The systolic algorithms perform well when processing narrow band matrices* (i.e. when processing band matrices with the widths of the bands much less than the dimensions of the matrices) [5]. For dense matrix operations the advantages of the systolic algorithms decrease. We will examine the efficiency of matrix-vector and matrix multiplication algorithms in processing both narrow band and dense matrices; the boundary between the two cases will also be considered.

3.1 Matrix-Vector Multiplication

Consider a matrix-vector multiplication

$$Y = A*X \qquad (4)$$

where A is a n by n matrix with elements $a_{i,j}$, X and Y are two n-by-1 vectors with elements x_j and y_i respectively. The operation can be performed as follows:

$$y_i = \sum_{j=1}^{n} a_{i,j}x_j \qquad 1 \leq i \leq n \qquad (5)$$

Narrow Band Matrix-Vector Multiplication

When the matrix A in the equation (4) is a band matrix with the width of the band, $W << n$, the number of computations, C, is approaching $W*n$ and the number of the words, I, is approaching $(W+2)*n$, the lower bound of PBT^2 is

$$CI \cong (W^2+2W)*n^2.$$

The systolic algorithm for matrix-vector multiplication in [4,5] required W processors, (W/2+1) units of bandwidth, and (2n+W) time units. Thus,

$$PBT^2 = W*(W/2+1)*(2n+W)^2 \cong 2*(W^2+2W)*n^2.$$

The ratio R is about 2.

Dense Matrix-Vector Multiplication

If the matrix A in the equation (4) is a dense matrix, the value of C is n^2 and I is n^2+2n; the lower bound of PBT^2 is

$$CI = n^4+2n^3.$$

The systolic algorithm for matrix-vector multiplication in [4,5] required 2n-1 processors, n/2+1 of bandwidth, and 3n time units. Thus,

$$PBT^2 = (2n-1)(n/2+1)(3n)^2 \cong 9n^4 + 13.5n^3 - 9n^2.$$

The value of R is about 9 for large n. All the formulas shown in this section are also shown in Table 1 (a) and (b) for comparison with the new parallel algorithms proposed in section 4.

3.2 Recursive Filtering Problems

Another application of the systolic array architecture is in evaluating a recurrence equation which is used, for example, for recursive digital filtering in signal processing problems. An m-th order recurrence problem is defined as

$$x_i = F_i(x_{i-1}, \ldots, x_{i-m}) \quad \text{for } i \geq 1 \qquad (6)$$

where F_i is a given recurrence function and x_i is calculated from its m predecessors. Assume x_i ($i \leq 0$) is given.

---

*A band matrix is a matrix with elements $a_{i,j}$ for which

$a_{i,j}=0$ if $j>i+p$ or $i>j+q$ $\quad 1 \leq i,j \leq n$ and $1 \leq p,q \leq n$;

the width (W) of the band matrix is (p+q-1).

In this computation, C is m*n and I is n+m; the lower bound of the PBT^2 is found as follows:

$$CI = mn^2 + m^2n.$$

The complexity PBT^2 for the algorithm in [4] is

$$m(1)(2n)^2 = 4n^2m.$$

The value of R is about 4 for this algorithm.

3.3 Matrix Multiplication

A matrix multiplication is represented as

$$C=AB \qquad (7)$$

where matrices A, B, and C are n by n band matrices with elements $a_{i,j}$, $b_{j,k}$, and $c_{i,k}$ respectively. (A dense matrix is a special case of a band matrix with a full band width.) Let W_1 and W_2 be the widths of band matrices A and B.

The operation of equation (7) can be performed by calculating

$$c_{i,k} = \sum_{j=1}^{n} a_{i,j} * b_{j,k} \text{ for } 1 \le i,k \le n. \qquad (8)$$

Narrow Band Matrix Multiplication

For a band matrix multiplication represented in equation (7) with the condition $W_1, W_2 << n$, the product CI can be derived as follows:

$$C \cong W_1 * W_2 * n$$
$$I \cong 2(W_1+W_2)n$$
$$CI \cong 2(W_1+W_2)W_1 * W_2 * n^2.$$

The complexity PBT^2 of the matrix multiplication systolic algorithm can be derived as follows:

$$P = W_1 * W_2$$
$$B = 2(W_1+W_2)/3$$
$$T = 3n+M \qquad M = \min(W_1,W_2)$$

and $PBT^2 = 6(W_1+W_2)W_1 * W_2 n^2$

therefore, R = 3

Dense Matrix Multiplication

For a dense matrix multiplication, the product CI can derived be as follows:

$$C = n^3$$
$$I = 3n^2$$
$$CI = 3n^5$$

The complexity PBT^2 of the systolic algorithm in [4,5] can be derived as follows:

$P = 3n^2$ (Only $3n^2$ out of $4n^2$ processors contribute toward the computation.)

$B = 2n$

$T = 5n$

and $PBT^2 = 150 * n^5$,

with R = 50

All the equations shown in this section are also shown in Table 2 (a) and (b) for comparison with other matrix multiplication parallel algorithms.

3.4 Remarks

The values of PBT^2, CI, and R derived in sections 3.1, 3.2, and 3.3 suggest that the systolic algorithms might be improved to obtain lower PBT^2 values and better performance.

From [4,5], we know that the systolic algorithms do not pipe data elements into every processor in every time unit in order to synchronize with other data streams when performing the expected computations. This, however, idles one-half to two-thirds of the processors at any given time.

Three straightforward methods can be used to overcome this problem. First, independent, functionally equivalent computations can be interleaved in the array to obtain high throughput. However, functionally equivalent computational tasks do not always exist in the system at the same time. Another method is to partition a computation into independent computations of the same size and interleave them in the array. Unfortunately, not all computations can be partitioned into independent computations; also, some overhead must be paid for end conditions. Finally, one processor element can be used to do the job of two (or three) adjacent processors, since the others are idle all the time in the existing schemes. Although this increases the efficiency of the system, this method still does not exploit all of the inherent parallelism in the given computation and will require a complicated data transfer and control scheme.

4 New Efficient Parallel Algorithms and Processor Arrays

A data broadcast concept is introduced in section 4.1. Incorporating this concept into processor arrays for processing recursive filtering problems, matrix-vector multiplication, and dense matrix multiplication results in better performance; these are discussed in sections 4.3, 4.2, and 4.4, respectively. In addition, more efficient algorithms for matrix multiplications using the two-dimensional hexagonal-connected systolic array [5] are given in section 4.4.

4.1 Data Broadcast

Data broadcast is defined as sending a data element to all processors at the same time in a multiprocessor system; it can be achieved by connecting a common bus to all processors in the system. Data broadcast to each PE may not make sense for many computations. However, for some particular computations, it provides a better performance and is an alternative approach for reducing the communication requests between the array and the host. Sections 4.2 and 4.3 will show that the processor arrays with data broadcast capability (called broadcast processor arrays) provide better performance than the systolic arrays in [4,5] when they perform the matrix-vector multiplication and recursive filtering problems.

A new inner product processor with data broadcast capability is shown in Figure 2(c) and called a type-C processor in this paper. The input and output of the register R_b of the type-C processor are connected directly; thus, a data element loaded into the input of register R_b will be broadcast to the registers R_b of all the processors in that row.

Although a driver is required to drive a broadcast path, it only takes a limited area [6]. Furthermore, the propagation delay of the broadcast data transfer may be larger than that for a nonbroadcast array. However, the delay is no worse than that of a clock driving a whole chip or system, and the data is transferred in parallel

with the computation (such as with a multiplication and an addition in inner product processor arrays) which usually takes many cycles. Thus the delay in data broadcast does not significantly affect the performance of processor arrays.

4.2 Matrix-Vector Multiplication

A broadcast array for band matrix-vector multiplication is constructed by connecting W type-C processors in a row where W is the width of the band of the matrix. An array with its associated data stream is shown in Figure 3(b) for the computation in Figure 3(a).

$$\begin{vmatrix} a_{11} & a_{12} & & & \\ a_{21} & a_{22} & a_{23} & & \\ a_{31} & a_{32} & a_{33} & a_{34} & \\ & a_{42} & a_{43} & a_{44} & a_{45} \\ & & \ddots & \ddots & \ddots \end{vmatrix} * \begin{vmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \end{vmatrix} = \begin{vmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \vdots \end{vmatrix}$$

Fig. 3(a) A Matrix-Vector Multiplication with p=2,q=3

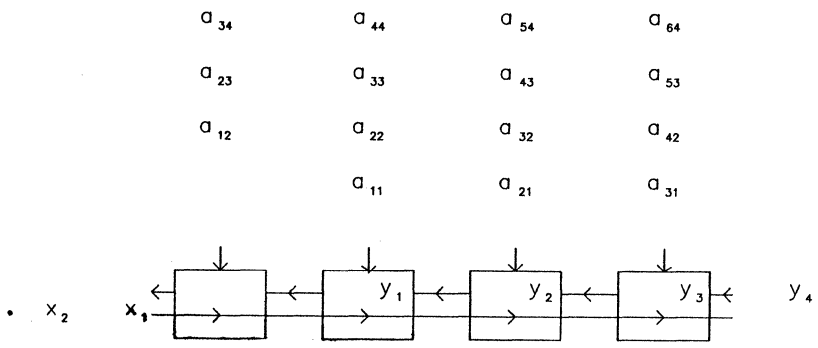


Fig. 3(b) A Broadcast Processor with Data Streams for the Multiplication in Fig. 3(a)

The algorithm of the computation is reviewed as follows. All the registers R_c in the array are initially set to zero. At time j ($j \geq 1$), x_j is broadcast to all R_b registers, the element $a_{(j+p-k),j}$ is loaded into register R_a of the k-th processor from the right and the element y_{j+q-1} enters the array from the right; the element y_{j+q-1} is initially set to zero, and it accumulates the product of $a_{i,j}$ and x_j in each processor as it flows to the left. Thus, the matrix A is loaded into the array column by column and all the computations associated with x_j are performed in the time unit j. In the remainder of this paper, the data flow of matrix A is called a Column-Diagonal Form (CDF) of the matrix A, since the matrix elements $(a_{i,j})$ are processed column by column and followed by its diagonal successors $(a_{i+1,j+1})$.

Narrow Band Matrix-Vector Multiplication

For the matrix-vector multiplication in equation (4) with W<<n, the algorithm for the broadcast array requires

$$P = W$$
$$B = W+2$$
$$T = n+W$$

and, thus,

$$PBT^2 = W(W+2)(n+W)^2 \cong (W^2+2W)n^2 \quad \text{for } W<<n.$$

From the value of CI derived in section 3, it can be seen that the ratio R approaches 1 and the algorithm is therefore optimal.

All the values of P, B, T, PBT^2 and R required by the algorithms in this section and by the algorithm in [4] to perform a narrow band matrix-vector multiplication are summarized in Table 1 (a).

Dense Matrix-Vector Multiplication

For matrix-vector multiplications with W > n, a data format transform called partial row translation (PRT) was proposed in [7] to modify the original systolic data flow for an efficiency improvement. The broadcast technique in combination with the PRT technique provides a greater efficiency when W>n.

4.3 Recursive Filtering Problems

A linearly connected broadcast array with m type-C processors and one buffer can be used to solve the m-th order recurrence problem in equation (6). In order to illustrate the idea and the improvement gained by the data broadcast technique, the example used in [4] is given here as follows:

$$x_i = a*x_{i-1} + b*x_{i-2} + c*x_{i-3} + d \qquad (9)$$

where a,b,c and d are constants.

Figure 4 shows the array structure and the data streams for this example. Before the computation starts, the constants a, b, and c are loaded and stored in registers R_a of each processor, respectively, and the constant d is stored in

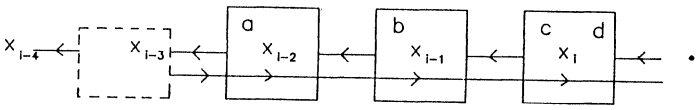


Fig. 4 A Broadcast Array for the Computation of a 3rd Order Recurrence Problem

the right most processor, for the entire computation. At the beginning of the time unit i, each xi ($i \geq 1$) with an initial value, d, emerges from the right most processor and accumulates its partial product terms as it passes through the system to the left. The left side of the linearly connected array is a buffer. It is used to latch the final value of x_i at time i ($i > m$) and to broadcast it back to R_b of all other processors in order to compute x_{i+1}, x_{i+2}, and x_{i+3}. During the first m time units, the given values of x_{-m+1}, $x_{-m+2}, \ldots, x_0$ are broadcast in sequence, one in each time unit.

A result is piped out from the buffer every time unit instead of every other time unit as in the original systolic array. The algorithm in this section requires

P = m+1
B = 1 (the constant a, b, c, and d are prestored in the array.)
T = n+2m (including m time units required to setup the constant a, b, c, and d in serial.)

thus,

$PBT^2 = (m+1)(n+2m)^2 \cong (m+1)n^2$ when $n \gg m$.

When $n \gg m$, which is the usual case in signal processing problems, the value of R approaches $(m+1)/m$; this implies that the algorithm is optimal in the limit.

4.4 Matrix Multiplication

Since the matrix multiplication systolic algorithm in [4,5] does not perform efficiently for band matrix multiplication with large W, an algorithm was proposed in [7] for an ortho-connected array, such as for the MPP system, to achieve a better performance in processing the band matrix multiplication when W>n. However, this array cannot perform narrow band matrix multiplication efficiently.

When we stack n matrix-vector multiplication broadcast processor arrays in a broadcast two-dimensional array, it can be used to perform dense matrix multiplication. Figure 5 shows the broadcast two-dimensional array and the data streams for processing a dense matrix multiplication. When the matrix A is broadcast from the left side of the array and B is fed into the array from the top edge, each processor in the array is used to accumulate the partial product terms for an element of the matrix C; the final result of the matrix C is shifted out after the computation is done. This array performs dense matrix multiplication very efficiently, but it has the same problem of inefficient narrow band matrix multiplication as the array in [7].

We therefore modify the original hexagonal-connected systolic array by reversing the data

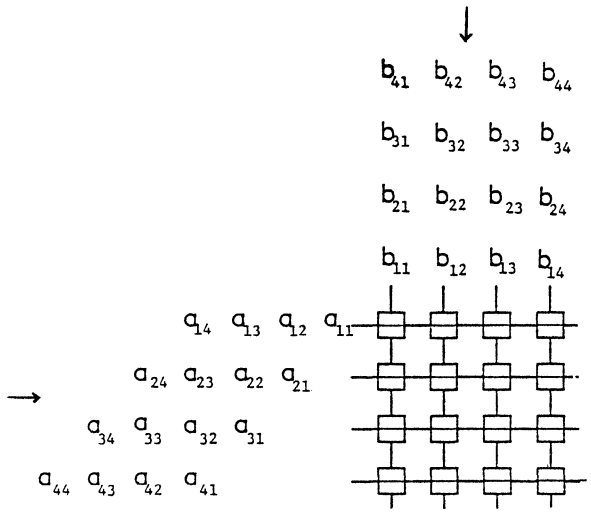


Fig. 5 The Data Streams for a Broadcast 2-Dimensional Array to Perform a Dense Matrix Multiplication

flow direction of the result. Our parallel algorithms performed in this structure provide better performance for both narrow and dense matrix multiplications.

Narrow Band Matrix Multiplication

In addition to the CDF, we introduce two more data flow formats called Row-Diagonal Form (RDF) and Backdiagonal-Diagonal form (BDF), which can be used in matrix multiplication algorithms with a greater performance. When a matrix is processed in RDF in a systolic array, the elements are loaded into PEs row by row and followed by their diagonal successors. The BDF owes its name to the fact that because the backdiagonal** of the processed matrix flows into the array in a line and each element is followed by its diagonal successor in the matrix.

For the band matrix multiplication with $W_1=3$ and $W_2=4$ shown in Figure 6(a), Figure 6(b) shows the systolic array and data streams flowing in the directions indicated by the arrows. The matrix A in RDF is loaded into the array from the left top boundary, B in CDF from the right top, and C in BDF from both sides of the top. All the elements in the bands of matrices A, B, and C move synchronously through the array in three directions. Each

** A backdiagonal of a matrix consists of those elements $a_{i,j}$ with the property $\{ a_{i,j} \mid i+j = \text{constant for } 1 \leq i,j \leq n \}$.

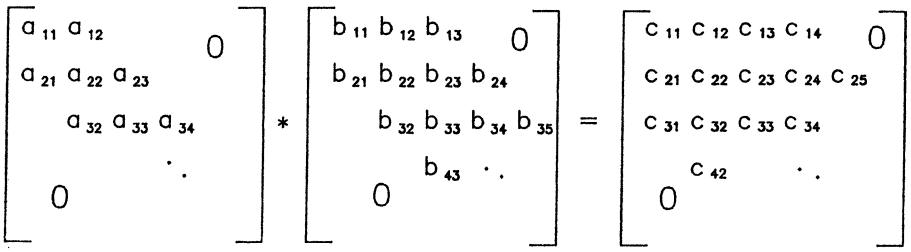


Fig. 6(a) Band Matrix Multiplication, W1=3 and W2=4

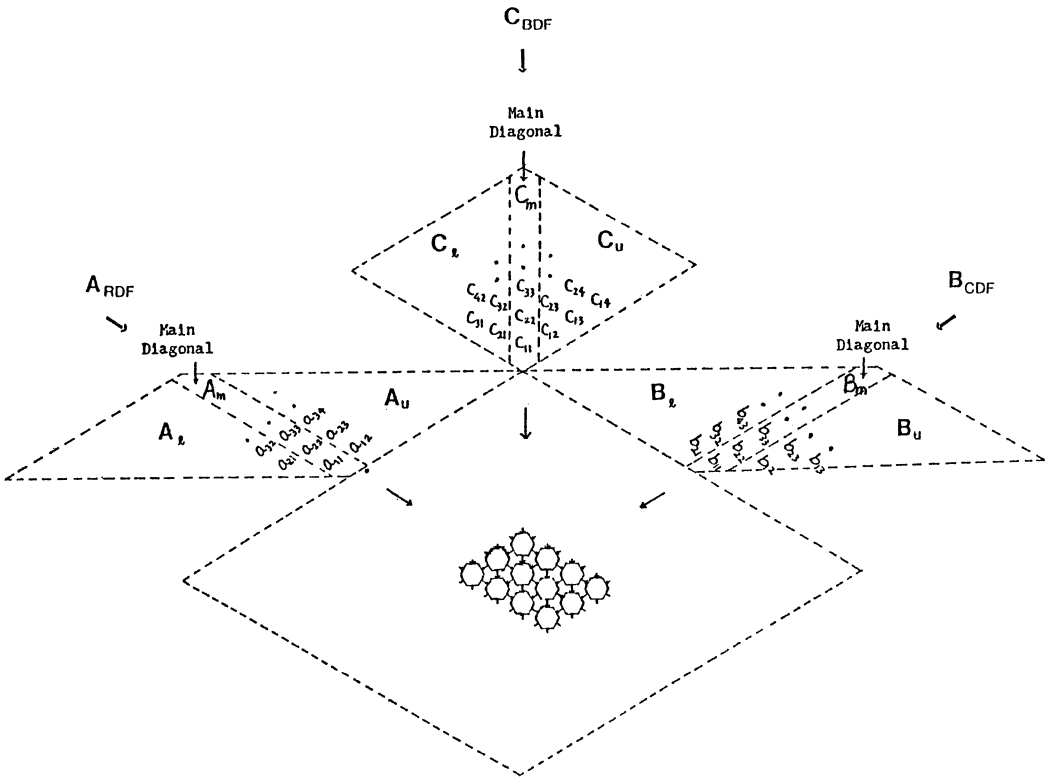


Fig. 6(b) Data Streams and Hardware Configuration for the Band Matrix Multiplication in Fig. 6(a)

$c_{i,k}$ is initialized to zero as it enters the array and accumulates all its partial product terms before it leaves the array through the bottom boundary. Figure 7(a), 7(b), 7(c), and 7(d) show four steps of the matrix multiplication in Figure 6.

The complexity PBT^2 of this algorithm can be derived from the data as follows:

$P = W_1 * W_2$

$B = 2(W_1+W_2)/3$

$T = n+M; \quad M = \min(W_1,W_2)$

and then

$PBT^2 \cong 2 * W_1 * W_2 (W_1+W_2) n^2$ when $n \gg M$

$R = 1$

The value of R is improved from 3 with the original algorithm to 1 with the algorithm in this section. The value of R of the algorithm and array in [7] is O(n); it is therefore inefficient for large n.

Dense Matrix Multiplication

For processing band matrices with W>n, the processors in the two ends of the array process only a few operations and are idle for the other times; this causes the algorithms in [4,5] and the ones in this paper for narrow matrix multiplications to be inefficient.

We now propose three new data flow formats (A_{DM}, B_{DM}, and C_{DM})*** constructed from the data formats -- matrices A in RDF, B in CDF and C in BDF. Applying these new formats to the hexagonal connected systolic array provides greater efficiency improvement for dense matrix multiplication. In order to describe the data rearrangement scheme for a multiplication with two band matrices with $W_1, W_2 > n$, Figure 6(b) shows that each of the matrices A, B, and C is divided into three parts

*** DM represents dense matrix.

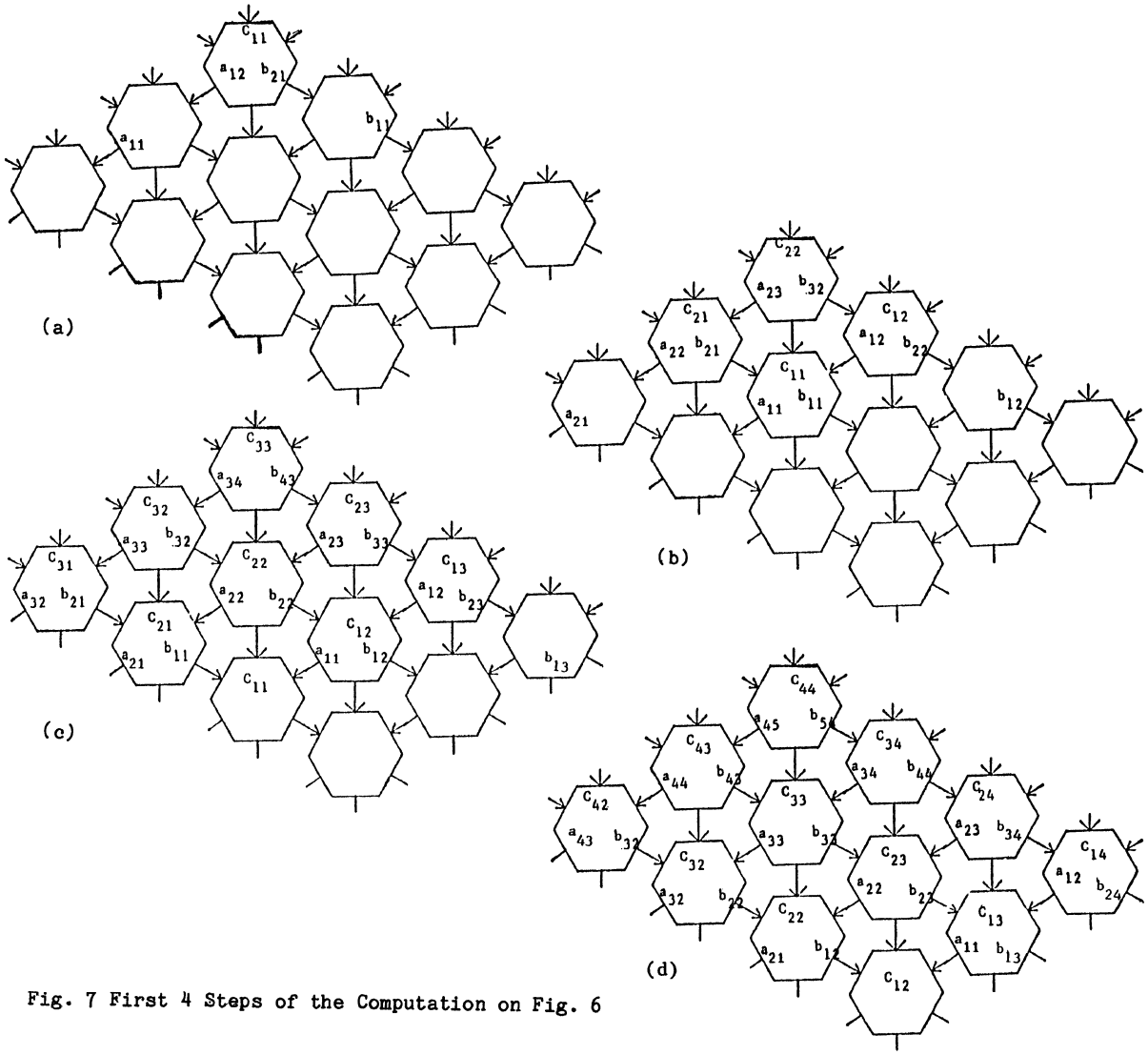


Fig. 7 First 4 Steps of the Computation on Fig. 6

-- main diagonal, upper part, and lower part; they are represented by the subscripts m, u, and ℓ respectively.

The data format A_{DM} of the matrix A is formed with A_u concatenated by A_ℓ and A_m, then by another copy of A_u as shown in Figure 8. The format B_{DM} is formed with B_ℓ concatenated by B_u and B_m, then by another copy of B_ℓ. The format C_{DM} is formed with the matrix C in BDF concatenated by C_u and C_ℓ. Both B_{DM} and C_{DM} are also shown in Figure 8.

The data in formats A_{DM}, B_{DM}, and C_{DM} are fed into the array through the boundary processors at the top left, top right and at both sides respectively as shown in Figure 8 to perform a dense matrix multiplication. Each element in the matrix C is initially set to zero the first time it is fed into the array. It passes through n processors in one or two columns to accumulate its n partial product terms. From Figure 8, we know only n^2 processors and 2n time units are required to perform a n-by-n dense matrix multiplication.

The values of PBT^2 and R of the algorithm can be derived from the values of P, B, and T as follows:

$$P = n^2$$
$$B = 6n$$
$$T = 2n$$

and then

$$PBT^2 = 24*n^5$$
$$R = 8$$

All the values of P, B, T, PBT^2, and R for different algorithms and arrays to perform dense matrix multiplication are summarized in Table 2(b). It shows that the broadcast two-dimensional array performs most efficiently for dense matrix multiplication, but it is not suited for narrow band matrix multiplication. The two algorithms proposed in this section can be chosen to obtain the best performance for matrix multiplication with distinct values of W in the same array. Using the algorithms in [5], [7], and this paper to perform matrix multiplications for various values of W, the values of the ratio R are plotted

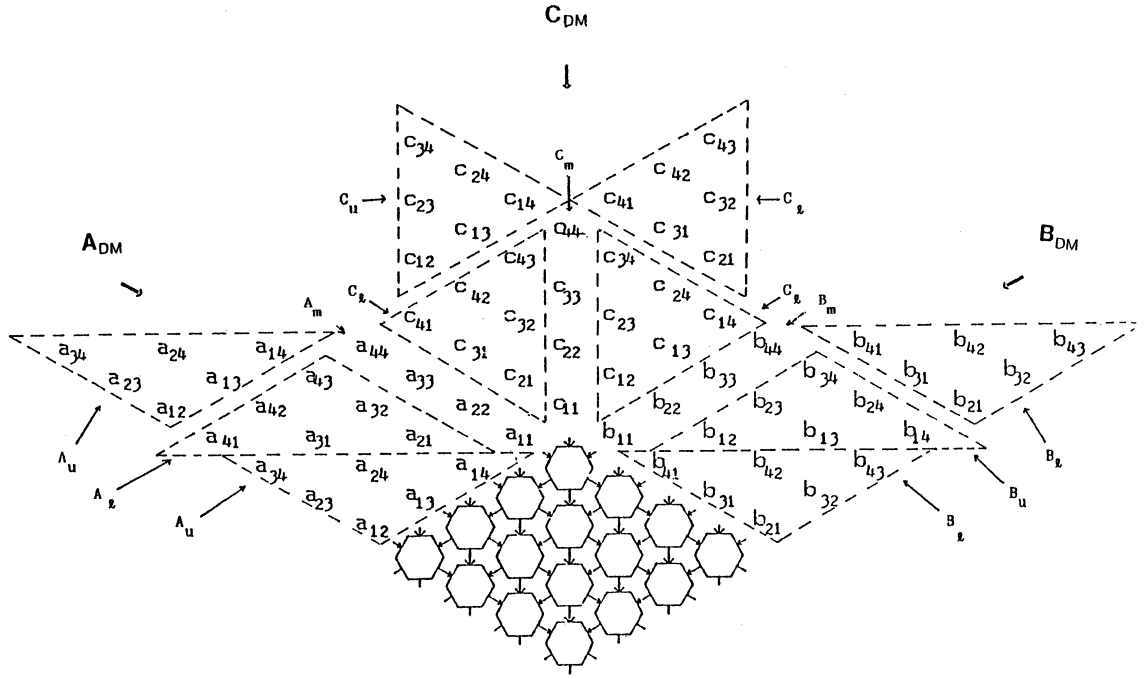


Fig. 8 The Data Streams A_{DM}, B_{DM}, and C_{DM} for a 4-by-4 Dense Matrix Multiplication

in Figure 9. It shows that the algorithms proposed in this paper provide the highest efficiency when performing matrix multiplication. They are also the most efficient for the cases between narrow-band and dense matrices.

Conclusion

The measures PBT^2 and R have been proposed in this paper to evaluate the efficiency of an algorithm when it is performed on a processor array. A data broadcast concept was introduced for processor arrays to obtain better performance for particular computations. Several parallel algorithms and processor arrays have been presented to obtain an efficient performance for the recursive filtering problem, matrix-vector multiplication, and matrix multiplication.

Acknowledgement

The authors like to thank Prof. Janak H. Patel for his contribution to the idea in Section 4.2.

References

[1] Sugarman, R. '"Superpower" computers,' IEEE Spectrum, April 1980, pp. 28-34.

[2] Batcher, K. E. 'Design of a Massively Parallel Processor,' IEEE Trans Computers, Vol. C-29, No. 9, Sept. 1980, pp. 836-840.

[3] Fountain, T. J. 'Toward CLIP 6 - an Extra Dimension,' IEEE Computer Society workshop on Computer Architecture for Pattern Analysis and Image Database Management, Hot Springs, Virginia, Nov. 1981, pp. 25-30.

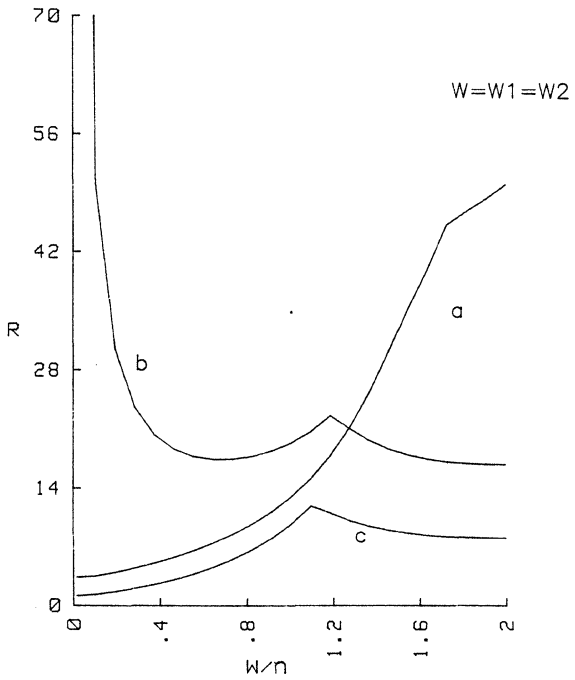


Fig. 9 The Ratio R for Matrix Multiplication vs W/n
(a) the Original Systolic Algorithm
(b) the Algorithm for the Ortho-Connected Array with PRT
(c) the Algorithm Proposed in this Paper

[4] Kung, H.T. 'The Structure of Parallel Algorithms,' Advances in Computers. vol 19: Academic Press, 1980, pp. 65-112.

[5] Kung, H.T. and Leiserson, C.E. 'Systolic Arrays (for VLSI),' Sparse Matrix Proc. 1978, Society for Industrial and Applied Mathematics, 1979, pp. 256-282.

[6] C.A. Mead and L.A. Conway, Introduction to VLSI Systems. Massachusetts: Addison-Wesley, 1980.

[7] Priester, R. W., Whitehouse, H. J., Bromley, K. and Clary, J. B. 'Signal Processing with Systolic Arrays,' 1981 International Conference on Parallel Processoring, Bellaire, Michigan, pp. 207-215.

	systolic algorithm in (4)	broadcast algorithm in this paper
P	W	W
B	W/2+1	W+2
T	2n+W	n+W
PBT^2	$\cong 2(W^2+2W)n^2$	$\cong (W^2+2W)n^2$
R	2	1

Table 1(a) Summary of the values of P, B, T, R, and PBT^2 for band matrix-vector multiplication for $W \ll n$, with $C = W*n$, $I = (W+2)n$, and $CI = (W^2+2W)n^2$.

	systolic algorithm in [4]	broadcast algorithm in this paper	algorithm in [4] with PRT	broadcast algorithm with PRT
P	2n-1	2n-1	n	n
B	n/2+1	n+2	n/2+1	n+2
T	$3n_4$	$2n_4$	3n	2n
PBT^2	$\cong 9n^4$	$\cong 8n^4$	$\cong 4.5n^4$	$\cong 4n^4$
R	9	8	4.5	4

Table 1(b) Summary of the values of P, B, T, R, and PBT^2 for dense matrix-vector multiplication, with $C = n^2$, $I = (n+2)n$, and $CI = n^4+2n^3$.

	systolic algorithm in [4]	ortho-connected algorithm in [8]	the algorithm in section 4.4 for band matrices
P	W_1*W_2	nM	W_1*W_2
B	$2(W_1+W_2)/3$	W_1+W_2	$2(W_1+W_2)$
T	3n+M	3n+M	n+M
PBT^2	$\cong 6W_1*W_2(W_1+W_2)n^2$	$\cong 9M(W_1+W_2)n^3$	$\cong 2W_1*W_2(W_1+W_2)n^2$
R	3	O(n)	1

Table 2(a) Summary of the values of P, B, T, R, and PBT^2 for band matrix multiplication for $W \ll n$, with $C \cong W_1*W_2*n$, $I \cong 2(W_1+W_2)n$, and $CI \cong 2(W_1+W_2)W_1*W_2*n^2$.

	systolic algorithm in [4]	ortho-connected algorithm with PRT [8]	algorithm in section 4.4 for band matrices	algorithm in section 4.4 for dense matrices	broadcast two-dimensional processor array algorithm
P	$3n^2$	n^2	$3n^2$	n^2	n^2
B	2n	2n	4n	6n	2n
T	5n	5n	3n	2n	3n
PBT^2	$150n^5$	$50n^5$	$108n^5$	$24n^5$	$18n^5$
R	50	17	36	8	6

Table 2(b) Summary of the values of P, B, T, R, and PBT^2 for dense matrix multiplication, with $C = n^3$, $I = 3n^2$, and $CI = 3n^5$.

PARALLEL SIMULATION BY MEANS OF A PRESCHEDULED MIMD- SYSTEM FEATURING SYNCHRONOUS PIPELINE PROCESSORS

M. Tadjan, R.E. Buehrer, W. Haelg

ETH (Swiss Federal Institute of Technology)
Institut für Reaktortechnik
CH-8092 Zürich, Switzerland

Abstract --- The software package PSCSP (power-series continuous simulation program) for the simulation of continuous systems being developed at ETH achieves a great deal of parallelism by making use of the power-series integration technique. A parallel version of PSCSP currently runs on the ETH-Multiprocessor EMPRESS.

The new processor scheduling strategy presented in this paper was developed in order to further improve the processing time of this integration method. A summary of different performance studies is presented, demonstrating that this method is very well suited for an MIMD- Parallel processor consisting of several synchronous pipeline processors connected to a powerful EMPRESS-type intercommunication memory. A description of such an architecture is given.

Introduction

The software package PSCSP (power-series continuous simulation program)[a] for the simulation of continuous systems makes use of the power-series integration method. In addition to a fundamental high integration speed this technique offers a great amount of parallelism ideally suited for an implementation in an appropriate MIMD- (multiple-instruction stream - multiple-data stream) parallel processor. Earlier predicted speed improvements [2] are currently verified on the ETH-Multiprocessor EMPRESS [5].

To give a survey of some features of PSCSP which are important in this context we first describe briefly the method of integrating by means of power-series and the parallelization concept used in the parallel version of PSCSP. The improved scheduling strategy is explained afterwards and - illustrated by a practical example - the theoretical gain in speed achievable by this technique is presented. In the final section a parallel processor consisting of synchronous pipeline processors connected by an EMPRESS-type intercommunication memory (intercom) is described.

[a] PSCSP, designed and written at ETH [3], actually exists in two versions: the sequential version is intended for implementation on a standard computer such as the PDP-11, the DEC-10, etc., while the parallel version is implemented on the ETH-Multiprocessor EMPRESS [3,4,5].

Parallel integration by means of power-series

A system of n first order differential equations is given as follows:

$$\frac{dy_i}{dx} = f_i(x,y_1,\ldots,y_n) \; ; \; (i = 1,2,\ldots n) \qquad (1)$$

where

$$y_i(x_o) = y_{io}$$

are given as initial values.

The solution for $y_i(x)$ at $x=x_o+h$ using the method of power-series expansion is

$$y_i(x) = \sum_{\nu=o}^{\infty} y_i(x_o)^{\nu} \cdot \frac{(x-x_o)^{\nu}}{\nu!} \qquad (2)$$

($f_i(x_o,y_1,\ldots,y_n)$ must be holomorphic in the interval $(x-x_o)$). In order to proceed numerically, (2) is separated as follows:

$$y_i(x) = \sum_{\nu=o}^{\nu_o} y_i(x_o)^{\nu} \cdot \frac{h^{\nu}}{\nu!} + R_i(\nu_o,x) \qquad (3)$$

When all $R_i(\nu_o,x)$ satisfy a given convergence criterion [3] the expansion is terminated and calculation of the next $y_i(x)$ is started with $x:=x+h$, $x_o:=x_o+h$). Otherwise, ν is further increased and an additional term is added to (3).

In [2],[3] it is shown that evaluation of the function f_i in (1) can be separated into individual tasks having the nature of simple arithmetic operations (e.g. r=p+q, r=p*q etc.) or elementary functions (e.g. r=g(p)). These tasks can be standardized and stored in a program library. (This eliminates the need for calculating specific and complicated recursion formulas for each individual f_i; instead one can define simple recursion formulas for the tasks mentioned. As a result calculation of the coefficients in (2) is rather trivial.) In general, depending on the structure of the problem to be simulated, several of these tasks are independent of each other and can therefore be calculated simultaneously (first stage of parallelism). As shown in the following example, the recursion formulas of the recursive tasks additionally show an inherent parallelism (second stage of parallelism).

Example

$$r = p * q \qquad (4)$$

0190-3918/82/0000/0280$00.75 © 1982 IEEE

By defining $r_\nu = r^{(\nu)} \cdot \frac{h^\nu}{\nu!}$ $p_\nu = p^{(\nu)} \cdot \frac{h^\nu}{\nu!}$

$$q_\nu = q^{(\nu)} \cdot \frac{h^\nu}{\nu!}$$

the corresponding recursion formula reads,

$$r_\nu = \sum_{s=o}^{\nu} p_\nu * q_{\nu-s} \quad ; \quad (\nu > o) \tag{5}$$

In [3] it is shown that all other recursive tasks lead to similar recursion formulas. The calculation of such sums of products can be done quite easily by means of "recursive doubling". The critical path length of such schemes is directly dependent on ν.

A sample calculation of component $f_{i\nu}$ of $f_i(x)$ is illustrated in figure 1. As a consequence the time for calculating $f_{i\nu}$ is also dependent on ν.

Performance improvement by means of a new processor scheduling strategy

Reduction of the critical path length

The calculation time for $f_{i\nu}$ (see figure 1) can be reduced substantially if one can find a scheme where a complete pass is independent of ν (i.e. the critical path length is constant). The existence of such a strategy is demonstrated below, using recursion formula (5),

$$r_\nu = \sum_{s=o}^{\nu} p_s * q_{\nu-s}$$

The formula can be separated as follows

$$r_\nu = p_o * q_\nu + p_\nu * q_o + \underbrace{\sum_{s=1}^{\nu-1} p_s * q_{\nu-s}}_{R_\nu} \tag{6}$$

The equivalent separation at $(\nu+1)$ can be written

$$r_{\nu+1} = p_o * q_{\nu+1} + p_{\nu+1} * q_o + \underbrace{\sum_{s=1}^{\nu} p_s * q_{\nu+1-s}}_{R_{\nu+1}} \tag{7}$$

where

$$R_{\nu+1} = p_1 * q_\nu + p_\nu * q_1 + \underbrace{\sum_{s=2}^{\nu-1} p_s * q_{\nu+1-s}}_{R^*_\nu} \tag{8}$$

In the standard PSCSP r_ν is calculated in one pass and $r_{\nu+1}$ in the next. The fact that $R_{\nu+1}$ of $r_{\nu+1}$ in (7) contains at most the ν'th derivatives of variables p and q makes it possible to compute $R_{\nu+1}$ already at the time r_ν is calculated. since

$$r_{\nu+1} = p_o * q_{\nu+1} + p_{\nu+1} * q_o + C, \text{ where } C = R_{\nu+1} \tag{9}$$

is completely independent of ν, recursive tasks are reduced to nonlinear ones, provided $R_{\nu+1}$ is available after r_ν has been calculated. The modified graph for the example of figure 1 is presented in figure 2. The critical path length is constant.

Processor scheduling strategy

Given

- a task system B=(T,<) in which T= $(A_1,\ldots,A_n)$ equals a set of tasks and < is the partial ordering relation.
  $A_i<A_j$ implies that A_j cannot start execution prior to completion of A_i.
- a weighting function $\alpha(A_i)$, representing the execution time $\tau_i = \alpha(A_i)$.
- a fixed number of identical processors, n.

The objective is to find a partition $T_1 \ldots T_n$ of T such that the largest execution time on any processor

$$t_{max} = \max_{\forall i} \left(\sum_{\tau_j \varepsilon T_i} \tau j \right) \tag{10}$$

is minimized. Considerable attention must be paid to the development of fast heuristic scheduling algorithms, yielding suboptimal results [7,8].

Whenever a simulation problem has a deterministic structure (i.e. if one has a time-invariant problem) its task system is identical for each integration step. Consequently, compilation and scheduling have to be done only once. The appropriate (scheduling) information for each individual processor can be computed in advance and loaded into the corresponding processor memories.

The standard parallel version of PSCSP is able to create graphs for deterministic problems as outlined in figure 1. By means of the previously discussed method of reducing the critical path length graphs like the one shown in figure 2 are obtainable.

In order to compare both versions in terms of total execution time t_{max} (10) a model of a parallel processor was defined and simulated on a PDP-11. This model consists of a number of synchronously working pipeline processors and an intercommunication memory intercom as will be outlined in the hardware description later on. The arithmetic of each processor is a dynamic multifunction pipeline [9], whereby the number of stages can be varied in the model. The execution times of any operation (e.g. addition, multiplication etc.) within a task are variable too, but are assumed to be identical in this example. Both graphs (figure 1,2) were scheduled according to the "level algorithm" [8]. As a reference we also determined the ideal calculation time t_{ideal}, given as

$$t_{ideal} = \sum_{\forall i} \left(\sum_{\tau_j \varepsilon T_i} \tau_j \right) \tag{11}$$

(any task T_i is part of the critical path).

As expected, execution times referring to the improved graph turned out to be significantly shorter. Results for one of these examples, the "restricted three body problem" [3], are presented in figure 3.

Hardware description of an appropriate multiprocessor

The multiprocessor described below has been designed in accordance with the requirements of the integration technique just described. Components and related functions are presented in table 1. The intercom, being a slightly modified version of the one installed in the ETH-multiprocessor EMPRESS (5), consists of a quadratic organized memory matrix whereby an individual processor duplicates its data into all elements of its associated row (see figure 4). Reading is possible in all elements of its associated column. In addition, every execute processor has the facilities to write into the supervisor row (in this mode, at a specific time slot only one execute processor or the supervisor itself gets access to the corresponding write lines wl_s). As mentioned earlier, scheduling of the execute processors is done at compilation time in the supervisor processor. As a result, prior to the start of the integration part the program memories of all execute processors are loaded by the supervisor.

The synchronization of the execute processors is controled by a dedicated logic in the supervisor processor. Note that intermediate results of the "recursive doubling" and the results a_i (figure 2) are available in the execute processor region of the intercom while the results of f_{iv} are transferred at a prescheduled time slot by the appropriate execute processor to the supervisor row to be available for further processing.

Conclusions

The need for efficient algorithms and powerful computer hardware is very acute in the field of digital simulation. The outlined method of integrating differential equations by means of power-series in a prescheduled MIMD- pipeline-multiprocessor points out possible solutions for some of the problems in this field. The relatively large effort of compilation (including the scheduling of processors) is worth-while because in many simulation problems one does not often have to change the model but only related parameters. Compilation needs to be done only once for different runs, allowing an unrestricted profit from the fast execution of the integration part.

References

(1) Flynn M.J., "Some Computer Organizations and Their Effectiveness" IEEE Transactions on Computers, Vol. C-21, No. 9, September 1972, pp. 948-960

(2) Halin H.J., Buehrer R. and Haelg W., "Software Development for the ETH-Multiprocessor Project: Partial Integration of Ordinary and Partial Differential Equations by Means of Power Series" Proceedings of the Second IMACS (AICA) International Symposium on Computer Methods for Partial Differential Equations, (Lehigh University, Bethlehem, Pennsylvania), 1977

(3) Halin H.J., Buehrer R., Haelg W., Benz H., Bron B., Brundiers H.J., Isacson A., Tadjan M. "The ETH Multiprocessor Project: Parallel Simulation of Continuous Systems" Simulation, October 1980, pp. 109-123

(4) Buehrer R., "A New Type of MIMD-Type Multiprocessor Handling Two-stage Parallelism by Means of a Dynamically Configurable Architecture" In Liu M.T. and Rothstein J., editors Proceedings of the 1981 International Conference on Parallel Processing, 1981, pp. 292-293

(5) Buehrer R., "Hardware eines dynamisch konfigurierbaren Multiprozessors" PhD thesis 6930, Swiss Federal Institute of Technology Zuerich, 1981

(6) Tadjan M., PhD thesis, Swiss Federal Institute of Technology Zuerich, to be published

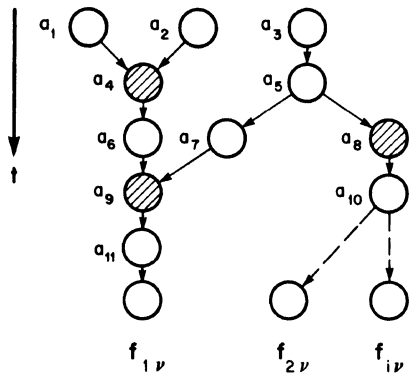
(7) Adam T.L., Chandy K.M. and Dickinson J.R., "A Comparison of List Schedules for Parallel Processing Systems" Comm. of the ACM 17, 12, 1974, pp. 685-690

(8) Coffmann E.G.Jr., Leung J.Y-T. and Slutz D., "On the Optimality of First-fit and Level Algorithms for Parallel Machine Assignment and Sequencing" Proceedings of the International Conference on Parallel Processing, Ed. J.L. Baer 1977, pp. 95-99

(9) Ramamoorthy C.V. and Li H.F. "Pipeline Architecture" Computing Surveys, Vol. 9, No. 1, 1977

Table 1: Functional survey of the multiprocessor components

Component	Function
Supervisor computer	- I/O activities - program compilation, preparation and execute processor scheduling - control of integration
Execute processors	- execution of arithmetic operations - data provision for further calculations
Intercom	- simultaneous transfer of intermediate or final results

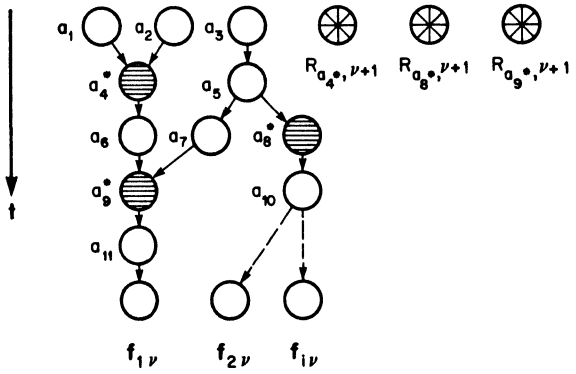


a_i ○ non-recursive task, single instr.; cpl ≠ f(ν)

a_i ▨ recursive task ; cpl = f(ν)

cpl: critical path length

Figure 1: Calculation of fi_ν : example



a_i ○ non-recursive task, single instr.; cpl ≠ f (ν)

a_i^* ● non-recursive task, multiple instr.; cpl ≠ f (ν)

$R_{a_i,\nu+1}$ ⊛ recursive task ; cpl = f (ν)

cpl: critical path length

Figure 2: Optimized calculation of fi_ν ; example

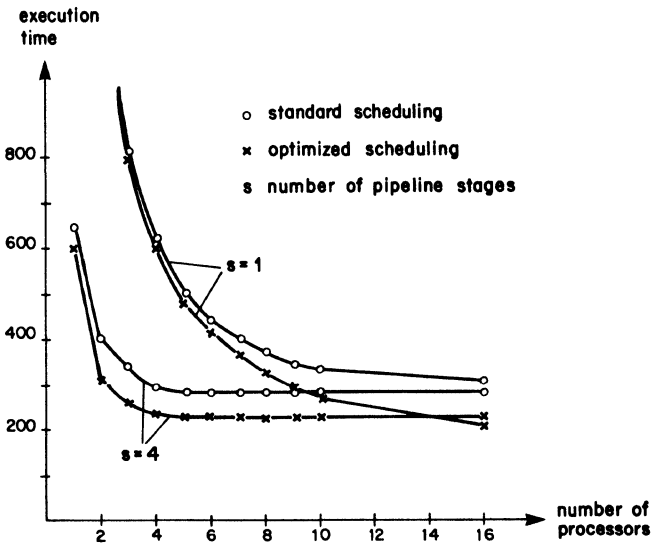
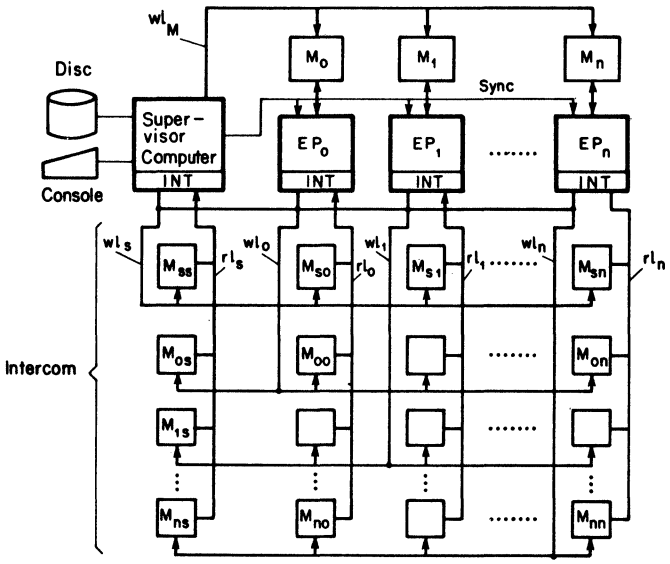


Figure 3: Performance improvement by means of an optimized scheduling strategy



EP_i : Execute (pipeline) processor

wl_i : write lines rl_i: read lines

M_i : Program memory

INT_i : Intercome Interface Sync: Synchronize signal

M_{si} : Dual Port Memory (≈ 32 k words)

M_{ij} (i≠s) : Dual Port Memory (≈ 64 words)

Figure 4: Architecture of the Multiprocessor

Pipelining Array Computations for MIMD Parallelism: A Function Specification.

by Dennis Gannon
Department of Computer Sciences, Purdue University
West Lafayette, Indiana

Introduction

This paper describes a formal link between the data flow model of MIMD computation and the design and analysis of systolic systems. To establish the relationship between these two models of computation we describe a small set of functional operators which will enable us to express many vector and array algorithms as networks of interacting data-driven processes. Using these tools, we will then show that the data flow graphs of many functions can ge reformulated as "systolic" systems.

The main result of the paper is a theorem which gives conditions which will garentee that the systolic version of the computation graph will perform asymptotically as fast as a fully concurrent execution of the original data flow graph.

Vector Valued Data Flow Operators.

The majority of highly parallel computation is based on array and vector data structures formed from primitive scalar types such as the integers Z, the reals R, booleans B, and complex number C. Let $\{T_i \ \ i=1,,n\}$ be a set of primitive data types. Define the direct product type, written as

$$\prod_{i=1}^{n} T_i \quad or\ as \quad T_1 x T_2 x \cdots x T_n,$$

to be the set of n-tuples

$$(x_1,x_2,x_3,\ldots,x_n) \quad with \quad x_i \in T_i \quad for \ i=1..n.$$

More generally we define a *domain* recursively as either

1 A primitive scalar type such as Z, R, B, or C.

2 The direct direct product of a finite set of domains.

For example, the set of real n-vectors is $\prod_{i=1}^{n} R$ which is denoted by R^n and the set of n by m integer arrays is $\prod_{i=1}^{m} Z^n$. An array of 'records' such that each record contains an integer, a boolean, and 2 reals could be described as $\prod_{i=1}^{n}(ZxBxR^2)$. The individual components of a member of some domain will addressed by indexing that describes the position of the component in the structure.

All of the programs constructed below will be described as "functions"

$$f : D_1 \longrightarrow D_2$$

from one domain D_1 to another D_2. More precisely, f will be a structured set of interacting processes that collectively define a finite state machine (a function with memory in the sense described by Ackerman [Acke82].) The basic components of a function are simple sequential processes that will be called cell functions. Each cell function performs a "small" set of scalar operation on a "small" set of variables. For example, the expression

```
Cfunction f(x, y, z: R): R;
 {
   f := y - (x+z);
 };
```

defines a function $f : R^3 \longrightarrow R$ which could be graphically represented as shown in figure 2.1.

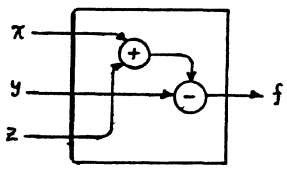


Figure 2.1 Cell function node

The inputs to a cell function represent queues of values. We shall define the execution semantics to be data-driven, i.e. if at time t_0 all input queues to a cell become nonempty, one value is removed from the head of each queue and at time t_0+1 the cell produces output values.

Cell functions represent small units of sequential computation. Explicit parallelism is expressed through the application of higher order functional operators. Of the many possible classes of operators four are described below.

A. The Product Operator. The simplest form of parallelism is the vectored application of a function. Given

$$f_i : D_{i_1} \longrightarrow D_{i_2} \quad i=1..n$$

The product operator defines a function

$$\prod_{i=1}^{n} f_i : \prod_{i=1}^{n} D_{i_1} \longrightarrow \prod_{i=1}^{n} D_{i_2}$$

which represents the concurrent operation of the n functions f_i.

B. Permutation and Data Movement Operators. Many important computations cannot be specified completely without defining certain complex data movement operations.

For example, the Rotation operation executes a right circular shift of a product structure.

$$Rotate_k(x_{1_2} \cdots x_n) \equiv (x_{n-k+1},x_{n-k+2},\ldots,x_n,x_1,\ldots,x_{n-k})$$

Many other usefull permutations can be defined but they will not be needed here.

C. Iterated Composition: The Chain Operator. Given a function

$$f : D_a x D_b \longrightarrow D_a x D_c$$

for some structured domains D_a, D_b and D_c, the chain operator defines a mechanism to iterate f over the values in D_a. More Specifically, if f is a function defined with the header

Research supported by NSF Grant MCS-8109512.

0190-3918/82/0000/0284$00.75 © 1982 IEEE

function f(x: D_a; y: D_b): $D_a x D_c$;

then the iteration

```
var x: Da x Db;

x := initial values;

for i := 1 to n do
  {
     x := f(x, y(i) );
  }
```

can have at least two interpretations when f is viewed as network of interacting cell functions. The simplest of meanings is given by the *chain* operator which constructs from a function f a sequence of copies of f where the output if the i^{th} copy is directed to the input of the $(i+1)^{th}$ copy. We denote this by

$$\underset{i=1}{\overset{n}{Ch}} f : D_a x \prod_{i=1}^{n} D_b \longrightarrow D_a x \prod_{i=1}^{n} D_b$$

which is represented by the graph in figure 2.2.

Any composition of the operators and function constructors described above can be viewed as defining the data flow graph of a program: graph edges correspond to the binding of function parameters and each edge represents a queue of values; graph nodes correspond to the basic cell functions. Because we have not specified any operators that may conditionaly select from a subset of input values, any function network has the property that the graph is acyclic and the order of arrival of operands to a cell node is determined by the structure and not the timing of the system. The latter property shall be referred to as

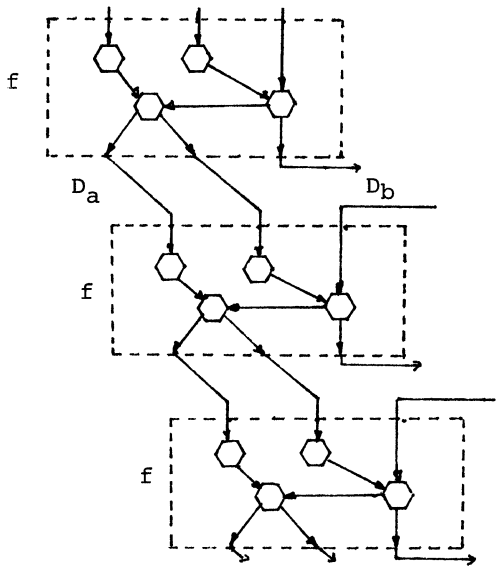


Figure 2.2. $\underset{i=1}{\overset{n}{Ch}} f\,(x, y^{(i)})$

dependence synchronization and is sufficient to guarantee that pipelining the system is entirely well defined. In particular, it implies that the "graph" may be executed on a data flow machine where only the cell function address and argument position is needed to form the "packet address".

D. The Systolic Iteration. For a single set if input $(x, y^1, \ldots, y^n)$ the low utilization of the cell functions in $\underset{i=1}{\overset{n}{Ch}} f$ represents a large memory (large silicon area) cost for any hardware implementation of this construct. The natural alternative is to allow one copy of f to be used in "feedback" loop, i.e. some of the outputs of f are connected to some of its inputs as illustrated in figure 2.3. While the resulting structure is not easy to initialize (see [Gann82] for details), it does permit the cells of f to be reused on each iteration.

As a data flow operation one can show that $\underset{i=1}{\overset{n}{Syf}}$ is dependence synchronized if and only if f is dependence synchronized. On the other hand it is not clear

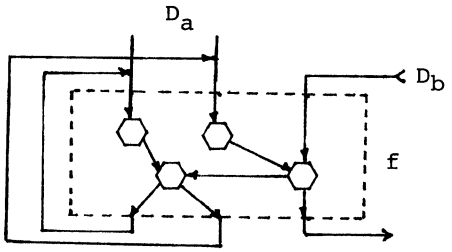


Figure 2.3 $\underset{i=1}{\overset{n}{Syf}}$ graph structure.

that the Sy operation can express all the parallelism that is provided by the chain construction. In fact, it is not hard to construct examples where $\underset{i=1}{\overset{n}{Ch}} f$ executes $O(n)$ times faster than $\underset{i=1}{\overset{n}{Syf}}$. There is, however, a situation where one can prove the the systolic iteration exhibits the same parallelism as the chain operation.

Recall that a function is said to be transitive if there exists a formal dependence of each component of the output on each component of the input. We shall say that a function f is weakly transitive if its k-fold self-composition (f^k) is transitive for some k. Weakly transitive functions occur in computations such as the L-U decomposition of a matrix, convolution based operations such as the FFT, the solution of partial differential equations, the solution of linear recurrences, and many graph algorithms such as transitive closure. In this case we have the result

THEOREM. Let f be a weakly transitive function that executes in constant time. Then

1 **The time complexity of $\underset{i=1}{\overset{n}{Ch}} f$ is o(n) and time($\underset{i=1}{\overset{n}{Syf}}$) = time($\underset{i=1}{\overset{n}{Ch}} f$).**

2 **As a data flow graph the edges of $\underset{i=1}{\overset{n}{Syf}}$ represent queues of values. These queues are of bounded length where the bound depends only on f and not on n.**

The proof is given in the report [Gann82].

The above paragraphs have stressed Data Flow semantics to describe systolic systems. The problem of transforming the data-driven semantics to a synchroneous set of processors has been considered by Cuny and Snyder [CuSn82].

To illustrate the above ideas and constructs, consider the q^{th} order linear recurrence relation

$$x_i := \sum_{j=1}^{q} a_j^i x_{i-j} \quad i:=q+1,\ldots,n$$

$$x_i := c_i \quad i:=1,\ldots,q$$

where x_i, c_i and a_j^i $i=1..n$, $j=1..n$ are all real numbers. Programmed in the standard manner shown below the sequential complexity is roughly 2nq.

```
for i:= q+1 to n do
    {
    x_q^(i) := Σ_{j=1}^{q} a_j^(i) x_j^(i-1);
    x_{q-1}^(i) := x_q^(i-1)
     ...
     ...
    x_1^(i)   := x_2^(i-1);
    write(x_q^(i));
    }
```

The superscripts indicate iteration count and are suppressed in the sequentail computation. A cell function to compute one step of the inner product formed by the summation is given by

```
Cfunction ipstep(a, x, s:R): RxR;
  {
    ipstep_2 := s + a*x;
    ipstep_1 := x;
  };
```

Applying the chain operator over the first parameter we generate the complete inner product

```
function IP(x,a: R^q): RxR^q;
  {
    IP := Ch_{i=1}^{q} ipstep(0,a_i,x_i);
  };
```

which is pictured as the network in figure 3.1.

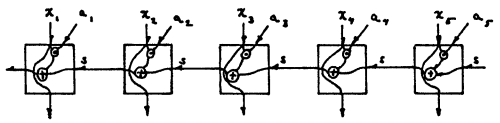


Figure 3.1 The inner product function.

The function has been constructed to compute the inner product as the first component of the result and return the values of x as the remaining q components. There is no parallelism in this function other than structural: each cell function can act only after its right neighbor acts. The complete recurrence is given by a second application of the chain operator.

$$\underset{i=1}{\overset{n}{Ch}} IP(c_q, c_{q-1}, \ldots, c_1, a_q^i, \ldots, a_1^i)$$

which is pictured in figure 3.2 below.

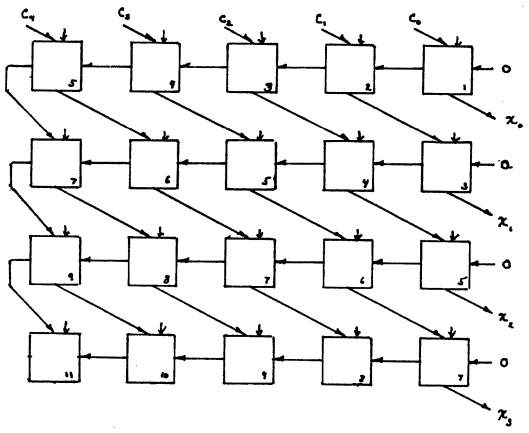


Figure 3.2 $\underset{i=1}{\overset{n}{Ch}}$IP(c,x).

Notice that this application of the chain used the associativity of the direct product operator to identify RxR^q with $R^q xR$. The reassociation of the components of the output turn IP into a weakly transitive function. Hence, one may apply the Sy operation as a replacement for the last chain operation. The result is

$$x := \underset{i=1}{\overset{n}{Sy}} IP(c_q, c_{q-1}, \ldots, c_1, a_q^i, \ldots, a_1^i)$$

which is illustrated in figure 3.3.

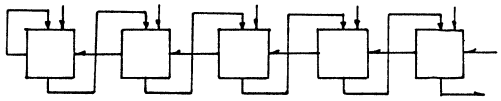


Figure 3.3 $\underset{i=1}{\overset{n}{Sy}}$IP(c,a).

This structure is identical to the systolic recurrence solver of Kung and Leiserson [KuLe80]. (There are several other derivations of systolic arrays from formal principles. See for example Kuhn [Kuhn80].)

References

[Acke82] W. Ackerman, "Data Flow Languages," Computer, Feb. 1982 vol 15. no.2.

[CuSn82] J. Cuny, L. Snyder, "The Coordination of Loop Data Flow Programs", Technical Report, Department of Computer Sciences, Purdue University, 1982.

[Gann82] Pipelining Array Computations for MIMD Parallelism: A functional Specification. Technical Report. Department of Computer Sciences, Purdue University, 1982.

[Kung80] H. T. Kung, C. E. Leiserson, "Algorithms for VLSI Processor Arrays", C. Mead and L. Conway, Introduction to VLSI Systems, Adison-Wesley, Reading, Ma., (1980). pp. 271-292.

[Kuhn80] R. Kuhn, Ph. D. Thesis, Dept. of Computer Science, University of Illinios, Urbana, Illinios, 1980.

COMBINING PARTIAL RESULTS IN AN MIMD COMPUTER

Harry F. Jordan
Department of Electrical Engineering
University of Colorado
Boulder, Colorado, 80309

Abstract

One of the most demanding types of computation in an MIMD computer is one in which all instruction streams are tightly coupled in producing a single result. This paper treats this problem with respect to a shared memory multiprocessor. Experimental verification of the analysis is obtained on the HEP computer, a pipelined multiprocessor. The specific problem analysed is directly comparable to a previous analysis of the same problem on a network computer. A comparison suggests that there is a strong correspondence between delays due to conflicting access to a shared memory cell in the current case and the conflict for use of communication links in the network computer case.

Introduction

In any MIMD computer an important type of computation is one in which a large number of processes contribute to a single result. The demands made by such a cooperative computation on the data communications and synchronization facilities of a parallel architecture are quite stringent if good performance is to be achieved. The author has previously been involved in a study of cooperative computation on a computer consisting of a large number of microprocessors sharing no memory but having several types of high performance communication structures [1]. The current paper deals with a true multiprocessor in which all data memory is shared. The experimental results presented are from the HEP computer [2,3,6], a pipelined, shared resource, MIMD machine.

Summation of Partial Results

In a multiple instruction stream computer numerical algorithms are carried out by multiple parallel instruction streams, or processes, which share data. A typical short term behavior is that N processes run completely independently through a computation P(i); i = 1,2,...,N after which some partial results V(i) must be combined across all processes. A typical form of combination is summation, treated in the discussion which follows. The discussion applies, however, to reduction (in the APL sense [4]) over any commutative and associative dyadic operator.

If we characterize the processes performing computations P(i) as producers of the partial results V(i) and identify a consumer process which uses the sum result R then two methods of performing the summation can be identified. The first method we call consumer driven because the consumer process executes instructions which actually perform the individual additions. We will use a dollar sign $ preceding a variable name to denote the combination of a value with a full/empty status as is done in the HEP multiple instruction stream computer [3]. Further we assume a hardware mechanism, as in HEP, which will delay the reading of such a variable until it is full and delay writing it until it is empty. Reading $V will set its status to empty while writing it will set the status to full.

A producer driven summation which runs efficiently regardless of the order in which partial results are produced makes use of a single communications location to pass partial results to the consumer process. The hardware synchronization mechanisms prevent a producer from storing its result until a previously stored value has been consumed. The programs for producers and consumers then appear as:

Producer

```
$V := partial result j ;
```

Consumer

```
s := $V ;
for K := 2 step 1 until N do
        s := s + $V ;
$R := s ;
```

Realizing that producers are essentially idle while waiting for the consumer to empty $V leads to the second method of summation which we call producer driven because the producers perform the actual additions. In this method the partial sum (s in the above programs) is shared by the N producers. This shared partial sum $S is initialized to zero and a count location $Count is initialized to N. The producer program is then:

Producer j

```
C := $Count - 1;
s := $S + V(j);
if C ≠ 0 then begin $S := s; $Count := C end
        else $R := s;
```

Most of the code executed by Producer j is concerned with counting the number of producers which have contributed to the sum and determining the last one, and hence completion of the result. This counting was done by the loop in the consumer driven method. If completion could be determined by some other means, then a producer could execute only: $S := $S + V_j. With the completion count the consumer need only use the result $R when it becomes full. It should be noted that the section of code from the use of $Count in line one to its filling in line three

0190-3918/82/0000/0287$00.75 © 1982 IEEE

forms a critical section which at most one process j can execute at a time. Since $S appears only in this critical section it need not have a full/empty status.

Data Conflict/Synchronization Time Analysis

The time required to complete such a summation is determined by two influences: the times required by the subcomputations P_j, which we will call $t(P_j)$, and the times spent by producers waiting to execute the critical section on $Count. The time required for completion of the result t($R) is certainty no less than

$$\{t(P_j)\ j = 1,2,\ldots,N\}$$

and if $t(P_j)$ has a variance which is much larger than the time spent by one producer in the critical section, t_c, then we expect critical section competition to have a small effect. The critical section will have its maximum influence when all $t(P_j) = t_p$ are equal. In this case the time required to produce the result will be $t(\$R) = t_p + N \cdot t_c$.

In general, with fairly tightly synchronized processes, we expect to be able to say that $t(P_j)$ is randomly distributed with a mean which is much larger than t_c and a variance which is larger than t_c but not larger than $N \cdot t_c$. In this case the order independence of the summation algorithm is useful but critical section conflict also influences the computation time. No matter what the variances of $t(P_j)$ this case will occur for N sufficiently large, assuming that the variance of $t(P_j)$ does not increase with N. Since the time delay due to critical section conflict results from the sharing of $Count by up to N-1 other processes, it can be reduced by employing a form of batch adding in which the N processes are divided into groups of G processes so that each group forms an independent sum. These partial sums are then combined G at a time in the manner of a base G tree until a single sum results.

A recursive procedure Sum for group summation is organized as follows. Assume that $N = G^K$ terms are to be summed in groups of size G and that for each level ℓ of the summation tree, there are $G^{K-\ell}$ sum variables and corresponding full/empty count variables. A recursive procedure to sum a term u into sum number sn at level ℓ of the tree involves each producer calling the procedure at level one. The last producer to contribute to a group sum carries that sum to the next level of the tree, adding it to the group sum at that level. One of the producers, the "last" one, will call Sum once for each level $\ell = 1,2,\ldots,K$ and terminate after filling the result $R. To adapt this procedure to arbitrary N it is only necessary to set

$$K = \text{ceiling } (\log_G N)$$

and to alter the initial values of the count variables.

A detailed analysis [7] of the time to complete a summation yields an upper bound of the form: t($R) $\leq$ time for largest partial result

+ time for initial entry to Sum

+ K * time for single level

+ K $\times$ (G - 1) $\times$ critical section time.

Assuming N to be fixed, it is of interest to determine an optimal group size G. Only the last two terms of the bound depend on G and minimizing the sum of these terms over G leads to a value of G satisfying $G(\ln G - 1) = r - 1$ where r is the ratio of the time for a single level to the critical section time. Since the critical section is contained within the code for a level it is clear that $r \geq 1$. The optimal value of G for several values of the ratio is shown in Table 1.

r	1	2	3	4	5
G	2.718	3.591	4.319	4.971	5.572

Table 1: Optimum Group Size G

Experimental Results

The group summation method discussed above was programmed for the HEP multiple instruction stream computer using the HEP Parallel FORTRAN language [5]; a set of 50 producer processes were created which, for simplicity, summed their process indices j ; j = 1,2,...,50. Thus all $t(P_j) = 0$ and the worst case conflict situation obtained. A main program started the 50 producers synchronously and waited for the sum to become available, timing the length of the wait. The results for N = 50 and several values of G are shown in Table 2.

Group Size G	Time to Sum microseconds
2	759.7
3	707.0
4	739.4
5	793.9
6	845.2
7	937.0
8	941.7
9	980.7
10	1066.4
25	2004.1
50	3616.4

Table 2: HEP Parallel FORTRAN Group Summation

The running times show a behavior which is consistent with a ratio r between one and two. An assembly code listing of the sum procedure yielded a ratio of 1.233 by actual instruction count. One way to look at the results is that in the best case, G = 3, an addition is being done every 14.1 microseconds. This point of view is misleading for three reasons. First, the procedure is not meant to compete with a

summation done by a single process but is used to combine results produced by independent processes set up in parallel for other purposes. Second, the HEP computer cannot run 50 processes at full speed. In its current prototype version it will execute one instruction from each process every 5 microseconds when all processes are active. Finally, the current (1980) HEP prototype has a severe indexing restriction which causes indexed expressions in FORTRAN to produce unreasonable numbers of machine instructions. To measure the latter effect, the indices for a group size of G = 3 were precomputed prior to execution time and a summation time of 233.3 microseconds was obtained. This respresents a speedup by a factor of 3 over the execution time indexing version and compares favorably with the time to sum 50 integers with a single process of 243.2 microseconds.

It should be noted that memory bank conflict is not an issue in HEP since memory accesses are pipelined [3] and a separate analysis [6] shows that 50 processes are more than sufficient to keep pipeline fall-through time from having any influence on computation time. Thus the above analysis, which relates only to the shared memory cells and not to larger blocks of memory, is the correct one in this case.

Communication Delay Versus Access Conflict

It is interesting to compare the qualitative aspects of the current analysis with those of the previous analysis of the Finite Element Machine (FEM) multi-microprocessor network [1]. In the FEM a time multiplexed bus connects all processors and a set of parallel communications paths connect processors with their eight nearest neighbors in a planar square array. The group summation considered in the present paper corresponds most closely to the "distributed computation" considered there. To make use of the parallel neighbor communication paths a group size of nine, corresponding to a processor and its eight nearest neighbors, was chosen for the lowest level. The group size at subsequent levels was two, corresponding to a binary tree.

The "distributed computation" had a fairly complex control structure but was still faster than the other algorithms studied in spite of this control overhead. The speed resulted from the use of non-shared parallel communications paths for most of the information transfer. This corresponds very closely to the use of independent group sum locations in the current analysis to limit the number of processes competing for the same resource (memory cell). The group size in the FEM case could not be varied reasonably since it depended on network structure. One of the other algorithms examined, however, the centralized algorithm, corresponds very closely to the use of a single group in the current study. The poor results obtained for that algorithm correspond well to those obtained for the single group of size 50 reported above.

There are enough qualitative similarities in the two analyses to indicate that communications link conflict in a network computer plays a role in performance analysis which is quite analogous to shared memory conflict in a multiprocessor. In fact, the HEP system with its pipelined memory access and full empty memory cells can be analyzed quite accurately by taking any cell shared between two processes as a one word buffered communications link between those processes.

References

[1] Jordan, H. F., Scalabrin, M. and Calvert, W., "A Comparison of Three Types of Multiprocessor Algorithms," Proc. 1979 International Conference on Parallel Processing, (August 1979), pp. 231-238.

[2] Smith, B. J., "Architecture and Applications of the HEP Multiprocessor Computer System," Real Time Signal Processing VI, Proceedings of SPIE, Vol. 298 (August 1981).

[3] Smith, B. J., "A Pipelined Shared Resource MIMD Computer," Proc. 1978 International Conference on Parallel Processing, (August 1978), pp. 6-8.

[4] Iverson, K. E., A Programming Language, John Wiley and Sons, New York (1962).

[5] Denelcor, Inc., "HEP Parallel Fortran Users Manual," Denelcor Publication 10002-00, 3115 E. 40th Avenue, Denver, Colorado, 80205.

[6] Jordan, H. F., "Performance Measurements on HEP - A Pipelined MIMD Computer," Report CSDG 81-5, Computer Systems Design Group, Electrical Engineering Department, University of Colorado, Boulder, Colorado.

[7] Jordan, H. F., "Combining Partial Results in an MIMD Computer," Report CSDG 82-1, Computer Systems Design Group, Electrical Engineering Department, University of Colorado, Boulder, Colorado.

AN APPROXIMATE ANALYTICAL MODEL FOR ASYNCHRONOUS PROCESSES IN MULTIPROCESSORS[†]

Michel Dubois
Thomson-CSF
Laboratoire Central De Recherches
Domaine De Corbeville, B.P.No. 10
91401 Orsay, FRANCE

and

Fayé A. Briggs
Department of Electrical Engineering
Rice University
Houston, TEXAS 77001

Abstract

Multitasked asynchronous processes on multiprocessors are subject to performance degradations due to the sharing of critical sections. The concurrent accessing of such critical sections also results in the familiar lock-out problem. A general methodology to estimate the performance degradations of such algorithms on the processor utilization is presented in this paper. We study in detail a simple multiprocessor system with P processes sharing one critical section. We then generalize our study to a system with an arbitrary number of critical sections. The approximation is good for the case in which the critical sections have low coefficients of variation. Such an analysis, when applied to the processor lockout problem, can result in an optimization of the distribution of the critical sections in a multiprocessor operating system.

1. Introduction

In order to guarantee the correctness of execution of multitasked multiprocessor algorithms, explicit synchronization is often required in parallel algorithms. The resulting blocked time is large if the synchronizing processes have significantly different processing times. The performance of synchronized iterative parallel algorithms in multiprocessors has been studied in [DUB82]. In some cases, the synchronization points may be removed. A synchronized algorithm in which all explicit synchronization conditions have been suppressed becomes an asynchronous algorithm. The concept of asynchronous algorithms is derived from the chaotic relaxation scheme investigated by Chazan and Miranker [CHA69]. Baudet has determined general convergence conditions for an asynchronous iterative algorithm [BAU78]. Kung defined the properties of an asynchronous algorithm and described several examples [KUN76]. An asynchronous algorithm is controlled through a set of global variables accessible to all processes. Each process computes independently (processing phase), reads the global variables, modifies some of them, then activates a new processing phase or terminates. Global variables are usually accessed in critical sections in order to ensure correctness.

There are also many situations in which a processor that tries to access a critical section (C.S.), such as a ready list, is blocked because the C.S. is being used by another processor. In this case the processor may spin until the lock is released. Therefore, the processor which attempts unsuccessfully to access the C.S. is locked out. The lockout problem is a direct result of multiple processors attempting to process common data structures asynchronously. This situation resembles the memory conflict problem in tightly-coupled multiprocessor systems discussed by so many authors [CHA77]. For the memory conflict problem, the resources were hardware resources (memory modules), whereas, the lockout is due to contention for software resources. There are numerous such shared data bases in a multiprocessor operating system besides the ready list. These include memory allocation table, page allocation table and I/O lists.

In order to evaluate the efficiency of various configurations of lockable software resources, we must consider the effect of processor lockout. The most significant potential cost arises because a process that blocks on a spin lock does not relinquish the processor on which it is executing. Thus if a process blocks on a lock for a lengthy period, an important system resource, a processor, will be lost to the system for the duration of this period.

Lengthy blocking arises when contention for a lock becomes too high. To keep contention at an acceptable level, locks must be used to provide mutual exclusion only when the grain size is sufficiently small [JON79]. Grain size is determined by two factors: the first factor is the amount of time for which mutual exclusion is necessary. A short critical section has a smaller grain size than a long one. The second factor is how frequently mutual exclusion is needed. A lock that must be locked often has a longer grain size than a lock that is touched infrequently. Locks are basically associated with pieces of code or data structures. As the number of processors and processes in the system increases, the grain size of such locks tends to grow because they are inevitably accessed more and more frequently.

In this paper, we introduce an analytical model based on the central server model to evaluate the performance of asynchronous processes and the effect of software lockout upon system performance. One classical approach to solving the central server model is to apply the BCMP model [BAS75] for closed queueing networks. In [KUM79], an aggregation approximation has been applied to this model for exponentially distributed critical sections. However, critical sections tend to behave more like deterministic servers, in which case the BCMP model is not very effective. In the following, we present a simple approximation to solve the central server model and to estimate the processor utilization due to the execution of asynchronous processes. The

[†]This research was supported by NSF Grant ECS 80-16580.

0190-3918/82/0000/0290$00.75 © 1982 IEEE

precision of the approximation is good only for critical sections with low coefficients of variation. This approximation is similar to the one used by Hoogendoorn to study the performance of multiprocessor memories [HOO77]. The model is introduced in its general form.

An implementation of an algorithm on a given architecture is characterized by a set of performance features, $\{f_1, f_2, \ldots, f_N\}$, extracted from the analytical model. Let F be the feature space for the given architecture and algorithm. F can be seen as the product space of the one-dimensional spaces generated by each feature:

$$F = \{f_1\} \times \{f_2\} \times \ldots \times \{f_N\}.$$

The topology of the space F is complex. The feature values may be dense along some coordinate axes, and discrete along some others. A performance index for a given architecture is a real function defined on F by the analytical model. Local maxima of the index locate operating points in F where the architecture and the algorithm implementation are particularly "well-matched" with respect to the index. The power of analytical models resides in the estimation of the impact on the performance of a given feature or subset of features in isolation. The average processor utilization, U, defined as the fraction of time a processor is busy, is used in this paper as the performance index. The feature-space approach permits the visualization of the effect of the performance features on the parameters of the algorithm and architecture.

2. Central Server Model for Asynchronous Processes

A simple multiprocessor architecture is shown in Fig. 1. A set of P independent processors execute tasks in a common shared memory through an interconnection network. This architecture is called "tightly-coupled" and is typified by the C.mmp [WUL81]. Such a multiprocessor system can implement multitasking in which a given algorithm is decomposed into a set of tasks that run independently in parallel [FLY72]. When these task modules communicate intensively, they are each associated with a processor, under a group scheduling strategy [JON79], i.e., the processes are swapped in and out simultaneously, and not individually. A process is not preempted when it is blocked at the beginning of a critical section; rather it "spins" (busy wait) [JON79] or waits for an interrupt without relinquishing the processor (in the second case, user hardware interrupts must be provided or else an operating system call is made). These strategies are designed to minimize the overhead and speed up the algorithm in an environment where the cost underutilizing processors is secondary.

One problem which occurs in multiprocessor systems is memory contention [CHA77]. Generally, the memory is made of a set of independent modules. It is interleaved. The instruction cycle of each processor comprises a variable number of machine cycles such that at most one memory reference occurs during a machine cycle. A rejected request is resubmitted at the next machine cycle. Under these conditions, a request to the shared memory can be characterized, in most cases, by a probability of acceptance P_a, resulting in a geometrically distributed access time [PAT81].

In the architecture of Fig. 1, the processors compete for the shared memory on a word-by-word basis. This results in performance degradations due to conflicts in accessing instructions and data [CHA77]. Let P be the number of processors and M the number of memory modules. Each processor references the shared memory with a probability r during any machine cycle. A widely used approximation, which is justified by the interleaved storage pattern, is that the references to the memory are independent and uniformly distributed among the M memory modules. This approximation leads to the probability of acceptance of a memory request as

$$P_a = \frac{M}{rP}\left[1 - \left(1 - \frac{r}{M}\right)^P\right]. \qquad (1)$$

For a derivation of Equation (1), see, for example, [PAT81].

Formula (1) was derived under the hypothesis of independent requests. In reality, however, a rejected request is automatically resubmitted during the next machine cycle. One correction was introduced in [DUB81] to take into account the wasted cycles due to memory conflicts in the computation of P_a. The behavior of any one process is described in the Markov graph of Fig. 2. W is the state corresponding to a wasted cycle due to memory conflicts, and A is an active cycle, during which a processor may issue a new request. Solving for (q_A, q_W), the stationary probability distribution of the states, one finds

$$q_A = \frac{P_a}{P_a + r(1-P_a)}\ ;\ q_W = 1 - q_A\ . \qquad (2)$$

From the graph of Fig. 2, r is defined more precisely as the probability of referencing the memory during an active machine cycle. In the absence of memory conflicts, all machine cycles are active. Because of the memory conflicts, memory references are also made during each wasted cycle. The effective rate of memory access cycles is thus

$$r_e = rq_A + q_W = \frac{r}{r + P_a(1-r)} \qquad (3)$$

and Equation (1) becomes

$$P_a = \frac{M}{r_e P}\left[1 - \left(1 - \frac{r_e}{M}\right)^P\right] \qquad (4)$$

Equations (3) and (4) define an iterative process by which one can compute P_a, for given M, P and r.

In a typical asynchronous MIMD algorithm, P processes share L critical sections. Outside of a critical section, a process can proceed freely. However, only one process can be executing a given critical section at any given time. Typically, the execution of critical sections con-

sists of updating one or more common variables. The fluctuations of their execution times, which are mainly due to memory contentions, are often small. Data-dependent fluctuations can also be present (e.g., conditional modification of a shared variable).

A process in an asynchronous MIMD algorithm can be seen as a succession of cycles. A cycle consists of two phases in which a process is in the non-critical-section phase or in the critical-section phase. More specifically, a cycle consists of some processing followed by a request for a critical section, a possible waiting time to obtain the right of access, and the execution of the critical section. This behavior can be modeled by a closed queueing network with a population of size P, a P-server node (PN) and L single-server queues, as shown in Fig. 3. Each server queue is for a critical section (CS).

In the following sections, an approximation for this closed queueing network is presented. We begin with the simple case in which L = 1, then generalize it to an arbitrary value of L. Simulations have shown that the model is adequate for a wide range of systems and for CS's with low coefficients of variation (say, less than .5). The coefficient of variation of the independent processing time (CVT) has little influence on the model prediction. When it is increased beyond 1, the approximation deteriorates very slowly. Finally, the model is not appropriate for the case in which CVT = 0, and the CS's and routing are deterministic. These figures are given to indicate the domain of validity of the model.

The model shown in Fig. 3 is similar to the model for time-sharing systems with L computing centers and P users [KLE76]. The terminology used in the approximation is borrowed from such systems. The processes are called "jobs." The independent processing time is the "think time," and its mean is noted by T. The critical sections are referred to as "servers," and the mean execution time of each server is S (when L = 1) or S_i (when L > 1). Further notations will be introduced in the following discussion.

3. An Approximation to the G/G/1//P Queue

Two queues of the G/G/1//P class, namely the M/G/1//P queues [JAI68] and the D/D/1//P queues [KIN78], have exact solutions. Based on this class of queues, we hereby propose a simple approximation to an algorithm with one critical section (Fig. 4). There are P jobs, one processor node (PN) with P servers and one single-server queue which represents the critical section (CS). Under the stochastic assumption, each job has a probability X of being outside of the CS (or, equivalently, of being in the PN). In such a state, the job can request access to the CS. By the ergodic property, X is also the average fraction of "think time" [KLE76] within each job cycle. From the value of X, one can derive the mean properties of the network. For instance, the mean job cycle time, denoted by C, is related to X by the formula $X = \frac{T}{C}$. Let

$I(t) = (i_1(t), i_2(t), \ldots, i_P(t))$,

with $i_j(t) = 1$ iff job j is not in the server, and $i_j(t) = 0$ iff job j is in the server at time t.

I(t) is called the indicator vector for the CS at time t. Each of its components indicates whether a given job is present in the server or not.

Theorem 1. For any G/G/1//P queue in equilibrium,

$$E\left[i_1 i_2 \ldots i_P\right] + \rho X = 1, \tag{5}$$

where $E\left[i_1 i_2 \ldots i_P\right]$ is the expected value of the product of the components of the indicator vector and $\rho = Pu$, with $u = \frac{S}{T}$.

Proof: Let X_s be the probability that the server is busy. Equating the flows of jobs in and out of the server at equilibrium, we obtain

$$X_s \frac{1}{S} = P \frac{1}{C} = P \cdot \frac{X}{T}, \quad \text{or } X_s = \rho X. \tag{6}$$

On the other hand,

$$\begin{aligned} X_s &= \text{Prob}\left[\text{"at least one job is in the server"}\right] \\ &= 1 - \text{Prob}\left[\text{"all the jobs are not in CS"}\right] \\ &= 1 - \text{Prob}\left[\text{"}i_1 i_2 \ldots i_P = 1\text{"}\right] \\ &= 1 - E\left[i_1 i_1 \ldots i_P\right]. \end{aligned} \tag{7}$$

The last equality results from the fact that the expected value of a random variable taking only the values 0 and 1 is equal to the probability of the variable being 1.

The theorem results from equating (6) and (7). []

Corollary 1.1. (Approximate Model)

If $i_1, i_2, \ldots, i_P$ are independent random variables, then

$$E\left[i_1 i_2 \ldots i_P\right] = E\left[i_i\right] \cdot E\left[i_2\right] \ldots E\left[i_P\right] = X^P$$

$$\text{and } X^P + \rho X = 1. \tag{8}$$

Properties of the Approximate Model

Below, we give three properties of the approximate model. These properties will be proven in a later section for a more general case.

P1. Equation (8) always has a unique solution X_a between 0 and $\text{MIN}(1, \frac{1}{\rho})$.

P2. $X_a = \frac{1}{1+u} + O(\rho^2)$.

For a constant P, X_a "behaves" as the function $\frac{1}{1+u}$ when u tends to 0.

The approximation X_a obtained from equation (8) is better when ρ is small. In this case, the waiting time is small and a job cycles as if it were the only one present in the network. The independence assumption is thus valid.

P3. For a constant u, $\lim_{P\to\infty} \rho X_a = 1$.

In [KLE76], these properties are shown to hold in the general case, i.e.,

$$X \simeq \frac{1}{1+u}, \text{ for } P << 1 + \frac{1}{u}$$

and $$X \simeq \frac{1}{Pu}, \text{ for } P >> 1 + \frac{1}{u}.$$

A consequence of property P3 is that the approximation of equation (8) is still good when the hypothesis leading to equation (8) (no interference between jobs) is most violated.

Unfortunately, X_a cannot be a bound for all systems. It is very easy to prove that the D/D/1//P queue [KIN78] is such that $X > X_a$ for all P and u. On the other hand, it is not difficult to find examples of M/M/1//P queues with $X < X_a$.

4. Discussion and Heuristics

Evaluating $E\left[i_1 \ldots i_P\right]$ is analytically impossible for most cases. Faced with such a complexity, we resort to extensive simulations. One interesting theoretical result that should guide us, however, is given in [PRI76], where it was shown that, among all M/G/1//P systems, the M/D/1//P has the largest value of X (and thus the best performance). Price also showed that when the coefficient of variation of the server (CVS) becomes large, the performance of the M/G/1//P queue depends very much on higher moments of the server's distribution.

For values of CVT and of CVS less than .5, the hypothesis of the model is violated because the job flow is practically deterministic [KIN78], and the interactions between the jobs in the network are very large. The approximation performs best for a short deterministic service time. Indeed, large instances of the service time are more likely to result instantaneously in longer queues and in more interactions between jobs. Some simulation results are summarized in Tables 1 and 2. In all cases, an offset exponential and a hyperexponential were used for the cases of a coefficient of variation less than one and greater than one, respectively. The model parameters have been selected such that (P-1) u = 1. This case is one of the most difficult to estimate, since it is an intermediate point for which the results of properties P2 and P3 do not apply [KLE76 pp. 208-209]. The relative error in X is less than 5% in most cases (the errors larger than 5% are underlined). The approximation worsens slightly when the number of processors and the CVT increase. It is not adequate for the cases when both distributions are either exponential or deterministic. In such cases, we can use the M/M/1//P queueing network [JAI68] or D/D/1//P queueing network [KIN78].

5. Extension to Multiple Critical Sections

We now consider the more general network, shown in Fig. 3. In this network, a job stays in the PN for a random "think time" and then branches to any one critical section, CS_i, i = 1,...,L with a branching probability p_i. The mean processing time of CS_i is S_i. Let $I_k(t)$ be the indicator vector for the k-th critical section.

$$I_k(t) = (i_{k,1}(t), i_{k,2}(t), \ldots, i_{k,P}(t)),$$
$$\text{for } k = 1,\ldots,L.$$

The definition of each component is the same as in section 3. Each component $i_{k,j}(t)$ of the vector indicates whether the job from processor j is in the k-th critical section or not, at time t.

Theorem 2. For each critical section,

$$E[i_{k,1} i_{k,2} \ldots i_{k,P}] + \rho_k \cdot X = 1, \qquad (9)$$

where $\rho_k = u_k P$ and $u_k = \frac{p_k S_k}{T}$.

Again, X is the fraction of time spent in the PN.

Proof: The proof proceeds as for theorem 1 and is omitted here. []

Theorem 3. Within the framework of the job independence hypothesis leading to equation (4), an approximate solution for the model of Fig. 2 is

$$X + (L-1) = \sum_{k=1}^{L} (1 - \rho_k \cdot X)^{1/P}. \qquad (10)$$

Proof: Given the independence hypothesis, equation (9) becomes

$$X_k^P + \rho_k \cdot X = 1, \qquad (11)$$

where X_k is the fraction of time spent by each job outside CS_k.

On the other hand,

$$\text{Prob("job is in PN")} = 1 - \sum_{k=1}^{L} \text{Prob("job is in } CS_k\text{")},$$

$$\text{or} \qquad X = 1 - \sum_{k=1}^{L} (1 - X_k). \qquad (12)$$

Combine equations (11) and (12) to obtain (10). []

Formula (10) is the approximate model. Note that at the solution, we must have

$$1 > \rho_k X, \quad k=1,\ldots,L. \qquad (13)$$

Equation (10) is obviously a generalization of

(8).

The existence of a solution to equations (8) and (10) is established below.

Theorem 4 Equation (10) has a unique real solution, X_a, such that

$0 \le X_a \le \text{MIN}(1, \frac{1}{\rho_{MAX}})$, where $\rho_{Max} = \max_k \{\rho_k\}$, $k=1,\ldots,L$.

Proof: We give a graphical proof. We initially assume that $\rho_{Max} < 1$. When X increases from 0 to 1, the L.H.S. of equation (10)) increases monotonically from (L-1) to L, whereas the R.H.S. decreases monotonically from L to $\sum_{k=1}^{L} (1-\rho_k)^{1/P} < L$. There must be an intersection point for a value of X between 0 and 1.

The alternative is $\rho_{Max} \ge 1$. In this case, when X increases from 0 to $\frac{1}{\rho_{Max}}$, the L.H.S. of equation (10) increases monotonically from (L-1) to $\frac{1}{\rho_{Max}} + L-1$, while the R.H.S. decreases monotonically from L to

$$\sum_{k=1}^{L} (1-\rho_k/\rho_{Max})^{1/P} < (L-1).$$

Again, an intersection point must exist for X between 0 and $\frac{1}{\rho_{Max}}$. []

Now that the existence of a unique solution is proved, one can find it by iterative methods or by graphical methods.

Let $u_{Max} = \max_i \{u_i\}$ and $\rho_{Max} = Pu_{Max}$. The following two theorems illustrate the asymptotic behavior of X_a.

Theorem 5.

$$X_a = \frac{1}{1 + \sum_{i=1}^{L} u_i} + O(\rho_{Max}^2). \qquad (14)$$

Because $\rho_i X < 1$, $i=1,\ldots,L$, a first order approximation of equation (10) yields

$$X + (L-1) = L - \sum_{i=1}^{L} Xu_i + O(\rho_{Max}^2),$$

from which the claim can be easily derived.

For a constant P, the first term of equation (14) dominates when u_{Max} tends to 0. This first term is also the value of X obtained by neglecting the waiting time at each queue [KLE76].

Theorem 6. For constant u_i, $i=1,\ldots,L$,

$$\lim_{P\to\infty} \rho_{Max} \cdot X = 1. \qquad (15)$$

Proof: As $X+L-1 \ge L(1 - \rho_{Max} X)^{1/P}$ and $\rho_{Max} \cdot X \le 1$, we have, for all P,

$$1 \ge \rho_{Max} X \ge 1 - (\frac{X+L-1}{L})^P.$$

If $S_i \neq 0$, then $X < 1$, and $\frac{X+L-1}{L} < 1$, so that the claim is proved. []

Note that $X = \frac{1}{\rho_{Max}}$ is the asymptotic value obtained when one server becomes a bottleneck [KLE76]. Theorems 5 and 6 show that the approximation of equation (10) is correct for asymptotic cases. It can be expected to be a good approximation for intermediate values of the parameters; i.e., values such that

$$P \simeq \frac{1 + \sum_{i=1}^{L} u_i}{u_{Max}} \text{ [KLE76 pp. 220-221]}. \qquad (16)$$

The results for a uniform and a triangular branching probability distribution are shown in Table 3. The approximate model is in agreement with the simulations performed for various possible values of CVS_i and CVT. Note that the approximation is good for the case when CVS_i=CVT=0 because of the random routing, which destroys the correlation between jobs, exhibited, for example, in the D/D/1//P system. The maximum possible value for X is .5, because $S_k = T$.

6. The Processor Utilization

The average processor utilization, denoted U, is used to evaluate the degree of matching between an architecture and an asynchronous algorithm, as modeled in sections 3 through 5. Besides asynchronous algorithms, the model can also be applied to evaluate processor lockout in the context of the multiprocessor operating system. In both cases, a processor is busy while it is outside and inside of the critical sections. Idleness is caused by blocking at the entrance to the critical sections and by memory conflicts.

For the G/G/1//P case, the processor utilization, U can be found as follows. The total time during which a processor is busy within each cycle is

$$m_o = q_A (T + S), \qquad (17)$$

where T and S are the mean "think" time and the mean service time respectively. q_A is the coefficient which accounts for memory conflicts. In general, a fraction r of machine cycles contains a reference to the memory, and each memory request is accepted with a probability P_a. If the request is not accepted, it is resubmitted during the next machine cycle. Under these conditions, we have shown that (see equation (2))

$$q_A = \frac{1}{(1-r) + r\frac{1}{P_a}} = \frac{P_a}{(1-r)P_a + r}. \qquad (18)$$

where P_a is given by equations (3) and (4) if we assume spin locks. On the other hand, the time taken by each cycle is $m_I = C$. The average processor utilization is thus

$$U = \frac{(T+S)\cdot q_A}{C} = q_A \cdot \frac{T+S}{T} \cdot X = q_A \cdot X \cdot (1 + \frac{S}{T}) \quad (19)$$

$$= q_A \cdot X \cdot (1 + \frac{S_o}{T_o}) ,$$

where $S_o (= q_A \cdot S)$ and $T_o (= q_A \cdot T)$ are the mean time to execute CS and the mean think time in the absence of memory conflicts, respectively.

This formula can be generalized to the case of L critical sections, provided that X is defined as the fraction of time spent in the processor node per cycle through the network, and S is replaced by $\sum_{i=1}^{L} p_i S_i$. In equation (19), $(1 + \frac{S_o}{T_o})X$ represents the penalty due to the synchronization at the critical section.

The relative error in the estimation of X is matched by a similar added error in the estimation of U. Whereas the estimation of X from the model is reliable, the estimation of R, the mean response time, from X can introduce an unacceptable error in R. To show this, we recall that $X = \frac{T}{T+R}$, where T is the mean think time and R is the mean response time. Let ϵ_X and ϵ_R be the relative errors in X and R, respectively. Then

$$\epsilon_X = \frac{\Delta X}{X} \triangleq \frac{dX}{dR} \cdot \Delta R \cdot \frac{1}{X} \triangleq \frac{R}{T+R} \cdot \frac{\Delta R}{R} \triangleq (1-X) \cdot \frac{\Delta R}{R} .$$

Hence, $\epsilon_R \triangleq \frac{\epsilon_X}{1-X}$.

It can be seen that if $\epsilon_X \neq 0$, then ϵ_R can be very large as X tends to 1.

However, it can be easily seen that the relative error in C is equal to that in X, because $X = \frac{T}{C}$. Therefore, the cycle time can be estimated with the same relative error as that in X. However, even for a small error in X, the error in R can be large.

As an example, we analyze a system with P independent and identically dependent processes sharing L identical critical sections. We denote the ratio $\frac{S_o}{T_o}$ by ξ_o. By equation (19),

$$U = q_A \, X \, (1 + \xi_o) .$$

X is the solution of equation (10):

$$X = \frac{L}{P\xi_0} \left[1 - \left(\frac{X+L-1}{L}\right)^P\right] .$$

q_A is obtained as $q_A = \frac{P_a}{r + P_a(1-r)}$, with

$$P_a = \frac{M}{r_e P} \left[1 - \left(1 - \frac{r_e}{M}\right)^P\right] , \text{ and } r_e = \frac{r}{r + P_a(1-r)} .$$

Recall that M is the number of memory modules and is considered fixed. P is the number of processors participating in the algorithm execution, and its maximum value is M. The performance features are P, r, L, ξ_o. These features generate a 4-dimensional space of which two plane cuts are displayed in Fig. 5. The cuts illustrate the combined effects of the contention for the critical sections and the memory modules when P = 64, M = 64, and L = 16 or 64. As L goes from 16 to 64, ξ_o (the critical-section to think-time ratio in the absence of memory conflicts) becomes the dominant feature over r (the probability of a memory reference per active machine cycle) in the typical operating region (r greater than .6).

7 Conclusion

A simple approximation to estimate the processor utilization in asynchronous MIMD algorithms has been presented in this paper. The model assumes that the time taken by the execution of a critical section is deterministic or has a low coefficient of variation.

The validation of an approximation with such a broad applicability requires extensive simulations. Only selected results have been reported here. The approximation has been compared to the simulation results for the G/G/1//P (Tables 1 and 2) and the more general system of Fig. 3 (Table 3). Note that the model does not include the service discipline at the queues. However, the simulations were run for a FCFS (First-Come-First-Served) policy. To appreciate the quality of the approximation from Tables 1 and 2, one should keep in mind that the most difficult cases to estimate are the ones corresponding to intermediate values of the parameters, as defined by equation (16).

Such simple approximate models have great importance in the understanding of multiprocessor program behavior. They permit software designers to compare alternative implementations and to estimate to which degree a given algorithm is fit to be executed on a tightly coupled system. For example, one interesting property of the model is that it depends only on the number of processors, critical sections and their total traffic ρ_i. This implies that, within the limits of validity of the model, the speed-up is equally affected by the wait on short but frequent critical sections and the wait on long but infrequent critical sections provided that the total traffic is the same. However, short critical sections require more additional instructions to open and close the critical sections.

LIST OF REFERENCES

[BAS75] F. S. Baskett, K. M. Chandy, R. R. Muntz, and F. G. Palacios, "Open, Closed, and Mixed Networks of Queues with Different Classes of Customers," Journal of the ACM, Vol. 22, No. 2, April 1975, pp. 248-260.

[BAU78] G. M. Baudet, "The Design and Analysis of Algorithms for Asynchronous Multiprocessors," Ph.D.Thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, April 1978.

[CHA69] D. Chazan and W. Miranker, "Chaotic Relaxation," Linear Algebra and Its Applications, Vol. 2, 1969.
[CHA77] D.Y. Chang and D.J. Kuck, "On the Effective Bandwidth of Parallel Memories," IEEE Transaction on Computers, May 1977, pp. 480-489.
[DUB81] M. Dubois and F. A. Briggs, "Efficient Interprocessor Communication for MIMD Multiprocessor Systems," Proceedings of the 8th International Symposium on Computer Architecture, May 1981.
[DUB82] M. Dubois and F. A. Briggs, "Performance of Synchronized Iterative Processes in Multiprocessor Systems," to appear in the IEEE Transactions on Software Engineering.
[FLY72] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," IEEE Transactions on Computers, Vol. C-21, No. 9, September 1972, pp. 998-1005.
[HOO77] C. H. Hoogendoorn, "A General Model for Memory Interference in Multiprocessors," IEEE Transactions on Computers, Vol. C-26, No. 10, October 1977, pp. 998-1005.
[JAI68] N. K. Jaiswal, Priority Queues, New York: Academic Press, 1968.
[JON79] A. K. Jones and P. Schwartz, "Experience Using Multiprocessor Systems. A Status Report," Carnegie-Mellon University, Technical Report, CMU-CS-79-146, October 1979.
[KIN78] L.L. L. Kiney, and R. G. Arnold, "Analysis of a Multiprocessor System with a Shared Bus," Proceedings of the 5th Annual Symposium on Computer Architecture, April 1978.
[KLE76] L. Kleinrock, Queueing Systems, Vols. I and II, John Wiley and Sons, 1976.
[KUM79] B. Kumar and T. A. Gonsalves, "Modeling and Analysis of Distributed Software Systems," Proceedings of the 7th Asilomar Conference on Operating System Principles, December 1979.
[KUN76] H. T. Kung, "Synchronized and Asynchronous Parallel Algorithms for Multiprocessors," in Algorithms and Complexity: New Directions and Recent Results. J. F. Traub Ed., New York: Academic Press, 1976.
[PAT81] J. H. Patel, "Performance of Processor-Memory Interconnections for Multiprocessors," IEEE Transactions on Computers, Vol. C-30, No. 10, pp. 771-780, 1981.
[PRI76] T. G. Price, "A Note on the Effect of the Central Processor Service Time Distribution on Processor Utilization in Multiprogrammed Computer Systems," Journal of the ACM, Vol. 23, No. 2, April 1976, pp. 342-346.
[WUL81] W. A. Wulf, R. Levin and S. P. Harbison, Hydra/C.mmp: An Experimental Computer System. New York: McGraw-Hill Book Company.

Table 1. Value of X estimated for different G/G/1//P systems for (P-1)u=1

P	u	CVS=0				CVS=.5				CVS=1	CVS=0	Approximation of Corollary 1
		CVT=.5	CVT=1	CVT=2	CVT=5	CVT=.5	CVT=1	CVT=2	CVT=5	CVT=1	CVT=0	
2	1	.443	.422	.416	.411	.430	.416	.412	.408	.398	.500	.414
4	1/3	.654	.631	.622	.617	.636	.622	.615	.612	.592	.750	.631
8	1/7	.777	.755	.748	.740	.760	.745	.738	.733	.715	.875	.768
16	1/15	.852	.835	.828	.816	.839	.826	.821	.818	.799	.938	.857
32	1/31	.901	.888	.881	.868	.890	.880	.875	.863	.858	.969	.914
64	1/63	.933	.922	.917	.894	.924	.915	.912	.890	.9	.984	.949
128	1/127	.954	.946	.941	.918	.948	.941	.937	.916	.928	.992	.970

Table 2. Relative error (%) on X different G/G/1//P systems.

P	u	CVS=0				CVS=.5				CVS=1	CVS=0
		CVT=.5	CVT=1	CVT=2	CVT=5	CVT=.5	CVT=1	CVT=2	CVT=5	CVT=1	CVT=0
2	1	-6.55	-1.90	-.48	+.73	-3.72	-.48	+.49	+1.47	+3.86	-17.2
4	1/3	-3.52	0	+1.45	+2.27	-.79	+2.57	+2.60	+3.10	+6.59	-15.87
8	1/7	-1.16	+1.72	+2.67	+3.78	+1.05	+3.08	+4.07	+4.77	+7.41	-12.23
16	1/15	+.59	+2.63	+3.50	+5.02	+2.15	+3.75	+4.38	+4.77	+7.26	-8.63
32	1/31	+1.44	+2.93	+3.75	+5.3	+2.70	+3.86	+4.46	+5.91	+6.53	-5.68
64	1/63	+1.71	+2.93	+3.49	+6.15	+2.71	+3.72	+4.06	+6.63	+5.44	-3.52
128	1/127	+1.68	+2.54	+3.08	+5.66	+2.32	+3.08	+3.52	+5.9	+4.53	-2.22

Table 3. Simulation results and model prediction for the central server model with multiple servers

P	L	branching probability distribution	(model)	(simulation) CVT=1 CVS=0	CVT=0 CVS=0	CVT=.5 CVS=.5
8	2	unif. ($p_i = \frac{1}{2}$)	.245	.228	.230	.225
8	4	unif. ($p_i = \frac{1}{4}$)	.372	.353	.359	.343
8	8	unif. ($p_i = \frac{1}{8}$)	.440	.435	.439	.424
8	16	unif. ($p_i = \frac{1}{16}$)	.471	.470	.472	.465
2	8	unif. ($p_i = \frac{1}{8}$)	.492	.492	.500	.492
4	8	unif. ($p_i = \frac{1}{8}$)	.475	.475	.479	.470
16	8	unif. ($p_i = \frac{1}{8}$)	.366	.344	.347	.333
8	4	triang. $[(p_1,p_2,p_3,p_4) = (\frac{1}{3},\frac{3}{8},\frac{3}{8},\frac{1}{8})]$	.341	.323	.325	.314
8	8	triang. $[(p_1,p_2,\ldots,p_8) = (\frac{1}{32},\frac{3}{32},\frac{5}{32},\frac{7}{32},\frac{7}{32},\frac{5}{32},\frac{3}{32},\frac{1}{32})]$	.422	.412	.416	.401
8	16	triang. $[(p_1,p_2,\ldots,p_{16}) = (\frac{1}{128},\frac{3}{128},\frac{5}{128},\frac{7}{128},\frac{9}{128},\frac{11}{128},\frac{13}{128},\frac{15}{128},\frac{15}{128},\ldots,\frac{1}{128})]$	.462	.460	.462	.452

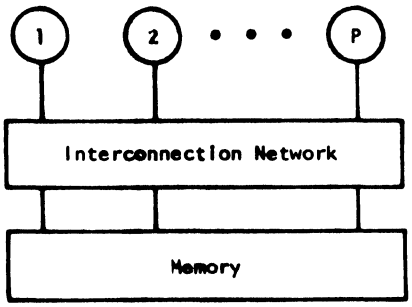


Figure 1. Tightly-coupled MIMD processor

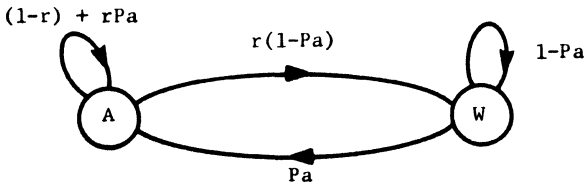


Figure 2. Markov graph for computing r_e

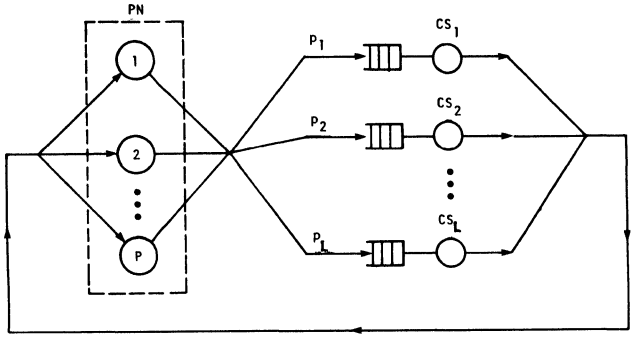


Figure 3. System with L critical sections shared by P processes

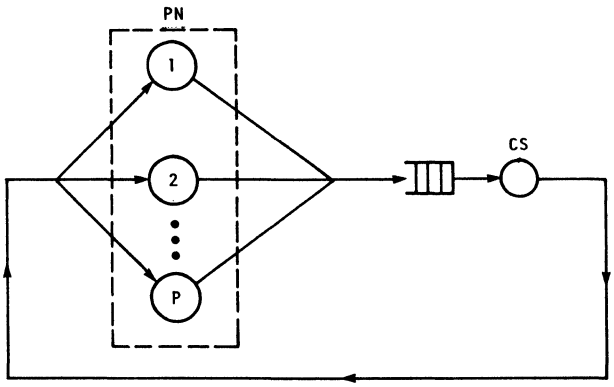


Figure 4. G/G/1//P system

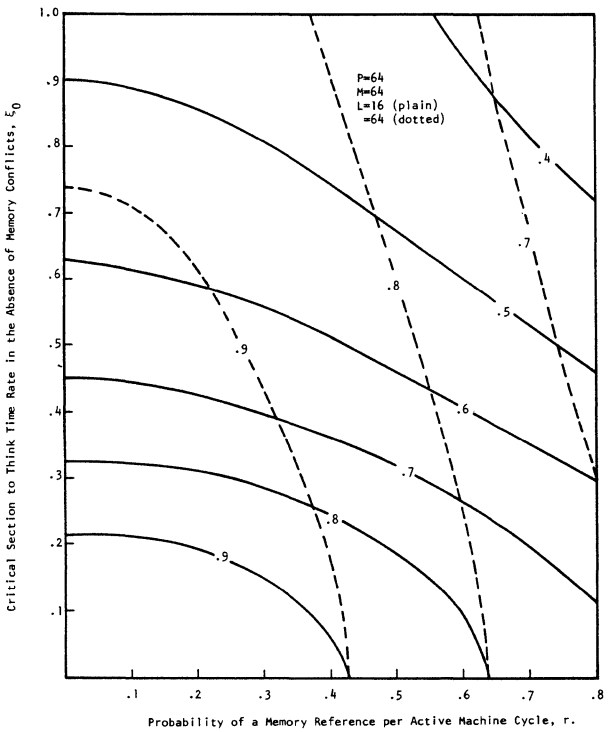


Figure 5. Feature planes for asynchronous algorithms in a tightly coupled system

THE AUTOMATED DESIGN OF TASK-SPECIFIC PARALLEL PROCESSING ARCHITECTURES

Matthew O. Ward
Visual Communications Research Department
Bell Laboratories
Holmdel, New Jersey 07733

1. Introduction

Interest in parallel processing has mainly stemmed from a requirement to perform vast amounts of computation at high speed on sometimes large quantities of data. For example, processing of video information in real-time for applications such as robotics requires analysis to be performed on data arriving at rates as high as 15 million picture points per second. Most systems to date which are capable of this speed are built almost entirely of special purpose hardware, which is expensive, time consuming to design and develop, and often restrictive in its applications. More flexible systems, such as SIMD machines, can perform tasks very quickly but are restricted as to the complexity of the tasks they can perform as well as being often difficult to program.

The research presented here is an attempt to overcome some of these problems. The goal is to take a set of algorithms which one wishes to execute very frequently, and automatically design an MIMD machine capable of executing the tasks at a desired speed. In the case of robotics these tasks will include image preprocessing, feature extraction, object classification, arm guidance and monitoring, and accessing and updating the knowledge base of the environment in which the robot works.

In this and many other environments designing systems around algorithms is a reasonable approach, due to the frequency of execution for each algorithm and the importance of high speed. Granted this methodology could not be used to develop a general purpose system, but it is fairly agreed upon that no single computer system is capable of satisfying all computing needs without exorbitant cost and underutilization. A major restriction is that once a system is designed the set of algorithms to be run on it are fixed, although it may be conjectured that algorithms in the same restricted environment may show enough similarity in data and control flow characteristics to allow fitting new algorithms to the architecture.

2. System Components

The basic subtasks involved in the research approach presented here are as follows.

2.1 Extraction of Parallel Processable Tasks from Sequential Programs

In a given program there are two types of parallelism which one can detect and utilize at the statement level without significantly modifying the original code. The first is noting when pairs of statements are mutually data independent,[2] i.e., one is not reading from a veriable which the other is attempting to write into. These are termed statement independent. The second type is recognizing when one iteration of a loop is unaffected by the progressing of another, i.e., iteration ordering is irrelevant. This is termed iteration independency. Thus, by comparing all pairs of statements as well as analyzing all loops one can learn for each occurrence of each statement in a program's execution the earliest time it can be executed (firing condition) as well as the latest point at which its results are needed by other statements (reset condition). The definition and format of these terms was introduced by Dervisoglu, [9] although the extraction method used in that work did not always produce correct results.

Two conditions must be true for a statement to execute properly, namely control flow which indicates the statement must execute at some time to insure correct results, and valid data must be available to use in calculations. These conditions together constitute a firing condition, which may be represented simply by a list of all statements which must execute prior to the firing of the given statement. Since multiple control paths may exist for each statement there may exist several possible conditions, of which at most one may be true at a given time. Thus, a sum-of-products representation is used, with the product terms being the individual statements and the sums being the separation of distinct paths. For example, in the following section of code statement 5 cannot execute until either the true branch of 2 fires or statement 4 executes. Statement 2 is a control component while all others are data components. Note the reduction by precedence rules.

```
                x = n*y          (1)
If              (x<m)            (2)
                m = m/y          (3)
                x = x/y          (4)
                r = x + n        (5)
        p(5) = 1*2t + 1*2f*4
             = 2t +4
```

Once the parallelism is extracted a simulation is performed of the parallel execution of the program. This is done to ensure that sufficient savings in execution time are possible, thus, meriting a continued effort at designing a corresponding hardware architecture. This is an

0190-3918/82/0000/0298$00.75 © 1982 IEEE

important stage as it has been found that many algorithms do not lend themselves to significant parallelization, either due to the form of the particular implementation of the algorithm or the nature of the algorithm itself.

2.2 Processor Allocation and Communications Requirements

As a first step towards designing architectures to fit particular algorithms the system must attempt to determine the minimum number of processors needed to take advantage of all of the available parallelism while at the same time reasonably minimizing the amount of interprocessor communications, as this is the main bottleneck of any well-balanced parallel processing environment. Although it can be readily shown that the optimization of either of these problems is NP-complete satisfactory results can often be derived using partially analytic and partially heuristic-guided construction in a bounded amount of time.

Maximum parallelism can be easily insured by assigning tasks to processors such that no two tasks on the same processor will ever be ready for execution at the same time. This is directly derivable from the firing conditions by noting that tasks which have data dependencies or control conflicts may reside on the same processor. Some methods used for reducing the number of processors as well as interprocessor communications include assigning processes in order of decreasing interprocess communications as estimated by approximating loop counts, and limited lookahead in evaluating more globally the cost incurred in assigning a process to various processors.

At this point, knowing the processes which will reside on each processor, estimates of both processing and storage requirements of each processor and some general information concerning interprocessor communications will be known. This information is useful both in avoiding excessive system cost by specifying minimum component requirements and also helping to decide how to group components of similar requirements in the event that cost or component constraints require 'collapsing' of the resulting architecture.

2.3 Architectural Specifications Based on Functional and Communications Requirements

Given the functionality (processing and data and control communications) requirements of the algorithms in their parallel form, an architecture must be designed with the appropriate functional capabilities. The previous two sections have outlined the extraction and grouping of the processing and communications characteristics required, and this information is now used in conjunction with an architectural components knowledge base to design one or several hardware configurations capable of executing the algorithm in parallel.

Some of the information included in this knowledge base are details of the computing capabilities of processing elements, size and addressing means of memory, and bandwidth and control protocol of links. Perhaps the most important information is that of interfacing specifications for each device, thus, avoiding the design of 'impossible' hardware architectures.

The procedure then is to first locally match processor requirements with processor capabilities and interprocessor traffic with link bandwidth, and then refine selection using compatability relationships, working outward until a totally defined, compatible (able to be interfaced) system is produced. Obviously provisions must be made to resolve deadlocks in the procedure, i.e., when there exists no alternatives which allow components to be linked. This often will entail decreases in speed or increases in cost, which will be user specified.

2.4 Compiling Parallel Processable Tasks Into Architecture Dependent Executable Form

Once an architecture has been designed and constructed the tasks assigned to each processor must be converted to an executable form, including message passing protocol, firing and resetting expression evaluation, and, of course, the program code itself. The simplistic operating system required on each processor to perform these tasks has several advantages over those on existing distributed systems. Firstly, the ordering of operations is totally deterministic in that all essential orderings are preserved by the parallelization process. In addition, communications is less a problem than in general purpose systems, as more is known of the interactions between processors which will take place. Thus, it is possible to do much pre-execution anticipatory work.

The basic series of tasks to be executed on each processor of the system will be as follows.

a. Receive a message concerning the firing or resetting of a statement.

b. Evaluate firing and resetting conditions for all statements awaiting this message on the processor.

c. If a firing condition is true then

   c1. Gather required data for execution.

   c2. Execute the statement.

   c3. Send messages and possibly resulting values to all processors which are awaiting its completion.

d. If a reset condition is true then

   d1. Reset the state for that statement so the firing condition is again evaluated for future reexecution.

   d2. Send messages to all processors which are awaiting the resetting.

As many of these tasks will be identical in form for each task and processor a hardware implementation of many of these components is logical, especially in communications and expression evaluation. This is important, as these tend to be the major bottlenecks in parallel processing systems.

3. Current Status

At the time of writing a significant percentage of the system has been designed, implemented, and tested. The work described in Sections 2.1 and 2.2 has been completed, accepting as input normal programs written in a large subset of C and producing firing and resetting conditions as well as processor assignments. The knowledge base has been designed and a skeleton for the entering and querying of information has been completed. A study is underway to determine the significant attributes needed to describe architectural components to use in the automated design process. Likewise, an algorithm for creating hardware architectures using the knowledge base and the algorithm requirements has been designed, the implementation of which will be completed when the knowledge base is available. A system for compiling the tasks into executable modules for a test bed of Motorola 68000 microcomputers has been completed. Other processors can be easily incorporated with a suitable cross-compiler and a small number of processor-specific I/O routines. Finally, a communications processor is being designed to reduce losses in speed due to communications and condition evaluation.

The simulated parallel execution of several algorithms has been compared to corresponding sequential execution to check for both equivalence in results and estimated speedup. Processor assignment for a number of short programs (< 40 lines) has been checked against optimal assignment located by analytic (exhaustive search) methods with highly satisfactory results. Assignments made for larger programs, although difficult to thoroughly assess, have been fairly satisfactory, although it can be seen that additional heuristics may be beneficial.

4. Conclusions

A general description has been presented of a methodology for automatically designing special-purpose parallel processing architectures given the tasks which are to be performed. Results to date have been quite encouraging as to the effectiveness of the technique. Obviously, the method would be relatively useless in designing general-purpose systems, unless a set of representative algorithms could be devised which would cover a nearly complete spectrum of program types. It is believed that this is not possible, agreeing with the idea that no single architecture could ever satisfy all possible user needs. A major determinant in the effectiveness of the method was the actual implementation of the algorithms used as input. It was observed that minor changes in the implementation could result in major increases in parallelism and reduction of interprocessor communications. Thus, as work progresses a set of rules is being developed as a guide for writing programs to best exploit parallelism, several of which will be implemented into a precompiler to relieve the user of the need to modify his or her programming style.

5. References

[1] Arvind, Decomposing a Program for Multiple Processor Systems, Proceedings of the 1980 IEEE Conference on Parallel Processing.

[2] A. J. Bernstein, "Analysis of Programs for Parallel Processing", IEEE Trans. on Computers, Vol. EC-15, No. 5, Oct., 1966.

[3] J. B. Dennis, K. Weng, An Abstract Implementation for Concurrent Computations With Streams, Proceedings of the 1979 IEEE Conference on Parallel Processing.

[4] D. J. Kuck, "Parallel Processing of Ordinary Programs", Advances in Computers, Academic Press, Vol. 15, 1976.

[5] Kuck, D. J. High-Speed Multiprocessors and Their Compilers, Proceedings of the 1979 IEEE Conference on Parallel Processing.

[6] H. Lorin, Parallelism in Hardware and Software: Real and Apparent Concurrency, Prentice Hall, 1972.

[7] W. C. McDonald, T. G. Williams "Evaluation of Multimicroprocessor Interconnection Networks for a Class of Sensor Data Procession Problems", SPIE Vol. 241, Real-Time Signal Processing III, 1980.

[8] C. V. Ramamoorthy, M. J. Gonzalez, A Survey of Techniques for Recognizing Parallel Processable Streams in Computer Programs, Fall Joint Computer Conference, 1980.

[9] B. I. Dervisoglu, Modeling Maximum Parallel Executions in Pipeline Executable Form Using Precedence Expressions, University of Connecticut Technical Report, CS-79-5.

A BIT-SEQUENTIAL MULTI-OPERAND INNER PRODUCT PROCESSOR

H.J. SIPS
Delft University of Technology

Abstract- A bit-sequential multi-operand inner product processor is described with $O(2N.n)$ complexity where N is the number of product pairs and n is the wordlength of the operands. The operands have variable precision. The result is available d clock cycles after the absorption of the last bit of the operands. The parameter d is a small positive constant.

I. The algorithm

The summation of product terms is necessary in a large number of scientific computations such as matrix manipulations, signal processing, etc.. The inner product function is defined as:

$$P = \sum_{j=1}^{N} A_j B_j \qquad (1)$$

To achieve a high computing speed most two operand sequential processors transport their data and variables in a bit-parallel manner. If instead of a sequential processor a parallel processor approach is taken this results in the use of many bit-parallel buses. This imposes severe restrictions on the constructability of a parallel computer system. These restrictions do not disappear in VLSI. As a means to overcome computational and interconnection complexity the use of bit-sequential processing can be considered. Swartzlander et al.[SWAR78] considered a quasi-serial implementation of the productterms using a parallel counter multiplier.
Because bit-sequential processing intrinsicly slows down the computation time it is very important to use the datatransport lines effectively i.e. the bits on the in- and output operand lines should be significant on each time step. On-line and semi on-line algorithms have this property. On-line algorithms are defined by the property that to generate the j-th digit of the result it is necessary and sufficient to have the operands available up to the $(j+d)$-th digit, where d is a small positive constant. On-line algorithms with the least significant digit first have been developed among others by Atrubin [ATRU65] and Chen [CHEN79]. Trivedi et al. [TRIV77] has developed algorithms for the most significant digit first using a redundant number system. On-line algorithms are only efficient if long expressions have to be evaluated. If, however, a recurrent equation is solved the result must be delayed by $(n-d)$ time steps.
Semi on-line (SOL) algorithms are defined [SIPb82] by the property that d clock cycles after the absorption of the last digits of the operands the first digit of the result is available. The parameter d is now a small positive constant which is independent of the word-size n and for which holds: $d(s)<d(p)$. $d(s)$ is the semi on-line delay and $d(p)$ is the delay in a full bit-parallel implementation of the arithmetical operation. In the multi-operand case we can further demand that the hardware is of $O(M.n)$ complexity where M is the number of operands and n is the wordsize. the effective delay of the semi on-line algorithm is $(n+d)$ cycles. The operation (1) is evaluated in $(n+d)$ clock cycles using semi on-line algorithms. The condition $d(s)<d(p)$ implies processing of the operands during transportation. Semi on-line algorithms can also be defined with the most significant digit first or with the least significant digit first.
The number A is represented by the bitstring $\{a(n)a(n-1)...a(1)\}$ where $a(1)$ is the bit that is entered into the processor first. If A is a sign-magnitude number bit $a(1)$ is always the sign-bit. If A is a two's complement number the bits proceed in normal order. The algorithm is based on the technique of incremental multiplication [TRIV77] [CHEN79]. The algorithm is extended for multi-operand operations and two's complement numbers. Let:

$$A_j(k)=\{a_j(k)a_j(k-1)...a_j(1)\} \qquad (2)$$

$A_j(k)$ is A_j up to the k-th bit. From (2) follows:

$$A_j(k)=A_j(k-1) + a_j(k).2^t \qquad (3)$$

$t=-k+1$ and $k>1$ for MSB-SM ($a(1)$=sign bit)
$t=k-2$ and $k>1$ for LSB-SM ($a(1)$=sign bit)
$t=k-1$ and $k\geq 1$ for LSB-TC
$t=-k$ and $k\geq 1$ for MSB-TC
$A_j(0)=0$ for TC; $A_j(1)=0$ for SM

It then follows that;

$$A_j(k).B_j(k)=A_j(k-1).B_j(k-1)+A_j(k-1).b_j(k).2^t+ B_j(k).a_j(k).2^t \qquad (4)$$

Suppose;

$$P_j(k)=A_j(k).B_j(k).2^{-t} \quad , \qquad (5)$$

$P_j(0)=0$ for TC, $P_j(1)=0$ for SM

From this the following recurrent equation can be derived;

$$P_j(k)=2^{-s}.P_j(k-1)+A_j(k-1).b_j(k)+B_j(k).a_j(k) \qquad (6)$$

$s=-1$ for the MSB first algorithms
$s=1$ for the LSB first algorithms

This can be done for every product in equation (1);

$$P(k)=2^{-s}.P(k-1)+\sum_{j=1}^{N}\{A_j(k-1).b(k)+B_j(k).a_j(k)\} \qquad (7)$$

Each recursion step the evaluation of (7) requires the full addition of $2N+1$ operands.
The generation of the partial product for sign-magnitude numbers is straightforward according to (6) since the magnitudes can be interpreted as positive numbers. The signs of A_j and B_j determine whether the partial product $P(k)^j$ is weighted as a positive or a negative number in (7).
A two's complement number can be expressed as:

0190-3918/82/0000/0301$00.75 © 1982 IEEE

$$A=-2^{n}a(n) + \sum_{i=1}^{n-1} a(i).2^{i-1} \qquad (8)$$

(LSB first case) which means that all the bits besides bit $a(n)$ can be treated as if they where positive. The bit $a(n)$ gives a negative weight to $B(n)$ in equation (7).

II. Implementation

To achieve enough speed in solving (7) the operation must be done in a pipelined way. There are two phases to be distinguished in the evaluation of equation (7):

1. Compression of the partial operands in a sum and a carry vector.
2. Addition of 1. and the shifted partial product generated in the previous time step.

The phases 1 and 2 can be done in a pipelined way. For large values of N internal pipelining of phase 1 may be necessary. The approach is to use a carry save cellular array of dimension $2N.n$. In this case the delay is linearly dependent on the number of operand pairs N. An example of a carry save adder array is shown in figure 1. The cells of the array consist of full adders. The array produces a sum and a carry vector. Note that in figure 1 the top row can be deleted for positive operands. The b_2 operand can be directly feeded into the full adders of row 2. The elements are included for regularity since additional logic must be included in each cell.

For the addition in equation (7) the operands must be in two's complement form. If the operands are sign-magnitude numbers they must be converted to two's complement numbers. When both operands in the multiplication process are sign-magnitude numbers and have the same sign they are already in the correct form so no conversion is needed. When the operands have opposite signs the weight of the product pair will be negative. The complementing of both partial operands can be done by complementing the individual bits of the operands and adding a 1 to the operand. The addition of a 1 to the operand is the same as putting a 1 on the c_{in} inputs in figure 1 if the corresponding operand is negative. So the sign bits in the sign-magnitude format are not involved in the computation of the partial operands according to equation (7). The combination of the sign bits of the operand pair only determines the positive or negative weight of the total operand pair.

When both operands are two's complement numbers the operands are already in the desired form. Only the sign bit must be interpreted as a negative weighting factor. Here the sign bits of the operands are explicitly involved in the computation of the operands accordig to equation (7).

For the sign determination of the result the method of Agrawal et al. [AGRA78] is followed. This method prevents the extension of the operands by inverting the sign bits and adding a fixed correction factor C to the sum (C inputs in figure 1).

The addition of the righthandside terms of eq. (7) besides $2^{-s}.P(k-1)$ can be done in the way described above. The result has $ent\lceil log_2 2N\rceil + n$ bits. Each time step the partial product generated in the previous time step must be added to the sum of the partial operands. This can be accomplished by $(5,3)$ counters. This is the same method as in [CHEN79],[SIPa82]. These counters are shown in figure 1. The outputs of the counters $(S,C(1),C(2))$ are wired according to the algorithm (MSB or LSB). The counters contain three storage cells to save the partial product. Figure 2 shows the wiring for the MSB and for the LSB algorithm. It can be seen that in the LSB wiring of the $(5,3)$ counters only nearest neighbour interconnections occur.

The carry save adder array description in figure 1 does not include the control mechanism needed to generate the partial operands. The aim must be to design a cell which is simple of structure and has a minimum of interconnections to the outside world. From eq. (7) it can be seen that each time step a new bit is appended to the operands. Whether or not the operand participates in the addition of that time step is dependent on the new bit of the corresponding other operand. The minimum needed is a full adder, one storage cell to hold the operand and a few gates. Figure 3 shows the basic cell layout. The operand line (here B_j has been chosen) is a line along all cells where the operand is to be stored (rowwise). They succesively activate each column. The control signal $q(k+1)$ loads each new bit of the operands in the next column. The control line $g=a_j(k)$ is the one step delayed value of one bit of the corresponding operand $A_j(k)$ and qualifies the operand $B_j(k)$ according to eqation (7). The treatment of $A_j(k-1)$ is a little different because of the difference in the k-index. Therefor the new bit $a_j(k)$ must be suppressed. This is indicated by the signal $q(k)$ which is the one step delayed signal $q(k+1)$. The complement signal determines whether the operand is positive ($com=0$ and $c_{in}=0$) or negative ($com=1$ and $c_{in}=1$).

After processing the last bit of the operands the result must be available as soon as possible. In the MSB first case the result is stored in the $(5,3)$ counters. A fast carry propagating adder is necessary to determine the result. The propagation delay determines the delay d of the operation. The result is then in two's complement form. If a result in sign-magnitude is required an extra complementing step is needed. This can be done while the sign bit is placed on the output. In the the LSB first case there is no fast carry propagating adder necessary if the result is required in two's complement form because the MSB half of the product can be calculated during the output transfer of the result. There is, however, in overlapped computation an extra $(5,3)$ counter counter necessary because the accumulating $(5,3)$ counter is needed in the next inner product evaluation. If the result has to be in sign-magnitude form a fast carry propagating adder is necessary to determine the sign bit.

Figure 4 shows a numerical example of the LSB first two's complement algorithm.

Another property of the multi-operand processor is the improved dynamic accuracy. If the inner product has a mixture of positive and negative operands the multiplication of product pair j may overflow in the sense that the result contains more than n bits without causing an overflow of the final result. (This is shown in figure 4).

III. References

1. [AGRA78] D.P. Agrawal, T.N. Rao, "On multi-operand addition of signed numbers", IEEE Transactions on Computers, Vol. C-27, November 1978.
2. [ATRU65] A.J. Atrubin, "A one dimensional real-time iterative multiplier", IEEE Transactions on Computers, Vol. EC-14, June 1965.
3. [CHEN79] I.N. Chen, R. Willoner, "An O(n) parallel multiplier with bit-sequential input and output", IEEE Transactions on Computers, Vol. C-28, October 1979.
4. [SIPa82] H.J. Sips, "Comments on: An O(n) parallel multiplier with bit-sequential input and output", IEEE Transactions on Computers, April 1982.
5. [SIPb82] H.J. Sips, "A bit-sequential approach to parallel processing", (submitted to IEEE Transactions on Computers)
6. [SWAR78] E. Swartzlander, B. Gilbert, I. Reed, "Inner product computers", IEEE Transactions on Computers, Vol C-28, December 1979.
7. [TRIV77] K.S. Trivedi, M.D. Ercegovac, "On-line algorithms for division and multiplication", IEEE Transactions on Computers, Vol. C-26, July 1977.

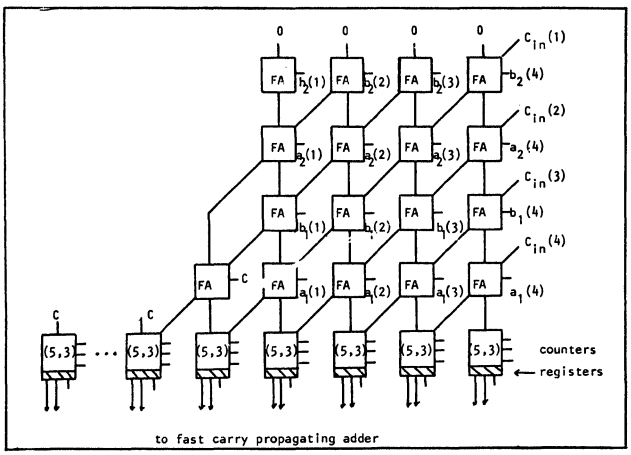


Figure 1. Carry save array

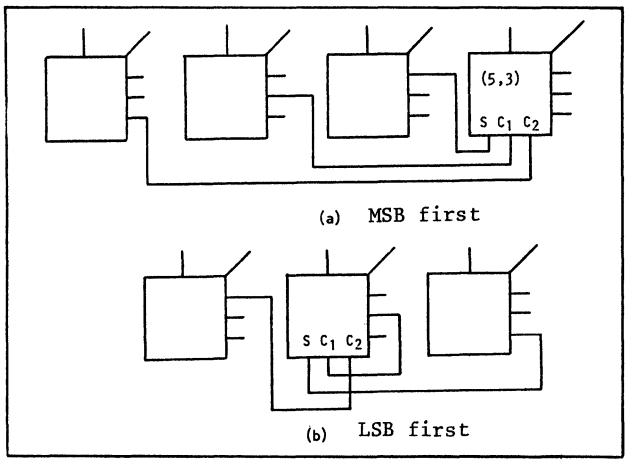


Figure 2. (5,3) counter wiring

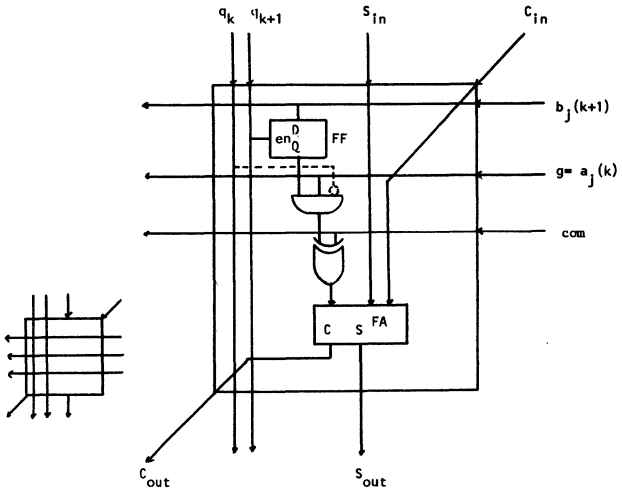


Figure 3. Basic cell layout

A * B + C * D

101110 * 001010 + 001110 * 001011 = 100110

-18 * 10 + 14 * 11 = -26

A(0).b(1)	1 00000		1 00110
B(1).a(1)	1 00000		1 01010
C(0).d(1)	1 00000		1 00110
D(1).c(1)	1 00000		1 01011
corr.fact.	110		110
	000 0 00000 +		000 1 00001 +
P(0)	000 0 00000 +	$P(3).2^{-1}$	000 0 00011 +
P(1)	000 0 00000	P(4)	000 1 00100
A(1).b(2)	1 00000		1 00000
B(2).a(2)	1 00010		1 00000
C(1).d(2)	1 00000		1 00000
D(2).c(2)	1 00011		1 00000
	110		110
	000 0 00101 +		000 0 00000 +
$P(1).2^{-1}$	000 0 00000 +	$P(4).2^{-1}$	000 0 10010 +
P(2)	000 0 00101	P(5)	000 0 10010
etc.	1 00000		1 00000
	1 00010		0 10101 +1
	1 00000		1 00000 C_{in}
	1 00011		1 00000
	110		110
	000 0 00101 +		111 1 10110 +
$P(2).2^{-1}$	000 0 00010 +	$P(5).2^{-1}$	000 0 01001 +
P(3)	000 0 00111	P(6)	111 1 11111

Figure 4. Numerical example

A DIGIT ONLINE ARITHMETIC SIMULATOR

Bryan Gerard Mackay
Mary Jane Irwin
Department of Computer Science
The Pennsylvania State University
University Park, PA 16802

Abstract -- Digit online arithmetic has a great deal of potential for the speedup of computation. Digit online algorithms have the property that in order to generate the j-th most significant digit of the result it is sufficient to have the first j+k most significant digits of the operands. The difference k is a small predefined constant corresponding to an online delay. This paper presents a software package that simulates the operation of computational systems which use digit online arithmetic. The simulator provides the ability to investigate the advantages and disadvantages of using digit online arithmetic for various applications.

Introduction

In recent years a good deal of research has been directed towards digit online algorithms and their corresponding architectures [2],[3],[4],[5],[6],[7]. These algorithms may be realized by special arithmetic systems, one of which uses digit online pipelines. Online algorithms have the property that in order to generate the j-th most significant digit of the result it is sufficient to have the first j+k most significant digits of the operands. The difference k is a small predefined constant. Thus, after a startup delay of k steps an online algorithm will generate one digit of the result at each step.

The advantage of using digit online pipelines is demonstrated in systems that involve the chaining together of many pipelines. Machines such as the CRAY-1 use the technique of chaining on words to achieve the fastest processing speed. This involves connecting the output of one pipeline to the input of another. If two conventional pipelines are chained together in this way the second pipeline cannot begin processing until the first pipeline has produced its first result. Online pipelines are not strung end to end but side to side to achieve chaining on digits, creating an online pipeline network [3]. The attractiveness of online pipeline networks can be seen most dramatically in the computation of recursive equations. In such a network the computation of f_{i+1} may begin as soon as the first digit of f_i becomes available. The improvement in processing speed over conventional pipeline networks can be dramatic.

This paper presents a software simulation for operations and expressions evaluated using digit online algorithms first presented in [4]. The simulator implements floating point addition (subtraction), multiplication, division, and square root in a fully digit online manner. The simulator was designed to provide the ability to create and analyze a wide range of digit online pipeline networks. In this way it helps to determine the advantages and disadvantages of using online arithmetic to solve various problems.

Design of the Simulator

The simulator package consists of about two thousand lines of PASCAL code on a VAX 11/780 system running UCB VMUNIX. The programs were designed to be highly interactive and easy to use. Consequently they can be used to demonstrate the concepts of digit online arithmetic to those who are unfamiliar with this subject. The simulator can be run so that the operand digits are requested from the user as they are needed and the result digits are displayed as they become available. In this way the operation of the algorithms may be directly observed.

The programs simulate the algorithms RADD, RMUL, RDIV, and RSQR presented in [4]. The algorithm NORM has been implemented to help normalize operands. The digit online algorithms were designed in such a way that they may be implemented with a limited number of hardware primitives requiring a small number of gate delays [4]. The four primitive operations implemented in the simulator are:

1) Selection of a fixed point value based on a two digit value. This operation may be performed using 2 gate delays in a simple table lookup fashion.
2) Addition of two fixed point values. This operation is performed by a signed digit addition that requires 4 gate delays.
3) Multiplication of a fixed point value by a single digit value. This operation takes 6 gate delays.
4) Shifting a fixed point value by a constant value. In hardware this operation could be accomplished by simply offsetting the interconnections between corresponding components in 0 gate delays.

All of the more complex operations in the floating point algorithms are manipulated so that they can be expressed in terms of these four simple functions. Each iteration of an algorithm is simulated by a series of calls to these primitive functions producing the desired result. In this way the simulation carries out the operations in the same way that they would be performed in hardware by a digit online pipeline. The simulator also keeps track of the number of gate delays that have elapsed at the end of each step. This will give the user an idea about the processing speed of the network.

The algorithms RADD, RMUL, RDIV, and RSQR operate by taking one digit of each operand per iteration and generating approximations to the

0190-3918/82/0000/0304$00.75 © 1982 IEEE

characteristic and mantissa of the result in the fashion of the traditional continued sums/products algorithms [1]. The algorithms are online with respect to their inputs but since the result is not available until the final iteration they are not online with respect to their output. Fortunately, there are algorithms that when given the approximations to the result out of these algorithms, can generate the result in an online manner. Two of these functions were programmed into the simulation.

The first such function is the discretization algorithm DISC [4]. When supplied with the approximations to some result z, DISC will generate z in a digit online manner. By using DISC, the operations of addition (subtraction) and multiplication will be performed with an online delay of one. Both division and square root will have an online delay of three. A problem with DISC is that it tends to generate unnormalized results. The algorithm RADD, for example, always preshifts the mantissa of the result one position to the right to avoid mantissa overflow, so usually the result will be unnormalized. Unnormalized results increase the error in a system [5]. They also cause problems when these results are used as the input to a process that does not accept unnormalized operands such as RDIV.

Another digit generating algorithm, MOSN, may be used to decrease the probability of unnormalized results. MOSN is constructed so that the last characteristic digit of the result is not computed until the first approximation to the mantissa of the result becomes available. Using the first mantissa approximation, MOSN determines how many places the mantissa can be shifted to the left without causing overflow. In this way, MOSN is able to normalize many results that would otherwise be unnormalized. The online delay of algorithms using MOSN will be one greater than the delay out of DISC. Table 1 shows a comparison of a divide operation using DISC and MOSN. As may be seen from this example the result when using MOSN is closer to the correct result. The penalty for using MOSN is an additional step. The simulator uses redundant base 8 arithmetic.

Table 1 - A Comparison of DISC and MOSN

Result of RDIV and DISC.

```
PROCESS NUMBER 1:  Began at time = 0.
02:221̄6̄ (1.803125E+01) is the dividend b.
10:453̄4̄ (9.617408E+06) is the divisor c.
1̄3:041̄4̄ (1.877546E-06) is the quotient a.
                   Ended at time = 288.
```

Result of RDIV and MOSN.

```
PROCESS NUMBER 1:  Began at time = 0.
02:221̄6̄ (1.803125E+01) is the dividend b.
10:453̄4̄ (9.617408E+06) is the divisor c.
1̄2:414̄3̄ (1.874752E-06) is the quotient a.
                   Ended at time = 320.
```

The actual quotient is 1.874855E-06.

Experimental Results

One of the major accomplishments of the simulator to date has been to clearly demonstrate the areas of concern that exist when using digit online arithmetic. By simulating various online pipeline networks, the magnitude of these concerns was observed. The simulator also served as a useful tool for finding solutions to some of these concerns. The primary concern is generation of unnormalized results. When unnormalized values occur in a network, gradual mantissa underflow may occur [5] since an unnormalized value may not be as precise as it could be. By shifting the mantissa left to remove leading zeroes, more digits of the result may be added increasing the precision. An even bigger problem with unnormalized values in a network is that they may not be used as the divisor in RDIV or as the radicand in RSQR.

The algorithms RADD, RMUL, and RDIV all have the possibility of generating unnormalized results. The algorithm RSQR however, is an interesting exception. RSQR will operate correctly provided that the mantissa is normalized and positive. If the radicand satisfies these requirements then the result of RSQR will also be normalized. When combined with DISC, RSQR will compute the square root in an online manner with an online delay of three. This result will be normalized, so it may be used as the radicand of another square root operation or the divisor of a division operation. Thus, it is possible to construct an online pipeline network to compute the n-th root of x, where n is a power of 2, provided that x is positive and normalized. This network will have an overall delay of $4(\log_2 n)-1$ steps. Table 2 shows the simulation of a network to compute the eighth root of a number.

Unfortunately, the algorithms RADD, RMUL, and RDIV do not have the desirable property that they always generate normalized results. In fact, when combined with DISC, these algorithms will usually generate unnormalized numbers. MOSN only decreases the probability of unnormalized results. But no algorithm which can truly be said to be digit online can guarantee that the result will be normal-

Table 2 - A Network to Compute Eighth Root

```
Enter the expression(s) (Type "." to stop.):
 1: d:=^c:=^b:=^a.
Enter the 2 characteristic digits of a.  1 4
Enter the 6 mantissa digits of a.  2 7 6 5 7 0

PROCESS NUMBER 1:  Began at time 0.
14:276570 (2.559993E+10) is the radicand a.
12̄:51̄0400 (1.600000E+05) is the square root b.
                   Ended at time = 352.

PROCESS NUMBER 2:  Began at time = 128.
12̄:51̄0400 (1.600000E+05) is the radicand b.
03:620000 (4.000000E+02) is the square root c.
                   Ended at time = 480.

PROCESS NUMBER 3:  Began at time = 256
03:620000 (4.000000E+02) is the radicand c.
02:240000 (2.000000E+01) is the square root d.
                   Ended at time = 608
```

ized for all possible values of the operands, especially in the case of cancellation during subtraction. One way to guarantee that the result will be normalized is to put restrictions on the operands so that cases such as the above do not occur.

Increasing the online delay of certain operations is another way of guaranteeing that a value in an online pipeline network can be normalized. The algorithm NORM has been incorporated in the simulator to provide the user with the ability to specify the online delay of a process. NORM receives the inputs in an online fashion and generates a result, with the same value as the operand, after an online delay of one. If the first digit of the operand is a zero, NORM will be able to shift the mantissa one position to the left and adjust the characteristic accordingly. NORM is also written so that by recursively applying NORM to an unnormalized number enough times, that number will eventually be normalized. Thus, an unnormalized mantissa such as $0.100\ldots00\bar{1}_8$ will become $0.100\ldots0\bar{1}7_8$ after one application of NORM, and $0.077\ldots7_8$ after n applications of NORM.

The simulator package has been used to simulate many practical online pipeline networks. One such network is for the LU decomposition of an n by n tridiagonal matrix using the recurrences:

$$d_0 = b_0$$

$$d_i = b_i - c_{i-1}\,(a_i/d_{i-1}) \text{ for } 1 \le i \le n-1$$

The computation $\ell_i = a_i/d_{i-1}$ is treated as a desirable by-product. Table 3 shows the simulation of one iteration of the LU decomposition for

$$\begin{bmatrix} 2 & 3 & 0 & 0 \\ 6 & 20 & 7 & 0 \\ 0 & 22 & 5 & 5 \\ 0 & 0 & 36 & 4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 \\ 0 & 2 & 1 & 0 \\ 0 & 0 & -4 & 1 \end{bmatrix} * \begin{bmatrix} 2 & 3 & 0 & 0 \\ 0 & 11 & 7 & 0 \\ 0 & 0 & -9 & 5 \\ 0 & 0 & 0 & 24 \end{bmatrix}$$

This example shows the advantage of using digit online arithmetic. As can be seen from the starting and ending times of the processes, the computations of ℓ_1, d_1, and ℓ_2 are performed in parallel.

One problem with this application is that the result d_{i-1} may be unnormalized and therefore the computation of $\ell_i = a_i/d_{i-1}$ may be incorrect. One solution to this problem would be to restrict the values of a_i, b_i, and c_i in such a way that d_{i-1} will be normalized. These restrictions would obviously reduce the usefulness of the system. Another solution is to apply the algorithm NORM to the divisor.

Conclusions

This paper has presented a highly functional simulator for digit online algorithms. Since it was designed to perform in the same way that a straightforward hardware implementation of the algorithms would operate, the simulations of online pipeline networks will experience the same problems that the actual networks would encounter. The possibility of unnormalized divisors occuring in a network is one such problem. The need to avoid unnormalized values leads to tradeoffs in processing speed, restrictions on inputs, and the precision of results. The simulator allows the user to investigate how these tradeoffs come into play for various applications.

Table 3 - LU Decomposition

```
PROCESS NUMBER 1:  Began at time = 0
01:600000 (6.000000E+00) is the dividend a[1].
01:200000 (2.000000E+00) is the divisor d[0].
01:300000 (3.0000E+00) is the quotient ℓ[1].
                  Ended at time = 384

PROCESS NUMBER 2:  Began at time = 160.
01:300000 (3.000000E+00) is the multiplicand c[0].
01:300000 (3.000000E+00) is the multiplier ℓ[1].
02:110000 (9.000000E+00) is the product.
                  Ended at time = 480.

PROCESS NUMBER 3:  Began at time = 256.
02:240000 (2.000000E+01) is the addend b[1].
02:110000 (-9.000000E+00) is the augend.
02:130000 (1.100000E+01) is the sum d[1].
                  Ended at time - 576.

PROCESS NUMBER 4:  Began at time = 352.
02:260000 (2.200000E+01) is the dividend a[2].
02:130000 (1.100000E+01) is the divisor d[1].
01:200000 (2.0000E+00) is the quotient ℓ[2].
                  Ended at time = 736.
```

Bibliography

[1] DeLugish, B.G., "A Class of Algorithms for Automatic Evaluation of Certain Elementary Functions in a Binary Computer", Ph.D. Thesis, Report 399, DCS, Univ. of Il., June 1970.

[2] Ercegovac, M.D. and A.L. Grnarov, "On the Performance of On-Line Arithmetic", Proc. 1980 Inter. Conf. on Parallel Processing, pp. 55-62, August 1980.

[3] Owens, R.M. and M.J. Irwin, "On-Line Algorithms for the Design of Pipeline Architectures", Proc. Sixth Annual Symp. on Computer Arch., pp. 12-19, April 1979.

[4] Owens, R.M, "Digit On-line Algorithms for Pipeline Architectures", Ph.D. Thesis, Report CS-80-21, Dept. of Computer Science, Penn State Univ., August 1980.

[5] Owens, R.M., "Error Analysis of Unnormalized Arithmetic", Dept. of Computer Science, Penn State Univ., August 1981.

[6] Raghavendra, D.S. and M.D. Ercegovac, "A Simulator for On-line Arithmetic", Proc. Fifth Symp. on Computer Arith., May 1981.

[7] Trivedi, K.S. and M.D. Ercegovac, "On-line Algorithms for Division and Multiplication", IEEE Transactions on Computers, Vol. C-26, No. 7, pp. 681-687, July 1977.

A PARALLEL ARCHITECTURE FOR ACOUSTIC PROCESSING IN SPEECH UNDERSTANDING

Edward C. Bronson
Leah J. Siegel

Purdue University
School of Electrical Engineering
West Lafayette, IN 47907

Abstract — Speech understanding is a complex task which requires extensive computation. To increase the processing speed, a speech understanding system can be decomposed into tasks which can be performed by a series of distributed processing sub-systems. An architecture to perform acoustic processing is described in this paper. The parallel architecture for acoustic processing calculates characteristic parameters which describe the input speech signal. The types of operations performed include digital filtering, FFTs, linear predictive coding, autocorrelation calculations, and pitch analysis. The architecture is a multiple-SIMD system using the MC68000 microprocessor as the basic processing element. Using realistic assumptions from existing speech understanding systems, the attributes of the parallel system to perform acoustic processing for real-time speech understanding are derived. In particular, details about the organization and the number of processors in each of the component SIMD sub-systems are obtained. Interconnection network requirements are determined from the SIMD algorithms used. Timing analysis is performed.

I. Introduction

A speech understanding system accepts spoken speech input, derives a conceptual understanding of the input, and produces a response. In a typical system, a number of knowledge source components interact to resolve the errors and ambiguity inherent in human speech. These knowledge sources perform operations such as acoustic parameterization, phonetic interpretation, lexical processing, syntactic analysis, semantic interpretation, and response generation. Existing speech understanding systems that have been developed are described in [6], [8], and [16].

The extensive computation required precludes real-time speech understanding on a conventional serial computer. To improve the processing speed, the different knowledge sources can act in parallel (possibly on different portions of an utterance), and in addition, computational tasks within each knowledge source can be performed in parallel. Advances in technology have made it realistic to consider large-scale parallel processing systems [e.g., 2, 5, 13]. By designing multiprocessor knowledge sources, real-time speech understanding (with a constant delay) should be achievable. The next section briefly outlines a general configuration for a multiprocessor system for speech understanding. In the following sections, a detailed description and analysis of a parallel architecture for acoustic processing is described.

II. A Parallel Architecture For Speech Understanding

An architecture proposed to handle the speech understanding task consists of a distributed series of computation stations [3, 4]. Each computation station corresponds roughly to a speech understanding knowledge source. This distributed parallel architecture is diagramed in Fig. 1. The interconnection of knowledge sources forms a linear pipeline in which each stage is a complete multiprocessor sub-system.

A typical computation station consists of an input memory buffer *(MB),* an output MB, control units *(CUs),* and processing elements *(PEs).* The organization of the PEs within each computation station is selected to exploit whatever parallelism is inherent in the specific task being performed by that station. The processing time for each station is a function of the computational complexity of the tasks to be performed and the amount and arrival rate of input data. Assuming a maximum input rate, the processing speed requirements can be met by employing parallelism within the task algorithms and also among the tasks to be performed. Minimum processing time for the computation station will be insured when the data in the input MB is processed as soon as it is available. When a subset of PEs has finished a processing task and stored its result in the output MB, it is available to be assigned another task by the computation station's primary control unit.

Each computation station is specialized to meet performance (speed) requirements of the overall system. Processing proceeds asynchronously with respect to adjacent computation stations. When the processing time for each station is approximately equal, then no bottlenecks occur and data flow through the system will be continuous, providing real-time performance (with a constant delay). Because the parallelism within each computation station permits processing of all probable utterance hypotheses simultaneously, there is no need to backtrack once any particular hypothesis has been determined improbable. Thus, extensive parallelism is being used at each stage of the speech understanding process in order to simplify the interaction among various knowledge sources.

III. Acoustic Processing

Acoustic processing is the task of transforming periodically sampled digitized speech into characteristic time and frequency domain parameters. Acoustic processing is described in [15] and [24].

The number and type of parameters used by the major speech understanding systems vary with each system. The complete set of parameters calculated, called *characteristic parameters,* represents a segment of speech data called a *frame.* A frame can range from 5 to 20 ms in length and corresponds to a uniform section of an utterance. A frame length of 12.8 ms is used by the architecture described here. For each frame, 37 characteristic parameters are calculated. In order to achieve real-time performance, the 37 parameters must be calculated in at most 12.8 ms.

The speech data is sampled at 20 KHz. Therefore,

This material is based upon work supported by the National Science Foundation under Grants ECS-7909016 and ECS-8120896.

0190-3918/82/0000/0307$00.75 © 1982 IEEE

a 12.8 ms speech data frame contains 256 data samples. This 256 point data set is called the *short data set.* Most of the acoustic parameters are calculated from these data points. Other parameters, especially those relating to the pitch of the speaker's voice, require a longer segment of speech data containing several vocal cord oscillations. For these parameters, a 51.2 ms segment of speech data, consisting of 1024 digitized sample points, is used. This data set, called the *long data set,* includes the current 12.8 ms frame plus the preceeding 38.4 ms of speech data. Both data sets are completed and available for processing simultaneously. The parameters calculated from both data sets characterize the interval of speech of the short data set.

The 37 characteristic parameters are listed below.

A1 → A24 Linear predictive coding (LPC) coefficients. The predictor coefficients uniquely specify the transfer function of the vocal tract.

AC The normalized autocorrelation coefficient at unit sample delay. This is a rough measure of the uniformity of the data within the frame.

EH Signal energy within a High frequency band (5000 → 10000 Hz).

EL Signal energy within a Low frequency band (625 → 2500 Hz).

EM Signal energy within a Mid frequency band (2500 → 5000 Hz).

ET Signal energy within the Total frequency range (0 → 10000 Hz).

EVL Signal energy within a Very Low frequency band (0 → 625 Hz). The energy within the speech signal characterizes the overall vocal tract configuration.

ERN LPC normalized minimum error. This parameter reflects the accuracy of the linear prediction model for describing the speech frame.

F0 Fundamental frequency (pitch). The fundamental frequency indicates the oscillation rate of the speaker's vocal cords.

F1 First formant (resonant) frequency.

F2 Second formant frequency.

F3 Third formant frequency. The values of the first three formant frequencies are useful in the characterization of vowels and sonorants.

RMS Root mean square energy of the preemphasized speech signal.

ZC Zero crossing density. The zero crossings can be used to separate fricative from non-fricative speech sounds.

The algorithms required to obtain these parameters will be discussed in section V.

IV. An Architecture For Acoustic Processing

The SIMD Architecture

The architecture to perform acoustic processing within the speech understanding system is called the *Acoustic Processing Computation Station.* This is the second stage of the speech understanding system diagramed in Fig. 1. The Acoustic Processing Computation Station is diagramed in Fig. 2. It consists of a primary CU which coordinates processor activity, 4 secondary CUs, an input MB, an output MB, 512 PEs, and a multistage cube interconnection network. The

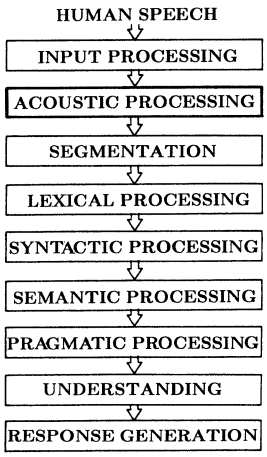


Fig. 1. A distributed speech understanding system.

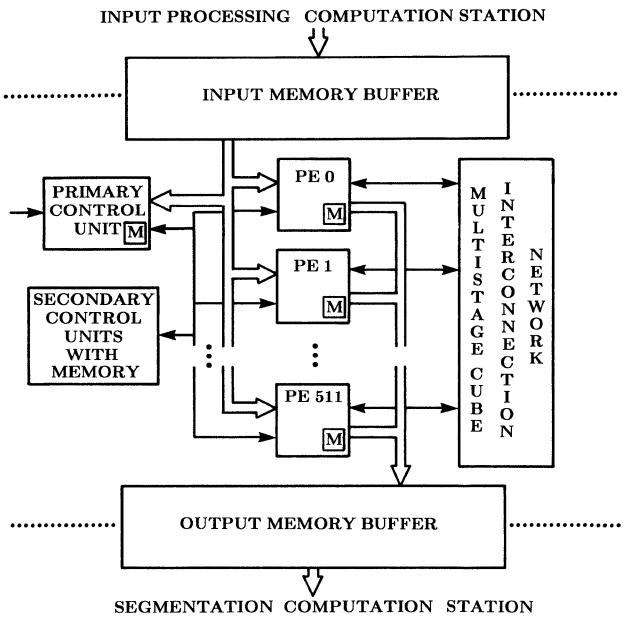


Fig. 2. The Acoustic Processing Computation Station.

computation station receives its data from the Input Processing Computation Station. It calculates the characteristic time and frequency domain parameters for frames of the input data and stores the results in the output MB. These parameters are then accessed by the Segmentation Computation Station. The components of the computation station form a multiple-SIMD *(MSIMD)* system designed to exploit the parallelism inherent in acoustic processing tasks.

The Processing Element

The MSIMD acoustic processing architecture uses 512 PEs. The PE model used is based on that presented in [7]. All control and arithmetic operations within the PE are performed by an MC68000 microprocessor. A memory management unit will arbitrate all read and write operations to the microprocessor. When the CU broadcasts an instruction to the PE, this instruction is stored within an internal instruction memory and subsequently read by the MC68000. The

CU can enable or disable the PE by utilizing masking instructions. The CU can also read various condition codes from the PE. The internal memory is used for data storage and is used only by the PE.

The MC68000 microprocessor is a powerful 16-bit device with 56 instruction types, 14 addressing modes, and eighteen 32-bit internal data and address registers [11]. Processor timing calculations were made with the microprocessor running Motorola's 68343 fast floating point software [12]. The execution times are calculated for the microprocessor running with a 12.5 MHz clock frequency. Processing times for arithmetic operations are given in Table 1.

Table 1. Processing times for the MC68000 with a 12.5 MHz clock.

OPERATION	TIME (μs)
Integer (16 bit)	
Add/Subtract	0.4
Load	0.4
Store	0.4
Floating Point (32 bit)	
Add/Subtract	14.1
Divide	48.6
Multiply	28.2
Square Root	124.2
Load	0.8
Store	0.8
Compare	1.6
Absolute Value	0.8
Negate	1.6
Complex (2 * 32 bit)	
Add/Subtract	28.2
Multiply	141.0

The Interconnection Network

Each PE is connected to all of the other PEs in the computation station by a 16-bit multistage cube interconnection network with independent box control [17]. The cube network can be partitioned into independent sub-networks of varying power of two sizes, allowing subsets of the PEs to act as independent SIMD machines. Routing through the network is established with routing tags generated by each PE. The multistage cube interconnection network was chosen because of its extremely high efficiency when performing many parallel processing algorithms. Network transfer times used are based upon the simulation studies in [1]. Network transfer times for different data types are given in Table 2. These times include the times for routing tag generation and configuration of the network based on the routing tag specification.

V. Algorithms

The calculation of the 37 characteristic acoustic parameters requires many signal processing operations. To achieve the necessary processing speed, each signal processing task was divided into parallel sub-tasks or algorithms that can be run on an SIMD machine. Nineteen parallel signal processing algorithms were used. Each of the SIMD algorithms is such that it can run on machines of different sizes, with execution time a function of the machine size. The processing time for the computation station can be adjusted by varying the number of PEs executing each SIMD sub-task.

Table 2. Network transfer times for each data type. These times include time to generate routing tags and set the network.

DATA TYPE	TIME (μs)
Integer (16 bits)	4.5
Floating Point (32 bits)	6.4
Complex (2 * 32 bits)	10.1

The nineteen parallel algorithms used by the Acoustic Processing Computation Station are listed below. The first number after each item in the list designates a reference to the calculation performed serially. Subsequent numbers indicate references to a parallel algorithm.

- Autocorrelation Calculation [15, 19, 22]
- Center Clipped Signal Construction [15, 20]
- Data Zero Padding
- Digital Inverse Filter [15, -]
- Energy Band Calculation [24, -]
- FFT [14, 21]
  - Radix 2 Decimation-in-Frequency (*DIF*)
  - Radix 2 Decimation-in-Time (*DIT*)
- Formant Frequency Analysis [9, -]
- Hamming Window [15, -]
- LPC Coefficients Calculation [10, 20]
- LPC Minimum Error Calculation [10, -]
- Maximum Calculation [-, 23]
- Minima Calculation [-, 23]
- Normalized Autocorrelation Calculation [18, -]
- Partial Autocorrelation Calculation [15, 19]
- Pitch Extraction [15, 20]
- Preemphasis [15, -]
- RMS Energy Calculation [24, -]
- Squared Magnitude Operation [15, -]
- Zero Crossing Calculation [24, -]

Several of the parameters computed by the Acoustic Processing Computation Station are obtained by combining a number of these algorithms. The 256 point Autocorrelation Calculation used to obtain the LPC coefficients is computed by combining four parallel algorithms: Data Zero Padding, Radix 2 DIF FFT, Squared Magnitude Calculation, and the Radix 2 DIT FFT. The Digital Inverse Filter used in obtaining the formant frequencies is composed of four parallel operations: Data Zero Padding, a Radix 2 DIT FFT, a Squared Magnitude Calculation, and a DBR $\rightarrow$ WRP Network Data Transfer.

Interaction points occur at the end of one algorithm and the beginning of another, when the parallel algorithms must interact to synchronize and exchange data. The specific problems which must be addressed at the interaction points are sub-system size and data allocation. The number of PEs used to execute each algorithm is determined by the real-time constraints. Therefore, the number of PEs operating on a given data set may change. In addition, the way in which data is assigned to the PEs may differ from one algorithm to the next, or the results from one algorithm may not be allocated in the pattern needed by the next algorithm. To simplify algorithm interaction, four distinct data orderings are defined for the algorithms used. For D PEs and data items d(0), d(1), ..., d(D-1):

Single Sequential Order (SSQ):

PE p contains d(p)

For D/2 PEs and data items d(0), d(1), ..., d(D-1):

Dual Bit Reversed (DBR):

PE p	contains	d(br(2 * p)
	and	d(br((2 * p) + 1))
	where	br(x) = bit reverse of x

Dual Sequential Order (DSQ):

PE p	contains	d(2 * p)
	and	d((2 * p) + 1)

Wrap Around Order (WRP):

PE p	contains	d(p)
	and	d(p + D/2)

Each of the above three data orderings can be generalized to $D/2^i$ PEs for D data items where i is an integer.

In addition to the parallel signal processing algorithms listed above, five data alignment/data transfer algorithms were designed and used by the computation station:

Load Data in DSQ Order
Distribute Data
DBR → WRP Network Data Transfer
DSQ → DBR Network Data Transfer
Network Data Broadcast

The Load Data algorithm is used for the initial assignment of data to the PEs. The Distribute Data algorithm distributes data by copying the data points from one set of PEs to another preserving the data ordering. The two Network Transfer algorithms perform reallocation of the data to obtain the specified data ordering. (For the algorithm sequences considered, no other reallocations were needed.) A Network Data Broadcast is a network data transfer in which a single data item is transferred to each PE in an SIMD machine.

VI. Operation and Performance

A different series of parallel algorithms is executed by the computation station on the short data set and on the long data set. The algorithms performed on the two data sets constitute two synchronous algorithms that are run asynchronously with respect to each other. Fig. 3 shows the assignment of principal algorithms to PEs and the algorithm processing times for one 12.8 ms segment of speech. The architecture consists of 512 PEs. PEs 0 through 255 are assigned the algorithms which are performed on the short data set. PEs 256 through 511 are assigned the algorithms which are performed on the long data set.

Assume that the short data set is available at time 0.0. At that time, the 256 data samples are loaded into PEs 0 through 256. These data samples are then transferred to the remaining PEs by using the interconnection network. The calculations on the short and long data sets proceed asynchronously from this point. The partitioning of the Acoustic Processing Computation Station's PEs to perform all of the above algorithms can be easily seen in Fig. 3.

A summary of processing times and the number of PEs utilized for each parallel algorithm are given in Table 3. When completion of an algorithm results in the generation of a characteristic parameter, that parameter is indicated after the algorithm name. The

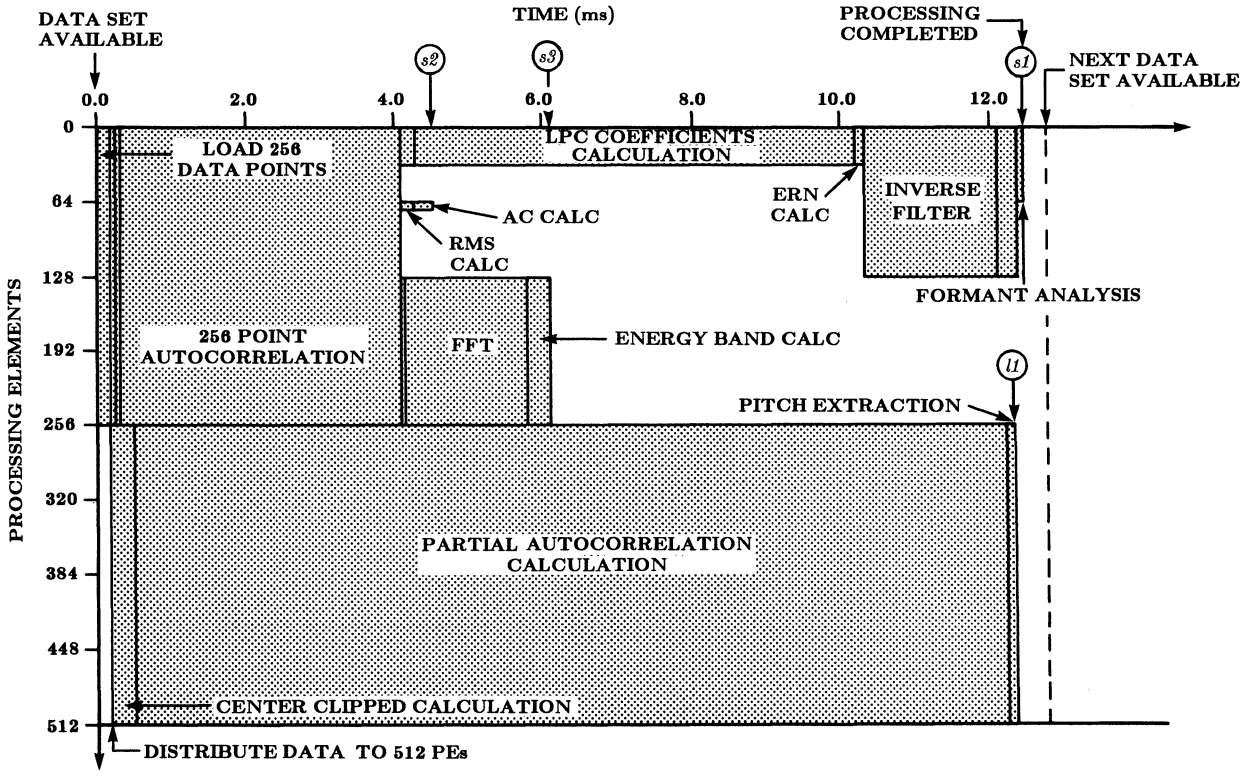


Fig. 3. Assignment of tasks to PEs and algorithm processing times for one 12.8 ms segment of speech. (Labels indicate the principal processing steps.)

Table 3. Summary of the processing times and the number of PEs used for each algorithm.

PARALLEL ALGORITHM	# PEs	TIME (ms)
Short Data Set Algorithms		
Subsequence One		
Load Data in DSQ Order	256	0.205
Distribute Data	512	0.006
Hamming Window	256	0.028
Distribute Data	256	0.006
Preemphasis	256	0.049
Autocorrelation Calc. - 256 point		
Data Zero Padding	256	0.001
Radix 2 DIF FFT	256	1.857
Squared Magnitude Calc.	256	0.085
Radix 2 DIT FFT	256	1.857
Distribute Data	128	0.013
Network Data Broadcast	32	0.160
LPC Coefficients Calc. (A1→A24)	32	5.952
LPC Minimum Error Calc. (ERN)	32	0.182
Digital Inverse Filter		
Data Zero Padding	128	0.001
Radix 2 DIT FFT	128	1.650
Squared Magnitude Calc.	128	0.085
DBR → WRP Network Transfer	128	0.013
Minima Calc.	128	0.259
Formant Frequency Analysis (F1,F2,F3)	64	0.090
	(*s1*)	12.499
Subsequence Two		
Time after Autocorrelation - 256 point		4.094
RMS Energy Calc. (RMS)	2	0.174
Norm. Autocorrelation Calc. (AC)	2	0.284
	(*s2*)	4.552
Subsequence Three		
Time after Autocorrelation - 256 point		4.094
Zero Crossing Calc. (ZC)	128	0.046
DSQ → DBR Network Transfer	128	0.013
Radix 2 DIT FFT	128	1.650
Energy Band Calc. (EH,EL,EM,ET,EVL)	128	0.327
	(*s3*)	6.130
Long Data Set Algorithms		
Load Data DSQ Order	256	0.205
Distribute Data	512	0.006
Maximum Calc.	256	0.261
Center Clipped Signal Construction	256	0.042
Partial Autocorrelation Calc.	256	11.714
Pitch Extraction (F0)	256	0.138
	(*l1*)	12.366

parallel algorithms which make up the longest synchronous path of each of the algorithm sequences are listed in order. Their processing times are tabulated and are indicated in Table 3 as *s1, s2, s3,* and *l1,* corresponding to the processing times for subsequences 1, 2, and 3 of the short data set and the subsequence on the long data set. These times are also shown on Fig. 3. The processing time for the computation station will be the slowest of the asynchronous sequences. The processing time of the computation station is 12.499 ms resulting from the processing of the short data set. Since all processing is completed before the arrival of the next speech data set ($\leq$ 12.8 ms), data flow through the architecture is continuous with no bottlenecks, providing real-time operation with a constant delay of 12.499 ms. The point at which processing is completed is indicated in Fig. 3.

VII. Discussion

This work focuses on the problem of using an MSIMD system to perform a large number of algorithms under real-time constraints. Major issues addressed include choice of partition sizes for the component algorithms, determination of the overall machine size, data allocation and alignment at the junctures between algorithms, and interconnections between the algorithms. The design resulted in a 512 processing element architecture in which data flow is continuous with no bottlenecks. Real-time performance is achieved with a constant delay of 12.499 ms.

At any point during processing, there may be from one to four independent SIMD algorithms being executed. The component SIMD machines range in size from 2 to 512 PEs. Nine different system configurations (i.e., partitionings) are used. These are accomplished dynamically, by reassignment of the control units to subsets of the PEs. A very rough measure of processor utilization can be determined by a ratio of the areas during which processors are performing algorithms and the total area of a single 12.8 ms frame. This calculation results in a processor utilization of about 75 percent.

The required processing speed to acheive real-time performance was obtained by increasing the number of PEs executing the parallel algorithms. The ability to acheive greater speed by increasing the number of processors is characteristic of the problem domain of acoustic processing. The flexibility of the MSIMD architecture presented is particularly well suited for these types of problems.

Many of the algorithms could be executed in less time than indicated in Table 3 if more PEs were used. However, this would have delayed other parallel algorithms and real-time performance may not be achieved. Other algorithms could not be run any faster because the maximum number of processors that can be employed in the algorithm are being used. For example, the LPC Coefficients Calculation algorithm can use only 32 PEs. Even though more processors are available (Fig. 3.), using more PEs will not decrease the processing time of the algorithm. The SIMD machine partition sizes were chosen in an attempt to create the smallest possible overall machine to perform all of the tasks within the real-time constraints.

The types of algorithms to be performed and the real-time constraints placed on the system design resulted in very constrained algorithm scheduling. In order to meet the real-time requirements, each algorithm must be run on the SIMD machine of the size and in the order indicated in Fig. 3. This restriction on the size of an SIMD sub-machine indicates a requirement that the operating system must acknowledge requests for an SIMD machine of a specific size. An open problem is whether or not this type of highly constrained scheduling could be done efficiently by an automatic scheduling algorithm.

Substantial speed was obtained by utilizing the small number of well defined data orderings at the interaction points between parallel algorithms. This eliminated the need for the architecture to store and

load data items between algorithms. Distribution and reallocation of data required only 0.2 percent of the total processing time of the computation station. The sequencing of the algorithms was chosen to minimize this reallocation time. In most cases, the parallel algorithms were designed such that no data alignment was necessary. For example, the 256 point Autocorrelation Calculation is performed by executing a sequence of four algorithms. The first algorithm, Data Zero Padding accepts data in SSQ order and outputs data in WRP order. The Radix 2 DIT FFT algorithm uses the WRP orderd data and outputs data in DBR order. The Squared Magnitude Calculation preserves the data in DBR order, which is then used by the Radix 2 DIT FFT. The last algorithm outputs data in WRP order. For this sequence of algorithms, no additional data allocation was needed than that provided by the algorithms in the sequence. Because of the speed increases which can be gained by avoiding frequent data reallocations, an intelligent scheduler should make use of data allocation information in sequencing the algorithms and in selecting among alternate versions of a given algorithm.

The work presented in this paper points the way to many directions for future work. Additional parallel algorithms could be explored. Since there are some idle processors during portions of the speech frame analysis, additional characteristic parameters could be calculated. The addition of floating point hardware to augment the instruction set of the MC68000 should be explored. All of the algorithms used in this work were deterministic and therefore had predictable processing times. Some signal processing operations, such as spectrum enhancement [15], have processing times that may vary depending upon the input speech data. Design of an architecture using these algorithms would require probabilistic modeling and computer simulation studies.

An issue which must be considered in the design of large scale systems such as the one presented here is the extent to which one wishes to employ special purpose hardware. Since speech processing is an evolving research area, it is desirable to have a flexible system on which new algorithms can be tested. This architecture provides such a research tool in which the amount of parallelism provided can be varied to execute a wide variety of algorithms. The design presented in this paper demonstrates the feasability of an MSIMD system to perform speech acoustic processing within real-time constraints.

VIII. References

[1] G. H. Barnes and S. F. Lundstrom, "Design and validation of a connected network for many-processor systems," *Computer,* Dec. 1981, pp. 31-41.

[2] K. E. Batcher, "The design of a massively parallel processor," *IEEE Trans. Comp.,* Vol. C-29, Sept. 1980, pp. 836-844.

[3] E. C. Bronson and L. J. Siegel, "A parallel architecture for speech understanding," *1981 IEEE Int. Conf. Acoust., Speech, Signal Processing,* Mar. 1981, pp. 1176-1179.

[4] E. C. Bronson and L. J. Siegel, "Overview of a distributed parallel architecture for speech understanding," *Proc. 15th Hawaii Int. Conf. System Sciences,* Jan. 1982, Vol. I, pp. 350-359.

[5] M. J. B. Duff, "Parallel algorithms and their influence on the specification of application problems," in *Multicomputers and Image Processing,* K. Preston and L. Uhr, eds., Academic Press, NY, 1982, pp. 261-274.

[6] D. H. Klatt, "Review of the ARPA speech understanding project," *J. Acoust. Soc. Am.,* Vol. 62, Dec. 1977, pp. 1345-1366.

[7] J. T. Kuehn, H. J. Siegel, and P. D. Hallenbeck, "Design and simulation of an MC68000-based multimicroprocessor system," *1982 Int. Conf. Parallel Processing,* Aug. 1982.

[8] W. A. Lea, Ed., *Trends in Speech Recognition,* Prentice-Hall, Englewood Cliffs, 1980.

[9] J. D. Markel, "Application of a digital inverse filter for automatic formant and F_0 analysis," *IEEE Trans. Audio Electroacoust.,* Vol. AU-21, June 1973, pp. 154-160.

[10] J. D. Markel and A. H. Gray, *Linear Prediction of Speech,* Springer-Verlag, NY, 1976.

[11] Motorola, *MC68000 16-bit Microprocessor User's Manual,* second edition, M68000UM(AD2), Jan. 1980.

[12] Motorola, *"68343 fast floating point source/object for MC68000,"* M68KFFP specification sheet, Nov. 1981.

[13] M. C. Pease, "The indirect binary n-cube microprocessor array," *IEEE Trans. Comp.,* Vol. C-26, May 1977, pp. 458-473.

[14] L. R. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing,* Prentice-Hall, Englewood Cliffs, 1978.

[15] L. R. Rabiner and R. W. Schafer, *Digital Processing of Speech Signals,* Prentice-Hall, Englewood Cliffs, 1978.

[16] D. R. Reddy, "Speech recognition by machine: a review," *Proc. IEEE,* Vol. 64, April 1976, pp. 501-531.

[17] H. J. Siegel and R. J. McMillen, "The multistage cube: a versatile interconnection network," *Computer,* Dec. 1981, pp.65-76.

[18] L. J. Siegel, "A procedure for using pattern classification techniques to obtain a voiced/unvoiced classifier," *IEEE Trans. Acoust., Speech, Signal Processing,* Vol. ASSP-27, Feb. 1979, pp. 83-89.

[19] L. J. Siegel, "Parallel algorithms for linear predictive coding," *1980 IEEE Int. Conf. Acoust., Speech, Signal Processing,* Apr. 1980, pp. 960-963.

[20] L. J. Siegel, "Using SIMD machines for speech analysis," *Proc. 14th Hawaii Int. Conf. System Sciences,* Jan. 1981, Vol. I, pp. 309-318.

[21] L. J. Siegel, P. T. Mueller, Jr., and H. J. Siegel, "FFT algorithms for SIMD machines," *17th Allerton Conf. on Communication, Control, and Computing,* Univ. of Ill., Oct. 1979, pp. 1006-1115.

[22] L. J. Siegel, H. J. Siegel, R. J. Safranek, and M. A. Yoder, "SIMD algorithms to perform linear predictive coding for speech processing applications," *1980 Int. Conf. Parallel Processing,* Aug. 1980, pp. 193-196.

[23] H. S. Stone, ed., *Introduction to Computer Architecture,* Science Research Associates, Inc., Chicago, 1975.

[24] V. W. Zue and R. M. Schwartz, "Acoustic processing and phonetic analysis," in [8], pp. 101-124.

A NOVEL APPROACH TO PARALLEL PROCESSING CRYPTOSYSTEM

Yoshiyasu TAKEFUJI, Koichiro TSUJINO, Mari IBUKI, and Hideo AISO

Department of Electrical Engineering

Keio University

3-14-1 Hiyoshi, Yokohama 223, JAPAN

Abstract -- A new cryptosystem based on multiresidue codes and on pseudorandom number generation is proposed in this paper. Parallel and pipelining computations in encryption and decryption can be realized by adopting multiresidue codes and the mixed-radix conversion scheme. The difficulty of cryptoanalysis in multiresidue codes is discussed in detail. A cipher unit for encrypting a 24-bit data block and for decrypting a 32-bit data block has been implemented by employing a low-cost microprocessor. The implementation of the unit is mentioned in this paper.

Introduction

Hiding information in secret codes has been spreading in communication systems among computers, terminals, or both of these. Since the Data Encryption Standard (DES) and several new cryptosystems have been presented recently [1][2], we have entered a cryptograph age. An ideal cryptosystem possesses the characteristics of easiness in both data encryption and decryption at low cost and those of hardness in breaking its cipher.

A new encryption system based on mixing multiresidue codes with a technique in pseudorandom number generation is presented in this paper. The proposed encryption system is a conventional cryptosystem. The cryptoanalysis is extremely exhaustive. The cryptosystem with simple data encryption and decryption has been implemented on a low cost microprocessor. The number of moduli (n) and the values of the moduli (m1,m2,...,mn) correspond to keys in the multiresidue system.

The multiresidue system has to satisfy

$$2^K \leq \mathrm{LCM}(m1,m2,\ldots,mn),$$

where k is the length of a data block to be encrypted.

The cryptosystem employing pseudorandom number generation is based on a block chaining scheme. In the block chaining scheme, the present data to be encrypted are influenced by other data previously encrypted. In the proposed system, information data in the present state to be encrypted are affected by both information of the multiresidue code in the previous state and the related pseudorandom number. The pseudorandom number is also influenced by information of the multiresidue code in the previous state.

The block chaining scheme has an inevitable drawback. Even if any single encrypted data block is dropped from the trasmission line or is not received by the decryption system, it would be very difficult or almost impossible to decrypt the succeeding encrypted data blocks.

The implemented cryptosystem employs 24-bit data block encryption and 32-bit data block decryption with six moduli. The length of one block to be encrypted can be easily expanded.

The data to be encrypted can be converted modulus by modulus in parallel. The data decryption can be pipelined adopting the mixed-radix conversion method [3].

In order to break the implemented cipher, a Marcov model of a complete graph consisting of 2^{24} nodes and $2^{24}\cdot(2^{24}-1)/2$ arcs has to be solved. Moreover, the difficulty of the cryptoanalysis can be enhanced by increasing the period of the pseudorandom number sequence.

The proposed encryption and decryption procedures are described in Fig.1 and Fig.2 , respectively.

Conversion from normal numbers to multiresidue codes

The multiresidue system is composed of multiple moduli m1,m2,...,mn which give the usable number range M=LCM(m1,m2,...,mn). Let a normal number X be represented in a residue form. The residue representation of $|X|_M$ is then,

$$|X|_M <==> \{x1,x2,\ldots,xn\},$$

where $xi=|X|_{mi}$ means that xi is the ith residue of X modulo mi.

* If $0<a<m$ and $|ab|_m=1$ are satisfied, a is called the multiplicative inverse of b mod m, and is denoted by $a=|1/b|_m$.

Example 1 : For the moduli m1=6,m2=7,m3=11, the usable number range is then

$$0 \leq X < M = \mathrm{LCM}(6,7,11) = 462.$$

When X is 26, the multiresidue representation of $|X|_M$ is

$$|26|_{462} <==> \{2,5,4\}.$$

A high speed parallel residue computation algorithm is proposed in this paper. When $2^K\pm1$ (k=1,2,...) are adopted as moduli, every residue is able to be calculated in parallel as shown in Fig.3. The residues of an n-bit data block mod 2^K-1 can be parallely calculated on every k-bit block. This can be proved using Eq.(1) [3] as follows:

$$\left|a+b+c+\ldots.\right|_M = \left|\,|a|_M+|b|_M+|c|_M+\ldots.\right|_M \qquad (1)$$

Proof : Consider X mod M. Let X be an n-bit data and M be 2^K-1 (k=1,2,....).

$M=2^K-1=0 \bmod M$

$2^K=1 \bmod M$

$(2^K)^i=1^i=1 \bmod M,\ i=1,2,\ldots.$

$$|X|_M=\left|\,|a_0+2a_1+\ldots.+2^{K-1}a_{K-1}|_M+|2^K|_M\cdot|a_K+2a_{K+1}+\ldots.+2^{K+1}a_{2K-1}|_M+|2^{2K}|_M\cdot|a_{2K}+2a_{2K+1}+\ldots.+2^{3K-1}a_{3K-1}|_M+\ldots.\right|_M$$

Q.E.D.

The residues of an n-bit data block mod 2^K+1 can be calculated on every 2k-bit block in the same manner.

The proposed parallel computation algorithm is suitable not only for multimicroprocessor implementation, but also for iterative VLSI implementation.

Conversion to the Mixed-Radix system

There are two schemes of conversion from the residue system to the normal number system. The one is based on the Chinese Remainder Thorem, while the other the Mixed-Radix Conversion [3]. We have adopted the latter conversion from the viewpoints of the parallelism and the pipeline processing capability involved.

The mixed-radix expression is of the form

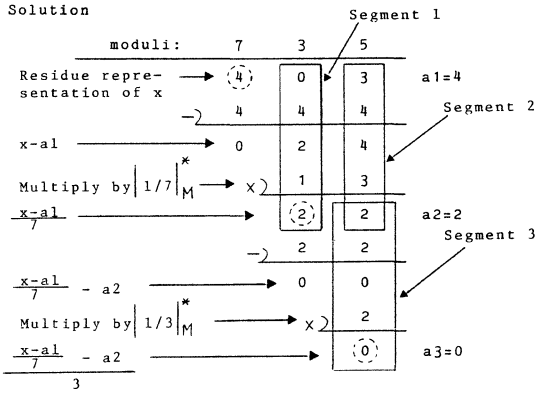
$$S=an\prod_{i=1}^{n-1} mi+\ldots.+a3\cdot m2\cdot m1+a2\cdot m1+a1, \qquad (2)$$

where ai is the ith mixed-radix coefficient. The ai (i=1,2,....,n) are required to obtain the normal number representation as shown in Example 2 .

Example 2 : For m1=7, m2=3, and m3=5, find the associated mixed-radix digits of {4,0,3}, where the mixed radix expression is

$$X=a3\cdot(7\cdot3)+a2\cdot(7)+a1. \qquad (3)$$

Solution



Then the mixed radix representation of X is {0,2,4}. Hence by Eq.(3), one obtains

$$x=0\cdot(7\cdot3)+2\cdot(7)+4=18.$$

0190-3918/82/0000/0313$00.75 © 1982 IEEE

Parallelism and pipelining

Assuming that t processors are employed for converting the normal data to t residues, the ith residue of X mod mi is computed by the ith processor, where X is the normal data and mi is the ith modulus. The maximum throughput of the multiprocessor system depends on the slowest residue computation. The experimental encryption program, which was designed to compute each of the six residues in sequential on a single processor,can be divided into the six independent program modules. If the modules are processed in parallel on the six processors, approximately. five times of the throughtput will be expected to be achieved with some of the waiting overhead for synchronization among the six processors.

On the other hand, the data decryption adopting the mixed-radix conversion scheme can be mapped onto the parallel and pipelined multiprocessor architecture as shown in Fig.4. Fig.4 describes a three-residue decryption system of Example 2 in the previous section. Processor P1 is a queue which transmits the residues x1, x2, and x3. Processor P2 performs the calculation of the segment 1 to obtain the mixed-radix coefficient a2 shown in the solution of Example 2. Similarly, P3 and P4 are for the segment 2 and for the segment 3, respectively. Processor Q performs the conversion from the mixed-radix system to the normal number system following Eq.(3).

If each queue of every processor provides an adequate length, the system requires no centralized synchronization at all, because the scheme is based on a data flow mechanism. When all the required data arrive at queues of a processor, the computation is fired and is autonomously performed in the processor to provide the computed result for the succeeding processors. In order to convert t residues into the normal data, N processors need to be given in the system, where N is $t(t-1)/2+2$. Since in the decryption system the processor Q shown in Fig.4 for the conversion from the mixed-radix system to the normal number system has the heaviest load in the computation, the unit time of the pipelined processing is determined by the computability of the processor Q.

The experimental decryption program for sequentially converting the six residues to the normal data can be divided into the seventeen independent program modules (t=6, N=17). If the modules are processed in parallel on seventeen processors, it could be estimated by the experimental decryption program that the unit time diminishes to approximately one twelfth of the time which was required for decryption on a single processor.

Pseudorandom number

The pseudorandom number genarated by a linear recurrence modulo 2 "shift register" [1] is utilized as a key of the Caesar Cipher [1] in the proposed system.

When a trinomial of the form x^p+x^q+1 whose degree is a Mersenne exponent is adopted as a generator polynomial, the period of the linear recurring sequence becomes 2^p-1 [1]. On the other hand, the period of a primitive polynomial becomes 2^p-1, where p is the degree of the polynomial [1]. Combinations of various generator polynomials can be chosen to generate pseudorandom numbers.

Let a polynomial H(x) be $G1(x)^{m_1}G2(x)^{m_2}\ldots Gt(x)^{m_t}$. The period of the linear recurring sequence becomes $LCM(n1,n2,\ldots,nt)\cdot 2^{i+1}$, where ni (i=1,2,....,t) are the period of the Gi(x) and the 2^{i+1}has to be satisfied as follows [4]:

$$2^i < \mathrm{Max}(m1, m2, \ldots, mt) \leq 2^{i+1}.$$

Example 3 : Find the period of $H(x)=(x^3+x+1)^2(x+1)^3$.

Solution : $G1(x)=x^3+x+1$, $G2(x)=x+1$, $m1=2$, $m2=3$, $n1=2^3-1=7$, $n2=2^1-1=1$

Hence, the period of the H(x) becomes

$$LCM(7,1)\cdot 2^{i+1}=7\cdot 4=28\cdot 2^{i+1}$$
$$\{2^{i+1} \mid 2^i < \mathrm{Max}(2,3) \leq 2^{i+1}\}.$$

The length of the generated pseudorandom number sequence corresponds to the key length of the Caesar Cipher. The number of shifted clocks in a state is influenced by multiresidue codes in the previous state. If a trinomial of the form x^p+x^q+1 whose degree is a Mersenne exponent is chosen as a generator polynomial, the generator requires a p-bit initial seed (not all zero) and the total number of the seeds becomes 2^p-1 [1].

Cipher unit

A cipher unit for realizing the proposed encryption and decryption has been implemented by the use of a microprocessor Z-80A as shown in Fig.5. RS232C, HDLC, and SDLC interfaces are realized in the unit employing a SIO chip for the Z-80 family. The encryption time and the decryption time are 1 ms and 2 ms, respectively. The encryption and the decryption programs are stored in a 2k-byte ROM. The number of moduli and the values of the moduli can be changed by manipulating DIP switches or by replacing the ROM with another one. The cost of the experimental unit was approximately 70 dollars.

Strength of multiresidue codes without pseudorandom number generation

It is assumed in this section that abundant encrypted data blocks are sampled by a cryptoanalysist and that the length of a data block is known. Consider the number of required blocks to be sampled. In order to cryptoanalyze the multiresidue code, the property of the inclination in the probability of the occurrence of 1's in the bit sequence of a residue can be utilized.

When the modulus m is even, the probability of the occurrence of 1's in a residue is estimated to be 1/2. When the modulus m is odd, the probability p_i of the occurrence of 1's in the ith bit of a residue is estimated to be r/m, where r is a positive integer. The p_i is satisfied as follows:

$$(m-2^K)/m=pk< \ldots. <pi< \ldots. <p1=p0=(m-1)/2m \qquad (4)$$

where p0 is the probability of the occurrence of 1's in the least significant bit of the residue and pk is that in the most significant bit. The equation $m=1/(1-2p0)=1/(1-2p1)$ is satisfied by Eq.(4).

Let $\widehat{p0}$ be the statistical probability of p0. In order to investigate whether an odd modulus m is employed or not, it is sufficient to examin whether the inequality $m-1 < 1/(1-2\widehat{p0}) < m+1$ is satisfied or not . Therefore,

$$1/2-1/2(m-1) < \widehat{p0} < 1/2-1/2(m+1),$$

$$1/2-p0-1/2(m-1) < \widehat{p0}-p0 < 1/2-p0-1/2(m+1),$$

$$1/2m-1/2(m-1) < \widehat{p0}-p0 < 1/2m-1/2(m+1),$$

$$-1/2m(m-1) < \widehat{p0}-p0 < 1/2m(m+1),$$

$$\widehat{p0}\sim p0 \cong N(\Phi, (m^2-1)/4nm^2),$$

where N means the normal distribution and n is the number of required blocks to be sampled.

In order to estimate modulus m with the reliability of 99%,

$$(3/2)(1/\sqrt{n})\sqrt{(m^2-1)/m^2} < 1/2m(m+1),$$

$$9m^2(m+1)^2(m^2-1)/m^2 < n,$$

$$9(m+1)^3(m-1) < 9(me+1)^4 < n$$

should be satisfied, where me is the estimated maximum modulus. For example, me=127 then $n \cong 2^{31}$. When the modulus is even, $9(me+1)^2 < n$ is introduced in the same manner.

The strength of multiresidue codes without pseudorandom number generation is determined by the number of required blocks to be sampled. The number depends upon the maximum modulus. The larger modulus is chosen, the longer block to be encrypted is needed.

Conclusion

A new cryptosystem based on mixing multiresidue codes with a technique in pseudorandom number generation is proposed. The cryptosystem has been implemented in the low-cost cipher unit using a microprocessor. If the characteristics of parallelism and pipelining involved in the multiresidue system are mapped onto multimicroprocessor systems or onto iterative VLSI systems, encryption and decryption of much higher speed could be achieved. It is investigated that the difficulty of cryptoanalysis of multiresidue codes depends upon the maximum modulus. It is expected that the proposed low-cost cryptosystem will contribute to spreading communication with hiding information in secret codes.

References

[1] Adel J. Goldberg, " Special Issue: Cryptology, " ACM Computing Surveys, vol.11, no.4, (December, 1979).

[2] Masataka Kato, " Series: Fundamental Cryptology, " Mathematical Sciences, Science Press, no.178, (April, 1978) - no.224, (February, 1982).

[3] Nicholas S. Szabo, Richard Tanaka, " Residue Arithmetic and its Applications to Computer Technology, " McGraw Hill, (1967).

[4] Y. Miyazawa, " Coding Theory, " Shokodo Press, (1973).

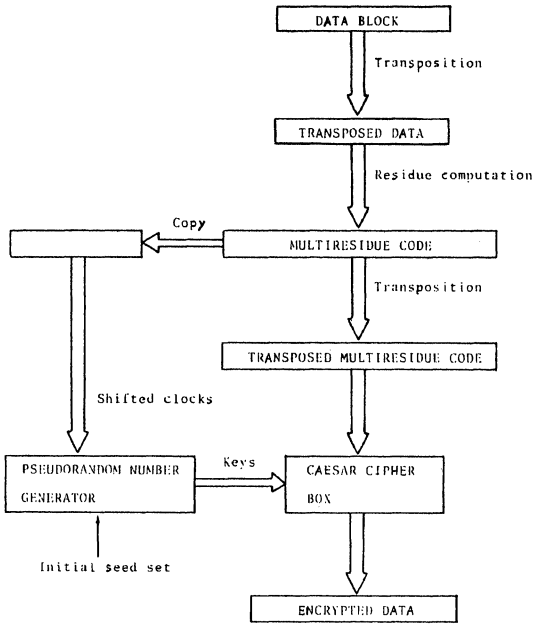


Fig.1 Encryption procedure

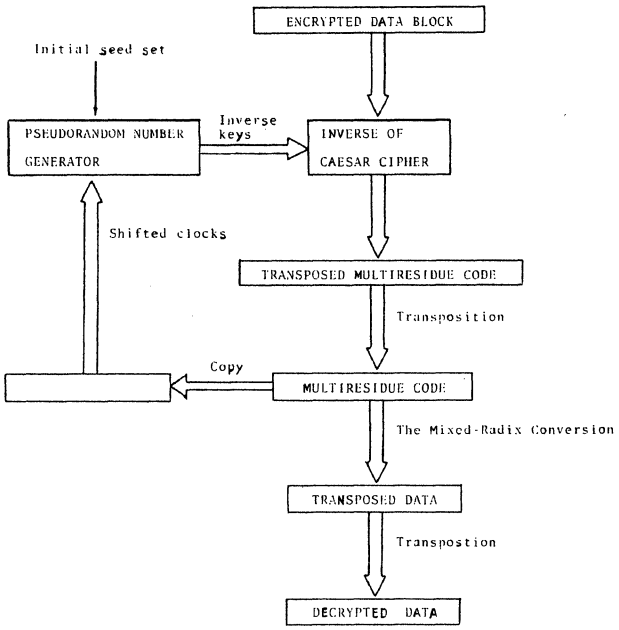


Fig.2 Decryption procedure

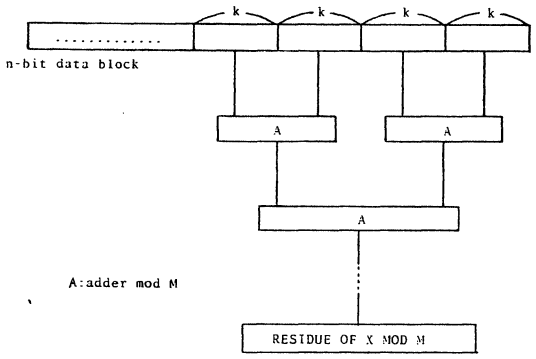


Fig.3 Parallel computation of a residue of X mod M, where M is the form of $2^k - 1$

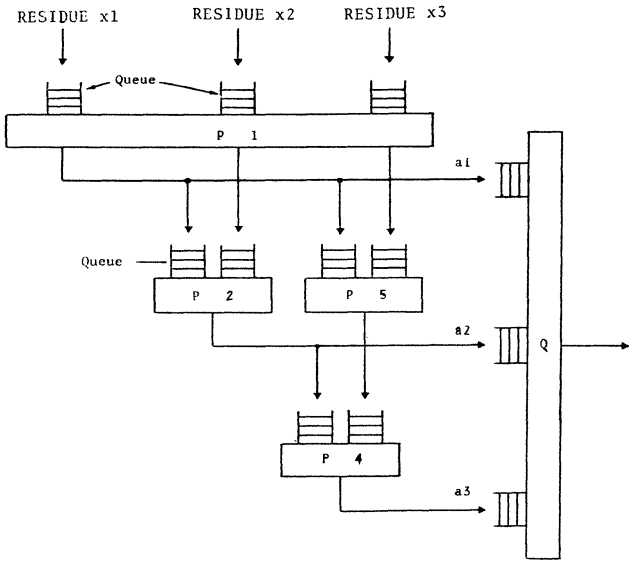


Fig.4 Three-residue decryption system

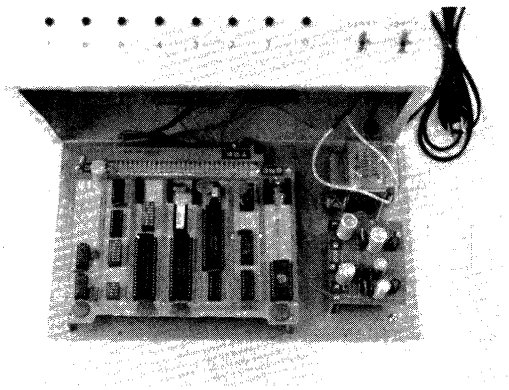


Fig.5 Cipher unit

A PARALLEL/PIPELINE PROCESSOR FOR FAST EXPONENTIATION

Bahaa W. Fam
The MITRE Corporation
Bedford, Massachusetts 01730

ABSTRACT

This paper presents a generalized architecture for a parallel/pipeline processor capable of performing exponentiation (raising some base α to a power x in a finite field) in $O(\log_2(\log_2 x))$ time. This device has applications in coding and public key data encryption.

INTRODUCTION

In this paper we present a highly efficient parallel processor architecture for finite field exponentiation.

Knuth gave a good technique for exponentiation with a single processor architecture having time complexity of $2(\log_2 x)t$ where t is the time delay of multiplying (or multiplying/reducing in a finite field) two N-bit numbers [3].

Knuth's algorithm was applied to a two processor device and the complexity of the procedure was subsequently reduced to $(\log_2 x)t$ shown to be optimal for any parallel architecture [1]. This bound was based on the number of squaring operations required in the worst case. We will consider a parallel architecture which performs finite field exponentiation in $O(\log_2(\log_2 x))t$ time.

This device has applications to coding and public key data encryption. In some cryptographic systems the generation of some public key (Y) from some secret key (x) consists of raising some known base α to the power x in some $GF(2^N)$ [2], [4].

MATHEMATICAL BASIS OF EXPONENTIATION

Assume we are performing exponentiation in $GF(2^N)$.

We can say that x, the desired exponent can be represented by some N-bit vector, $\bar{b}$.

$$\bar{b} = (b_{N-1}, b_{N-2}, \ldots, b_0)$$

Knowing that $\alpha^{y+z} = (\alpha^y)(\alpha^z)$ we can express α^x as $\alpha^x = \prod_{i=0}^{N-1} \alpha^{2^i b_i}$

If the powers of α^{2^i} $(i = 0,1, \ldots, N-1)$ could be provided to a number of multiplication units operating in parallel (multiplication being commuative in Galois Fields) the time required to produce the desired power of the base α could be greatly reduced.

ARCHITECTURE CONCEPT

The structure of processor to be discussed is based on the concept presented above.

The processor architecture is topologically similar to that of a binary tree. A single multiplication element in our processor would correspond to a node in the binary tree while a line (parallel/serial) for the undirectional transfer of information within the processor would in turn correspond to an arc in the tree .

Now consider the procedure by which such a machine might compute the product of 2^n numbers $(m_1, m_2, \ldots, m_{2^n})$. The set of numbers would first be partitioned into pairs and assigned among the 2^{n-1} multiplication elements (ME) at level n.

The processors at level n would each multiply the two numbers assigned to them and pass the product to their "father" at level n-1. The "fathers" at level n-1 would then multiply the values in their registers and pass the product to their "fathers" an so on. The process would continue until the root processor had the value

$\prod_{i=1}^{2^{n-1}} m_i$ in one register and $\prod_{i=2^{n-1}+1}^{2^n}$

in the other. Multiplying the contents of its two registers it:

would produce $\prod_{i=1}^{2^n} m_i$, the desired result.

If we had half the number of multiplication elements (an n-1 level architecture) we could partition the 2^n numbers by assigning two pairs of values to each multiplication element on the lowest level.

Each bottom level multiplication element would proceed by multiplying its first pair of numbers and then its second pair which would pipeline the products of $(m_1, \ldots, m_{2^{n-1}})$ through to the root first followed by the products of $(m_{2^{n-1}+1}, \ldots, m_{2^n})$. The top most element (an accumulating

0190-3918/82/0000/0316$00.75 © 1982 IEEE

multiplication element above the root of the tree structure) would then store the value of the product of the first 2^{n-1} numbers until it receives the product of the second 2^{n-1} numbers. It would then multiply the two to produce the desired product. The delay in achieving the result in this design would be that of n+1 multiplications as opposed to a delay of n multiplications in the example with n levels. Clearly the processor could have any number of levels with the number of levels being inversely proportional to the time delay in achieving the product.

To compute α^x we would, rather than multiplying 2^n arbitrary numbers as in our example, multiply the precomputed powers of α^{2^i} selectively based on the binary representation of the exponent x to obtain α^x.

A device was recently developed which produces all necessary powers of α^{2^i} in time (1)t [5]. In combination the two devices can produce any α^x in GF(2^N) very efficiently.

GENERAL ARCHITECTURE AND ALGORITHM

In this section we will present the design of a general (J level) processor for the exponentiation problem under consideration as well as the formal version of the processing algorithm. It is important to note that the majority of the decision making specified in the algorithm will be implemented through the hardware in the the multiplication elements. Some branching that is specified in the algorithm, that which requires that certain steps of the algorithm be skipped over in the very early and very late iterations, will be controlled by a comparator/counter which will control clock inputs to the various levels of the structure.

In an efficient implementation, x would probably be placed in a shift register with appropriate connections to the 2^{J-1} multiplication elements on level J and shifted 2^J bits at each iteration providing the control for that wave of computation.

There are three distinct types of multiplication elements in this architecture. All have a multiplication/reduction unit (a device capable of multiplying two N-bit numbers and reducing the product in GF(2^N) in time delay t.) They differ only in their decision logic.

Prior to the computation of α^x the powers of α would be distributed two to each ME along the Jth level until each ME had two, while these are being multiplied the next set of pairs would distributed. The process would continue until all N of the α^{2^i} had been distributed.

The multiplication elements at level J have logic (hardware) to perform the decision operations specified in the algorithm. The binary representation of x can be considered to be a control element for these devices. These elements also have one register containing the value 1.

The multiplication elements on levels 1 through J-1 have a multiplication/reduction unit and two input registers A and B.

The multiplication element used to compute the accumulated partial products and ultimately produce the result has a logical feedback from its output to its B register. It outputs the result only when the counter/comparator so directs it. Below is the general algorithm for computing α^x in GF(2^N).

Let us consider the general structure of the processor.

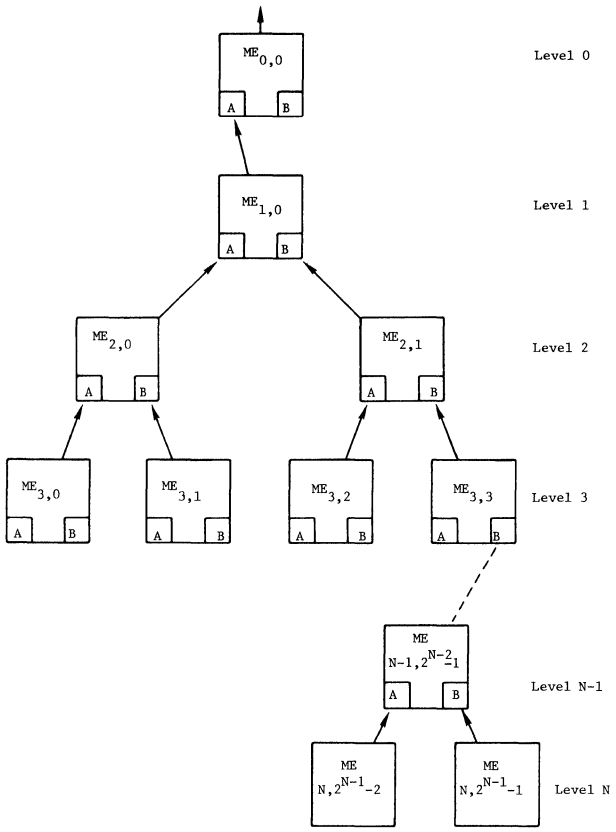


Figure 1

ALGORITHM

0 I=0

1 a) Level J Multiplication Elements ($ME_{J,i}$) for i=0,1,2, . . . ,$2^{J-1}-1$

If ($b_{2i+(I)2^J}$, $b_{2i+1+(I)2^J}$) =

(0,0) Move the value 1 from the register to $ME_{J-1, \lfloor i/2 \rfloor}$

(1,0) Move the value $\alpha^{2^{2i+(I)2^J}}$ to $ME_{J-1, \lfloor i/2 \rfloor}$

(0,1) Move the value $\alpha^{2^{2i+1+(I)2^J}}$ to $ME_{J-1, \lfloor i/2 \rfloor}$

(1,1) Move the product of

$\alpha^{2^{2i+(I)2^J}}$, $\alpha^{2^{2i+1+(I)2^J}}$

to $ME_{J-1, \lfloor i/2 \rfloor}$

b_{J-1}) (level J-1 ME) $(ME_{J-1,i})$ for $i=0, \ldots, 2^{J-2}-1$

If $I \geq 1$ multiply contents of A register and B register and move result to $ME_{J-2, \lfloor i/2 \rfloor}$

b_{J-2}) (level J-2 ME) $(ME_{J-2,i})$ for $i=0, \ldots, 2^{J-3}-1$

If $I \geq 2$ multiply contents of A register and B register and move result to $ME_{J-3, \lfloor i/2 \rfloor}$

●
●
●

b_1) (level 1 ME) $(ME_{1,1})$

If $I \geq J-1$ multiply contents of A register and B register and move result to $ME_{0,0}$

c) Level 0 ME $(ME_{0,0})$

If I=J move contents of A register to B register.

If $J+1 \leq I < \lceil N/2^J \rceil + J+1$ multiply A register by B register and store product in B register.

If $I = \lceil N/2^J \rceil + J+1$ output product and STOP.

2 $I \longrightarrow I+1$

If $I < \lceil N/2^J \rceil$ go to 1 a)

If $\lceil N/2^J \rceil \leq I < \lceil N/2^J \rceil + J$ go to 1 $b_{J-(I-\lceil N/2^J \rceil)}$)

If $I = \lceil N/2^J \rceil + J$ go to 1 c)

ANALYSIS OF COMPLEXITY

It was shown earlier that the number of levels in the processing stucture influences the speed of the exponentiation procedure.

First we will examine the number of multiplication elements in a given architecture of J levels. By definition of the topology, each level (k) of the processor has twice as many multiplication elements as the previous level(k-1). Hence the first level(k=1) has one ME the second has two, the Kth 2^{K-1}. There are J levels plus one accumulating element thus a total of: $(2^J-1)+1=2^J$ multiplication elements in the design.

The time complexity of a J-level device can be calculated as follows:

There are 2^{J-1} multiplication elements on level J. If we wish to compute α^x in $GF(2^N)$ and x is represented by an N-bit vector then there are at most N values of α^{2^i} to be multiplied

If there are 2^{J-1} processors on level J then each multiplies $\lceil N/2^{J-1} \rceil$ values. Thus there are $\lceil N/2^J \rceil$ iteration levels in the procedure hence a time delay of $\lceil N/2^J \rceil t$

We note that partial products are being pipelined through the system to the accumulating ME at the top level (level zero). In order to output the result the accumulating ME must receive the product of the last wave of values. They must pass through J levels (level J-1 through level 1) before reaching the accumulating ME. This requires an additional time of J t. When $ME_{0,0}$ receives the product it must multiply it by the partial product in its B resister and output the result taking time t, thus the total time delay of the procedure is.

$$\left(\lceil N/2^J \rceil + J + 1 \right) t$$

We stated previously that such a design could yield a complexity of $(\log_2(\log_2 x))$. This is true if $2^J = N$. This is the lower bound on the computation of α^x in $GF(2^N)$ using this device. In this case no accumulating ME is needed saving one delay.

References

[1] A. Borodin, The Computational Complexity of Algebraic and Numeric Problems, American Elsevier, New York, 1975.

[2] W. Diffie and M. E. Hellman, "New Directions of Cryptography", IEEE Transactions on Information Theory, Vol IT-22, No. 6, pp 644-654, November 1976.

[3] D. Knuth, Seminumerical Algorithms, (Second Ed.), Addison-Wesley, Menlo Park, 1981, Pg 444

[4] B. Schanning, S. Powers, J. Kowalchuk, "MEMO: Privacy and Authentication for the Automated Office", Proceedings of the Fifth Conference on Local Computer Networks, Oct 6-7, 1980.

[5] B. Fam, J. Kowalchuk, "A VLSI Device for Fast Exponentiation in Finite Fields", M82-31; The MITRE Corporation, June 1982, ppl-27.

ISLAND UNIVERSES:
DISTRIBUTING A SINGLE-USER OPERATING SYSTEM

Victor P. Holmes, Bruce N. Malm, and Tom H. Little
Departments of Computer Science, and Electrical/Computer Engineering
New Mexico State University
Las Cruces, New Mexico 88003

Abstract -- A fully decentralized operating system is specified for a single user multiple computer environment. Most operating systems have been designed around the architecture of a specific machine. We propose to design the operating system in a modular fashion specifying the needed hardware architecture as the operating system evolves. Each module is assigned to its own processing element and these communicate when necessary via a message passing scheme. Process swapping is not necessary since multi-processing does not take place on the computing element level. Although connected to a local network most work is done at the local user station and there is no dynamic load balancing at the network level. A distributed design leads to small and simple operating system modules. By distributing functions to independent processors protection is greatly simplified and the inherent concurrency gained improves performance.

INTRODUCTION

The decreasing size and cost of integrated circuitry suggests a new direction in the development of computer systems. It is quite reasonable to expect the current state of multiple users per processor to be totally reversed: one user will have an array of processors at his disposal. This paper specifies a multiple processor architecture and its accompanying distributed operating system for a single user environment.

To date operating systems have been designed around the architecture of a specific machine. In contrast, we propose to first design the operating system in a modular fashion. The overall architecture of the desired machine will then evolve as the design of the operating system evolves and will be specified to meet its needs. This proposal is based on the everpresent reality that hardware is no longer an expensive resource, and need not place the traditional constraints on system design. The phrase "Island Universe" is used to suggest the extensive processing power available to an isolated, single user under this design.

Network Levels

We view network activity on three levels. On the large scale, there is a remote network which links geographically distant sites, providing potentially global communication and specialized services. The next level, the local network, is of the Ethernet variety (1), and operates in the niche traditionally occupied by a centralized, time-sharing system. On this level are a variety of basically independent users operating out of their own stations, with occasional cooperation on specific tasks. In general, we feel a user should complete work based on resources at the user's station and not arbitrarily send work out to other stations. This prevents local performance degradation due to the load of others, as well as providing a measure of protection between users. However, we still recognize isolated instances of network use beyond that of a mail service, but the local resources of an individual user are considered sacrosanct. Central to the design of the "Island Universe" environment is the third level of network, the user station itself. The components of the distributed operating system constitute a miniature network of cooperating processes within the user's station and is the topic of this paper.

THE DISTRIBUTED OPERATING SYSTEM

We propose to partition the operating system into cooperating modules, similar in concept to task force utilities in Medusa (2), each with specific functions mapped onto physically separate hardware components. This greatly simplifies the overall complexity and protection needs. Each component has a small resident nucleus and a software process to perform the implied operating system function. A major function of each nucleus is concerned with message passing and is therefore reminiscent of other nuclei (3,4), but is even less complex than these examples. A diagram of the interconnections between these components is shown in Figure 1. Each hardware component consists primarily of a processor and memory to hold its assigned software process. Some modules have a natural association with certain devices such as the User Interface with the display terminal and the File System Manager with secondary storage. Each module is partitioned in such a way as to make its task quite simple. We propose to eliminate many of the complexities brought on by resource sharing.

Software Modules

There are two types of software modules, system and user processes. System processes are statically assigned to the User Interface(UI), Task Scheduler(TS), File System Manager(FS), Device Controller(DC), External Communication Controller(ECC) and the Inter-process Communication Manager(IPC). These processes are resident at all times and because of their limited task are limited in size. On the other hand, user processes are run in the Processor Array(PA), a collection of identical processors. Because of the inherent unpredictability and varying needs of user processes, these processors need more memory and perhaps more speed. All processes cooperate on common tasks via a message passing system. Messages between system processes or between system and user processes take place on the Service Bus while those between user processes take place on the Inter-process Communication Bus. The function of each

0190-3918/82/0000/0319$00.75 © 1982 IEEE

system process will now be briefly described.

The UI handles all interaction with the user and essentially acts as a command line interpreter and, possibly, an editor. The DC manages all use of devices not associated with the user display terminal or the on-line file system and caters to specific device idiosynchrasies. The ECC performs all external networking functions for the station and, thus, is concerned with protocol as well as security matters. The FS manages access to the file system and maintains a local file cache in primary memory as well as the file system itself on secondary storage. The TS and the IPC both manage and support the needs of user processes running in the PA. The TS acts as a user process manager while the IPC manages the communication. Each process is assigned dynamically to a processor in the array and is allowed to block as well as run to completion without swapping. A "process cache" is maintained to re-use processes already in the PA without having to reload them.

Communication takes place between these modules in two ways, messages and data. As previously mentioned, requests for system activity takes place in the form of messages on the Service bus. Similarly, communication needs between user processes take place as messages, but on the IPC bus. Messages are short transmissions, typically of fixed length, occurring quite often. Data communication occurs less often but involves much longer transmissions. For example, this might be the loading of a user program to a processor in the PA from the file system. While this form occurs less often, we would not want to dominate the Service bus for the considerable amount of time it would require. Hence, this creates the need for the separate Data bus. Although these connections are described as buses, we do not rule out the presence of dedicated links for high density traffic.

ADVANTAGES

The decreasing cost of processor and memory resources has made possible experimentation in the distribution of operating system functions. We believe that division of the operating system into distinct physical subsystems offers many advantages in terms of simplicity, efficiency, protection, and security. Of course, we also expect an improvement in performance due to the concurrency in such a system.

From the perspective of the working environment implied by this architecture, there are several obvious advantages. The isolation of the user from centralized control increases both responsiveness and security. The paradigm of a local network of autonomous stations (with communications capability) more accurately reflects the work habits of most computer users than does that of a centralized time-sharing system.

Also, the system has many advantages stemming from its internal organization. The operating system and user processes are all totally distributed, allowing significantly faster response (resulting from the parallelism), and additional security (resulting from autonomous nature of the individual processors).

Many traditional problems of operating system design disappear in this architecture. There is no need for memory management, CPU scheduling (in the timeslice sense), or inter-user security. The operating system itself is also inherently secure from user intrusions. The modularization realized by the separation of the components simplifies the programming of the individual functions. Evidence of this is seen in many examples of multi-users time sharing systems where multi-layered tables are needed to implement sharing and allow processes to be swapped out.

The message-based nature of the system yields a measure of protection. Furthermore it is a simpler problem because it is defensive in nature and, since each computing module has only a nucleus and one process running, there is little need for extensive hardware protection mechanisms.

CONCLUSION

Implicit in our design is the assumption of the availability of inexpensive resources. The number of processing elements may seem overwhelming at first glance, but advances in VLSI technology will allow fabrication of systems in a fraction of the space of current centralized computing systems in the very near future. Current technology allows line widths of 1 micrometer. Use of X-rays will make line widths of .1 micrometers possible (5). These improvements point to densities that will allow several times the number of devices per chip than are now possible. The modules of our proposed system might well be put on just a few chips, and this will reduce cost and improve reliability. In particular, the processor array appears to be appropriate for high density fabrication techniques. In the next few years, micro-computers as complex as the PDP-11/34, complete with processor, memory, and I/O interfaces will be available on a single chip (6). The recently announced multimicroprocessor chip, Texas Instruments' RIC (7) merely reinforces the practicality of this view for the future.

One of the major goals of this work is to provide an environment which supports execution of true concurrent algorithms in the system's Processing Array. Furthermore, it is important that this capability be designed into the system from the beginning. The operating system should provide a set of tools which aid in the specification or parallel programs and it is our intent to do so.

Currently, work is in progress to model this system. Weaknesses can be observed, in this manner, to aid in the final specification of the design. Of particular interest is usage pattern of the communication paths. We hope to begin the development of a prototype system within the next year. Only through the construction of such a system can we witness the benefits in terms of complexity of system design, performance and protection.

(1) R.M. Metcalfe, and D.R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks", Comm. of the ACM, (July, 1976), pp. 395-404.

(2) J.K. Ousterhout, D.A. Scelza, and P.S. Sindhu, "Medusa: An Experiment in Distributed Operating System Structure", Comm. of the ACM, (Feb., 1980), pp. 92-105.

(3) P.B. Hansen, "The Nucleus of a Multiprogramming System", Comm. of the ACM, (April, 1970), pp. 238-250.

(4) J. Hoppe, "A Simple Nucleus Written in Modula-2: A Case Study", Software-Practice and Experience, vol. 10, 1980, pp.697-706.

(5) B. Fay, et. al., "X-Ray Replication of Masks Using the Synchrotron Radiation Produced by the ACO Storage Ring", App. Phys. Lett., (September, 1976), pp. 370-372.

(6) L. Wittie, et. al., "MICROS, A Distributed Operating System for MICRONET, A Reconfigurable Network Computer", IEEE Trans. on Computers, (December, 1980), pp. 1133-1144.

(7) R. Budzinski, J. Linn, and S. Thatte, "A Restructurable Integrated Circuit for Implementing Programmable Digital Systems", Computer, (March, 1982), pp. 43-54.

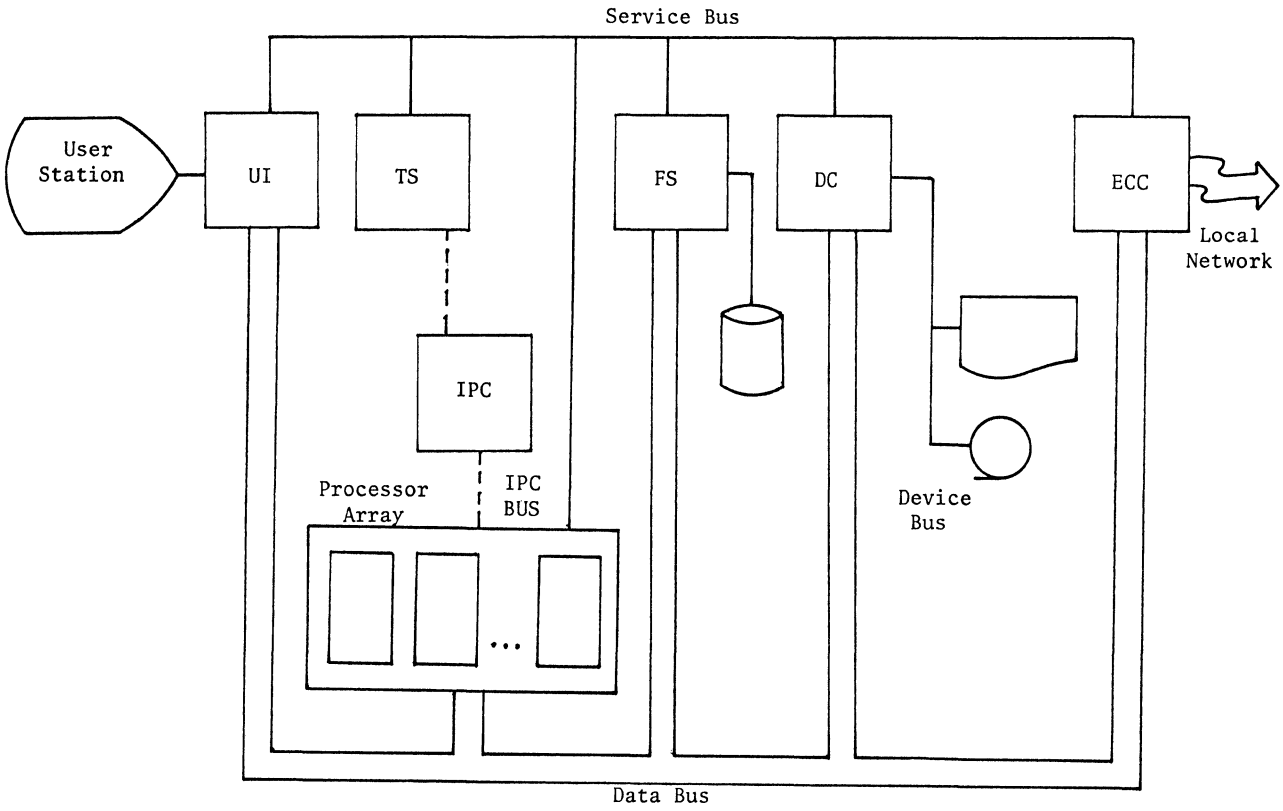


Figure 1 - Single-user distributed OS: Subsystems and Interconnections

A VARIED STRATEGY PROGRAMMABLE ARBITER

M. COURVOISIER
L.A.A.S.-C.N.R.S.
Université Paul Sabatier
7, avenue du Colonel Roche - 31400 TOULOUSE - France

ABSTRACT

The use of arbiters can be very efficient in shared bus multimicroprocessor structures. As these structures become more and more complex the use of arbiters having very sophisticated arbitration rules is needed. Presently most of the arbiters which have been studied are based on two simple arbitration rules : linear or circular (one only is cyclical and allows mixed priority schemes[5]) and no systematic design rules exist.

In this paper we present a contribution to the systematic design of arbiters having complex arbitration strategies based on three rules : linear, circular and cyclical.

The basic structure corresponds to a modular synchronous arbiter and the problem consists in designing the decision part as a state machine whose construction is obtained by three successive rules.

INTRODUCTION

Local parallel shared bus structures require fast access procedures to the bus. Among the different possible techniques [1] selection techniques are the most efficient in this case. As distributed structures become more and more complicated, the definition of effective priority rules of access is needed. This can be obtained by using centralized arbiters able to implement varied arbitration rules.

At present some authors have proposed different structures of arbiters [2,3,4,5,6] among which synchronous ones are well suited to shared bus multimicroprocessors systems. Nevertheless the arbitration schemes are for the most part very simple : linear or circular and no systematic construction rule is given.

The aim of this paper is to define systematic construction rules for the decision block of a synchronous arbiter previously proposed [6]. By using three operators corresponding to elementary allocation strategies : linear, circular, cyclical, assembling of these operators can lead to very sophisticated arbitration decisions. The rules proposed lead to the progressive construction of the decision block whatever its complexity can be.

BACKGROUND

The signaling convention uses the request-grant mode.

The structure of the arbiter proposed in [6,7] is made up of five blocks (Figure 1) : input, detection and end of requests, decision, sequencing, output.

The meaning of the signals is as follows : $\{R_i\}$: request lines, $\{G_i\}$: grant lines, $\mathcal{R}$: detection of requests, $\mathcal{E}$: detection of an end of request, LIR : load input register, DEC : decision, LOR : load output register, COR : clear output register, CP : clock pulse.

The transitions of the arbiter are controlled by the sequencing block whose state diagram is given in Figure 2.

DEFINITION OF ARBITRATION RULES

After having loaded the input requests the arbiter must select one of them according to the arbitration rule chosen. The arbitration rule is programmed in the decision block of the arbiter as a state machine.

In this paper, we propose to use and combine three arbitration rules :

L linear (1L2L...LN) represents a strict priority between users 1,2, and N decreasing from 1 to N.

R round robin or circular (1R2R...RN) represents a fair allocation strategy. If user K has been served, user K+1 has priority on all other users.

C cyclical (1C2C...CN) represents also a fair allocation strategy in which a user is served according to all the previous services granted by the arbiter. For instance consider 1C2C3 ; suppose that user 2 has been served, followed by user 1 and that users 2 and 3 are simultaneously requesting. A circular strategy serves user 2 whereás a cyclical strategy serves user 3.

Figure 3 is an example of the decision block for a circular strategy and a 4-user arbiter.

The combination of these rules is performed by using brackets.
Example : ((1L2L3)R(4R5)) is an arbiter which gives a circular priority between the block (1L2L3) and the block (4R5). In the block (1L2L3) the priority is linear ; it is circular in the block (4R5).

The construction of a strategy from elementary ones allows to define very sophisticated arbiters according to the requirements of the multimicroprocessor structures in which they must be used.

The problem consists in designing the state machine of the decision block and this task can be very tedious if the strategy is complex. For instance, as will be shown in the next part, the decision block of an arbiter with strategy ((1R2)R(3R4)R(5R6)) is a 32 states machine with 160 labelled arcs. Our contribution consists in defining rules for the systematic construction of decision blocks using any block based combinations of L, R and C strategies.

DEFINITION OF CONSTRUCTION RULES

The systematic construction of the state machine of the decision block of an arbiter is carried out in three steps :
1. Determination of the states
2. Determination of the arcs
3. Labelling of the arcs

0190-3918/82/0000/0322$00.75 © 1982 IEEE

(Lenght limitations of this paper imply that construction rules are given without proofs).

Determination of the states

Let us consider the three basic cases : (1L2...LN), (1R2...RN) and (1C2...CN), which are fully linear, circular and cyclical strategies respectively and let us call them L-block, R-block and C-block. The L and R blocks are representable as n states machines each state i being associated with user i, whereas the C block is represented as a n! states machine because (n-1)! states must be associated with each user i in order to keep track of the past services (represented by the permutations on n-1 users) consider now the following arbiters in which U is also a user and S can be any of L R or C strategies.

((U)S(1L2...LN)) ((U)S(1R2...RN)) ((U)S(1C2...CN))

In the first case, when U has been served, the next user to serve is the highest priority user requesting in the second block regardless of the past. In the second case the next user to serve in the second block must be determined according to the position of the last user served on the priority ring. In the third case all the past of the second block (all the possible permutations between users) must be memorized. Consequently in the first case one state is sufficient for (U) whereas n and n! are necessary in the second and third cases, respectively. This leads to the following definition :

Definition : The multiplicity of a block B_i is a number M_i which gives the number of times states of other blocks B_j at the same level of the factorized expression of the strategy must be repeated. This is to keep memory of the state of block B_i when it is left.

Example : Let 1,2,...N be n users.
The multiplicity of (1L2...LN) is 1
The multiplicity of (1R2...RN) is n
The multiplicity of (1C2...CN) is n!

A one user block may be considered as being of any L,R or C type because its multiplicity is always 1.

In case of embedded blocks the calculation of the number of states of the upper block which is in fact the complete decision block itself must be made according to the subblocks which constitute it. Then two parameters are necessary to caracterize a block : - the number of its states NS
- its multiplicity M.

It can be shown that these parameters are obtained by recursive formulas, which at a given level of the factorized expression are :

$$NS^{(\ell)} = \sum_{b=1}^{B} NS_b^{(\ell-1)} \cdot \prod_{\substack{i=1 \\ i\neq b}}^{B} M^{(\ell-1)} \quad \text{if the level } \ell \text{ operator is L or R}$$

or $$NS^{(\ell)} = (B-1)! \sum_{b=1}^{B} NS_b^{(\ell-1)} \cdot \prod_{\substack{i=1 \\ i\neq b}}^{B} M^{(\ell-1)} \quad \text{if the level } \ell \text{ operator is C}$$

$$M^{(\ell)} = \prod_{i=1}^{B} M_i^{(\ell-1)} \quad \text{if the level } \ell \text{ operator is L}$$

or $$M^{(\ell)} = B. \prod_{i=1}^{B} M_i^{(\ell-1)} \quad \text{if the level } \ell \text{ operator is R}$$

or $$M^{(\ell)} = B! \prod_{i=1}^{B} M_i^{(\ell-1)} \quad \text{if the level } \ell \text{ operator is C}$$

with B : number of subblocks of the considered level.

The application of these formules starts from the level1blocks. Once the upper block is reached, the number of states which is necessary for each user is known.

Examples : * (((1R2)R(3R4))R(5R6))

{NS=32 users 1,2,3 and 4 need 4 states
users 5 and 6 need 8 states
M=32

* ((1C2C3)L4)

{NS=12 users 1,2 and 3 need 2 states
user 4 need 6 states
M=6

REMARK :
- the number of states of a n user fully linear block is n,
- the maximal number of states of a n user fully circular block is 2^{n-1},
- the maximal number of states of n user fully cyclical block is n!

A fully L(or R(or C)) block is a block made up of L(orR(orC)) subblocks only.

Determination of the arcs

Each state assigned to one user has (n-1) outgoing arcs because any other user (n users) may be served after it. Consequently, the total number of arcs is NS.(n-1). The determination of the destination of these arcs can be made systematic if an indexing of the states pointing out the origin of their multiplicity is given.

Indexing states

The following rule is applicable.

Rule 1. Indexing of states is applicable from the lowest block level. Each state is indexed relatively to the blocks which cause its multiplicity to increase, by the names of the users the memory of which must be kept.

Example : (((1R2)R(3R4))R(5L6))

gives : 1_3 2_3 3_1 4_1

1_4 2_4 3_2 4_2

5_{13} 5_{14} 5_{23} 5_{24} 5_{31} 5_{32} 5_{41} 5_{42}

6_{13} 6_{14} 6_{23} 6_{24} 6_{31} 6_{32} 6_{41} 6_{42}

8 states are necessary for user 5(the multiplicity of ((1R2)R(3R4)) is 8) because after 5 has been served it must be possible to decide which one of users 1,2,3 and 4 must be served according to the priority rules. 5_{13} is an abbreviated from of 5 preceded by 1 preceded by 3.

Determination of the outgoing arcs of a state

Consider state i_α and let us call $\alpha = \{\alpha_1,\dots,\alpha_p\}$ the set of components of index α. An arc exists which comes out i_α to one of the states associated with user j. Determining which one of the j states is reached by i_α needs the following consideration : the past of the system which was kept in state i_α must also be kept in state j_β in order to agree with the arbitration stategy rule. Consequently the rule below holds :

Rule 2. An arc connects i_α to j_β if β contains subscripts belonging to $\{\{\alpha,i\} - j\}$ in the same order as i_α if the order is significant.

Example : ((1C2)C3C4) Each state has 3 outgoing arcs. The outgoing arcs of state 1_3 are directed to 2_3 3_{14} 4_{13}.

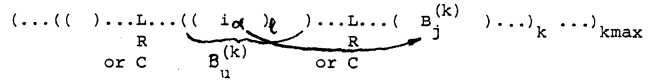
Labelling the arcs

This is the last step in the construction of the state machine of the decision block of an arbiter. Apply the following procedure :

Step 1. Each outgoing arc of a state receives as label the name of the user corresponding to the state it is directed to.

Step 2. The labels of the arcs originating from the same state must be made exclusive according to the strategy of the arbiter. Let i_α be this state and ℓ its block level. Apply iteratively from the higher block level k_{max} to level ℓ.

2.1. Let k be the current level and u the position of the block which contains i_α in that level. Consider arcs whose destination states belongs to a block $B_j^{(k)} \neq B_u^{(k)}$ ($B_j^{(k)}$ stands for the kth. level block in position j). Let $b^{(k)}$ be the number of blocks at level k.



a). The level k operator is L

Arcs are labelled by the product of complemented users names at the left of $B_j^{(k)}$. Exclusion between these labels is performed according to the priorities of the corresponding users in the block $B_j^{(k)}$ with respect to its operator type (L R or C). If the operator is L the priority is from left to right ; if the operator is R the priority is given to users whose name is not in α according to their order in the block ; if the operator is C the priority is determined according to the order of the users of the block $B_j^{(k)}$ in the subscript α .

b). The level k operator is R. Two cases have to be distinguished : * $j > u$. The arcs are labelled by the product of complemented users' names which are in blocks $B_{u+1}^{(k)}$ to $B_{j-1}^{(k)}$. Exclusion between them is performed like in a.

* $j < u$. The arcs are labelled by the product of complemented users' names which are in blocks $B_{u+1}^{(k)}$ to $B_{b(k)}^{(k)}$ and $B_1^{(k)}$ to $B_{j-1}^{(k)}$. Exclusion between them is performed like in a.

c). The level k operator is C. Each arc is labelled by the product of complemented users' names
* which are not in α and not in $B_j^{(k)}$,
* which are in α at the right of the name of the user to which the arc ends in $B_j^{(k)}$.

2.2. Let k be the current level. Consider arcs belonging to $B_u^{(k)}$:

a). The level k operator is L. These arcs are labelled by the product of complemented users' names at the left of $B_u^{(k)}$.

b). The level k operator is R or C. These arcs are labelled by the product of complemented users' names which are not in $B_u^{(k)}$.

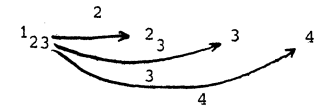
Example : (1C(2R(3C4))) NS = 8 ; M = 8

Indexed states : 1_{23} 1_3 1_{24} 1_4 2_3 2_4 3 4

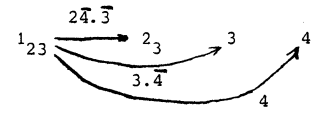
Outgoing arcs (24)

$1_{23} \rightarrow 2_3$	3	4	$2_3 \rightarrow 1_{23}$	3	4
$1_3 \rightarrow 2_3$	3	4	$2_4 \rightarrow 1_{24}$	3	4
$1_{24} \rightarrow 2_4$	3	4	$3 \rightarrow 1_3$	2	4
$1_4 \rightarrow 2_4$	3	4	$4 \rightarrow 1_4$	3	4

Labels :



: step 1



: step 2.1.c

...etc (Figure 4)

IMPLEMENTATION

The structure of the arbiter is modular and 4 modules are independant (for a given number n of users) of the strategy employed. The implementation of a given strategy can be obtained by programming the decision module constructed as shown in Figure 5. The PLA_1 and PLA_2 contain the equations of the excitation variables and of the output variables respectively corresponding to the state machine of the decision block designed by the above procedure. By using a state variable register of lenght $Log_2(n!)$ bits any n user strategy can be implemented by programming the two PLASonly.

By using off-the-shelf TTL-S circuits and 82S100 FPLAS for the implementation a response time of 150 ns is obtainable (CP=50 ns).

CONCLUSION

In this paper, a method for the systematic implementation of complex strategy arbiters has been presented. It is based on the use of a modular synchronous arbiter. Further studies include the definition of multistrategy arbiters and cascadable arbiters based on the same principle.

REFERENCES

1 Thurber K.J., Masson G.M., "Distributed processor communication architecture", Lexington books, 1980.

2 Plummer W.W., "Asynchronous arbiters", IEEE T.C., vol.C-21, n°1, January 1972.

3 Pierce R.C., Field J.A., Little W.D., "Asynchronous arbiter module", IEEE T.C., vol.C-24, n°9, September 1975.

4 Højberg K.S., "One step programmable arbiters for multiprocessors", Computer Design, November 1979.

5 Courvoisier M., "A programmable arbiter for multiprocessor systems", Digital Processes, vol.5, n°3-4, 1979.

6 Courvoisier M., "Un arbiter N-utilisateurs - une ressource, programmable", Electronics Letters, July 1979.

7 Courvoisier M., Geffroy J.C., Seck J.P., "A self-testing arbiter circuit for multimicrocomputer systems", 10th Fault-Tolerant Computing Symposium, Kyoto, October 1980.

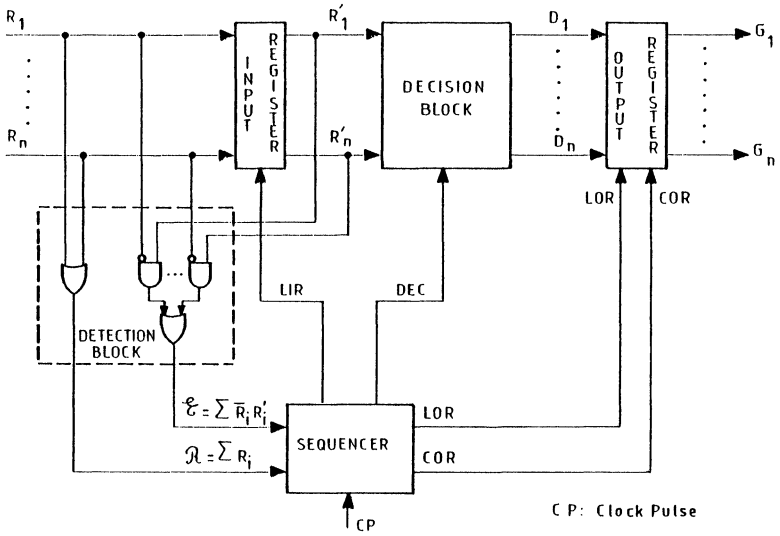


FIGURE 1. Structure of the arbiter

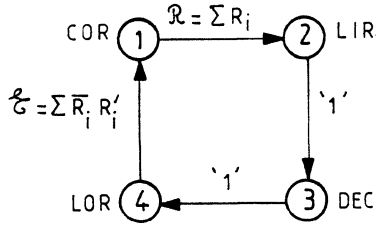


FIGURE 2. The sequencing block

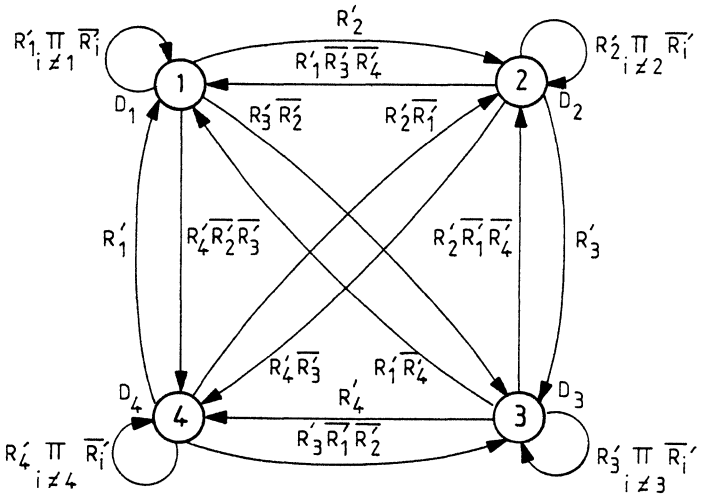
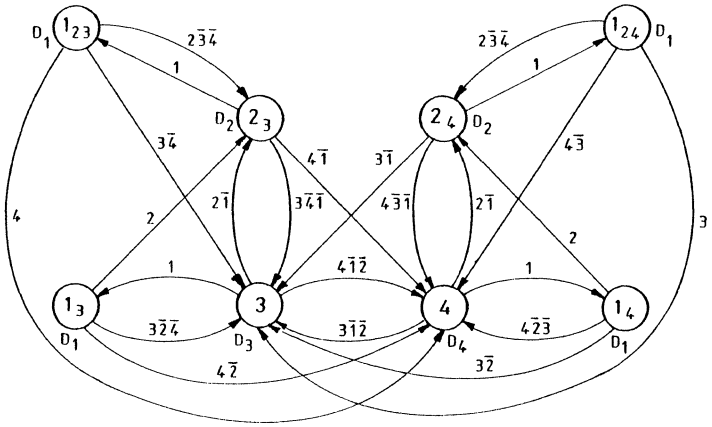


FIGURE 3. The decision machine a 4 user circular arbiter



1,2,3,4 : stands for R'_1,-,R'_4 (stored users' names)

FIGURE 4. The decision machine of (1C(2R(3C4)))

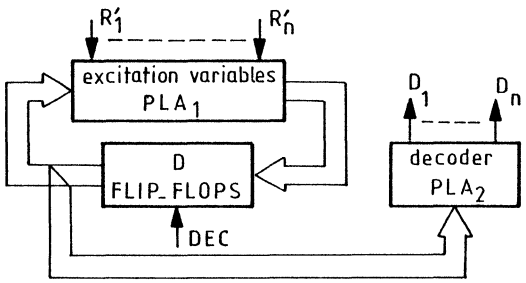


FIGURE 5. Structure of the decision block

USING WRITE BACK CACHE TO IMPROVE PERFORMANCE OF MULTIUSER MULTIPROCESSORS*

R. L. Norton and Jacob A. Abraham
Computer Systems Group
Coordinated Science Laboratory
University of Illinois
Urbana, Illinois 61801

Abstract

In the context of a multiuser multiprocessor system with private cache, we consider the write through versus the write back policy of main memory update. The write back policy has the advantage that the bus traffic is reduced compared to the write through policy. It is usually assumed that the coherence problems of write back require hardware such as global directories to detect potential coherence problems. For this reason a write through cache is usually used which provides coherence for all transactions.

In this paper we suggest ways to avoid coherence problems altogether in user code, and examine the potential savings due to being able to use a write back rather than a write through cache, in terms of bus traffic. Using a detailed instruction level simulation it was found that in the typical case the write back policy will allow greater than double the number of processors on the bus at a given traffic level, compared to write through.

I. Introduction

The shared bus approach to multiprocessing is very attractive since it is simple to implement and easy to use in a multiuser timesharing environment. Standard busses such as the Multibus, Versabus, and S-100 bus all have provision for multiple processors on the bus [1,2]. Larger computer systems such as the VAX 11-780 have been converted for shared bus multiprocessor operation [9]. Modern operating systems which are process based (VMS, UNIX, AOS) are particularly well suited to such an environment [3,7].

The obvious disadvantage of the shared bus approach is that the bus (and memory), being the only shared resource, is a bottleneck. In [9] a dual processor VAX is described and it is reported that bus saturation occurs somewhere between 2 and 3 processors. Providing multiple paths to memory which can be switched to allow concurrent access by multiple processors eases this problem, and a great deal has been written on this approach [5,6]. In the case of relatively few processors, however, it is convenient to avoid the complexity of cross bar or delta switches and attempt to connect the processors on a single bus. One can then consider connecting this substructure to others through various switching networks, as in Cm* [11].

*This research was supported in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy, and U.S. Air Force) under Contract N00014-79-C-0424 and in part by the Naval Electronics Systems Command under VHSIC contract N00039-80-C-0556.

In a shared bus system the number of processors which can be supported depends on the bus bandwidth available, hence it is important to consider ways of reducing the traffic on the (single) memory bus. One method is to use a private cache for each processor. The effectiveness of cache memories in improving performance of computer systems is well known [12,13]. The most obvious advantage of the cache is the reduced access time for cache relative to that of main memory. The use of a private cache can also reduce traffic on the memory bus. While this is of secondary interest in uniprocessor systems, it is of critical importance in bus coupled shared memory multiprocessor systems.

In this paper we consider methods of cache organization which offer reduced bus traffic compared to the methods commonly used in existing uniprocessor designs. We consider a general purpose time sharing system for which a detailed bus transaction level simulation has been constructed. Under realistic assumptions, we find that the bus traffic can be reduced in the typical case by a factor of greater than 2, and for some systems by a factor of greater than 8, by employing these techniques. This surprising result directly translates to having more than twice as many processors in the system at a given level of bus saturation. These techniques deal mainly with the update policy of main memory. This improvement is made with no degradation of common or desirable operating system functionality. In particular, neither interprocess communication nor symmetric multiprocessing are precluded.

II. Cache Coherence and Potential Gain due to Writeback

The two major categories of cache organization, shared and private, are shown in figures 1 and 2 respectively. Examples of these structures (in the uniprocessor case) are commercially available. The VAX 11-780 from Dec uses private cache while the Data General MV/8000 uses a shared cache [4,11]. The ATU (address translation unit) is shown between the CPU and the cache, indicating that the virtual addresses which are issued by the CPU are translated into physical addresses which index the cache. There are advantages to placing the ATU between the cache and the system bus (the cache is then indexed by virtual addresses) and this organization is under study now. For the remainder of this paper we will assume the usual case of translating the addresses before indexing the cache, as illustrated in Figures 1 and 2.

As in any hierarchical memory system the question of coherence among multiple copies of logically identical data items (e.g. a cached item and its copy in memory) must be resolved [10]. The shared cache in figure 2 has no coherence

0190-3918/82/0000/0326$00.75 © 1982 IEEE

problem, since there is no device that modifies the memory without going through the cache. This allows the use of a write back rather than a write through policy for main memory update. This is the organization used in the MV/8000.

In the private cache structure of figure 1 there is potential for cache coherence problems even in the uniprocessor case, since DMA I/O can modify cached data. In the VAX 11-780, which uses this structure (with a single CPU) the cache monitors the bus for writes to locations that it has cached. When it detects one it marks the corresponding cache slot empty so that the next access will be forced to read the modified value from memory. Writes to cache can be immediately passed on to main memory, and the memory system is able to queue write requests, so that the processor can continue without waiting for the write to complete. Read access to the cache, of course, requires no transaction on the system bus, hence the private cache saves bus traffic over the shared bus cache.

There is a third alternative, namely using a write back policy in a shared cache. Since cached values are written only on a cache fault that requires a replacement into memory, or at context switch time, cached values may be modified in cache more times than they are written to memory. This represents a potential savings in bus traffic over the write through case. In addition, the complexity of queued writes to main memory can be avoided. The disadvantage is that a "modified" bit must be maintained in the cache and at context switch time any modified words (or blocks) must be written out. This causes bus traffic to be related to context switch rate. The main problem with write back is the coherence problem, which we will consider now.

A user process as modeled in figure 3, executing in a timesharing environment, will typically do all of its I/O via system calls and in the usual case will be doing blocking I/O. Interprocess communication will also be done via system calls (as opposed to directly writing shared memory). It seems then that user code need not worry about coherence, so that any write through operation from a user process represents an unneeded bus transaction. This is the motivation for considering how much traffic is used for write through, and whether it can be avoided.

In a process based operating system, which is typical of what is run on the systems considered here, a process can be blocked, ready, or running. We will assume that a ready process can execute on any of the processors in the system, and that when a processor is ready to run a process, that process is taken from a central queue in an atomic operation which is not susceptible to races among processors. This is not a difficult objective to achieve, and it allows the operating system to be largely independent of the number of processors that are connected to the bus. Figure 3 illustrates the major states that a process can be in, and some conditions under which transitions occur. This simple model is not at all unrealistic for consideration of the execution phase of a process. We will ignore process initiation and termination, since these are boundary conditions during which operating system control of cache can be assumed.

Hence, if the following three rules are observed, we can at least ignore the coherence problem for nonsystem code.

(1) The process does no I/O itself. This does not restrict the operating system from initiating DMA I/O into the process address space.

(2) When a process is in the blocked or ready state, there are no values from the process address space in the cache.

(3) When the process communicates with another process, it does so via a system call, as opposed to (for example) writing into physical memory that the receiving process is expecting to use for communication.

While it is beyond the scope of this paper to treat operating system implementations relative to cache policy, we have considered the problem. Suffice it to say that these rules do not preclude services such as nonblocking I/O, multiple event wait, and interprocess communication, which we feel are essential in any multiprocessing system. Having confined the coherence problem to the operating system we appeal to the fact that the system can be aware of when coherence problems can arise. In the case of interprocess communication it is possible to implement message passing by mapping a block of memory into the receiver's address space. Since the pages were previously unmapped (not in the receiver's address space), they certainly are not in cache, so there is no coherence problem. A less elegant alternative which has been used in DEC-10 dual processor systems under SMP (symmetric multiprocessing) is for senders to cause a cache flush in the receiver's cache. There are more intricate hardware solutions in the literature as well [10].

In the later sections we will consider the amount of bus traffic that can be saved by using a write back cache in various system configurations. In what follows we assume that the problem of coherence is dealt with as suggested above. We now discuss the simulation system used.

III. Simulation System

There are many ways of analyzing a complex system such as the one in Figure 1. These range from the stochastic approach of characterizing a system in terms of a small number of statistical parameters to the empirical investigation of a realization of the system. We have chosen to simulate the system at a fairly low level; i.e. instruction timing and bus conflict behavior are faithfully replicated, but those phases of operation which are not directly of interest relative to cache performance, such as instruction decode details, are not included. The simulation will accurately reflect, for example, alternating bus access by the processors. The system is driven by execution of target system code.

The amount of concurrency inherent in this system precludes the exclusive use of a standard sequential programming language. We use the C programming language to express the sequential parts of the target system. To extend the language for the simulation of highly concurrent systems, we have constructed a simulation environment which provides for process creation, termination, synchronization, and communication. This allows a very natural expression of the semantics of a digital system since typical hardware systems can be accurately viewed as a collection of processes. In our example we have only 3 processes. These are the bus process and two processes to realize the processors. Figure 4 illustrates the structure of the system. The kernel portion is written in assembly language since it needs to be able to maintain multiple data segments for the processes. The utilities are written in C, as are the simulation modules CPU0, CPU1, and BUS.

Part of the benefit of using the simulation environment is that code can be shared. For example, there is only one copy of the code which implements the CPU element, and there are two independent processes that execute this code. In this sense the simulation system is modular. To increase the number of processors on the bus we merely invoke a third copy of the CPU process by changing two lines of code in the simulation. Changing the design of the target in this fashion is quite simple and this allows the designer to evaluate several different system configurations in a matter of a few hours.

The most important functional capability provided by the simulation system is the ability to manage several processes in a single address space. Management includes the following.

Process creation and termination
: Processes can be created and terminated dynamically.

Priority scheduling of processes
: Processes can be initiated at any of 4 priority levels, with all processes at a given priority executing in a round robin fashion.

Maintenance of a sleep queue
: Processes typically indicate that they are going to incur a time delay by calling sleep. A memory module would for example issue the call "sleep(450)" to indicate that a memory access requires 450 ns. Other processes will continue executing during this 450 ns period if possible.

Signal, wait, and semaphores
: These are provided for interprocess communication and synchronization.

The simulation system implements the target as a collection of processes that run in the single address space of a UNIX process. This feature is critical to good performance. We incur a penalty of only about 30 instructions for signal and wait, since there is no need to call the operating system to communicate with other processes, in contrast to other simulation systems [14].

IV. Analysis of the Effects of Cache Policy

Within the context of a system such as that in Figure 1, there are many parameters which can be varied without violating the basic structure. We consider the following.

(1) Cache policy: write back vs. write through

(2) Length of time slice.

(3) Timing parameters such as bus speed and behavior with and without cache.

(4) Cache block size.

Our main result deals with the amount of bus traffic that can be saved by using a write back rather than a write through policy. For the write back case, the length of time that a process runs without a context switch (and attendant write back) is also examined. We refer to this time as the timeslice.

As we have mentioned, the simulator used here is a low level deterministic simulator. To evaluate a given design parameter, a test program is run on the (simulated) target. In our case a compiler for a simple variant of Pascal was written to allow reliable and convenient generation of nontrivial target programs. This language was chosen for convenience since some support software for its execution was already in existence. The hypothetical target processor was chosen because it is architecturally interesting and straightforward to implement. There is nothing inherent in our analysis technique that precludes evaluation of existing real machines. We have in fact done so in evaluating a similar system incorporating PDP-11 processors, and an effort to compare the current system to a similar structure using the Motorola 68000 processor is under way. Since we are concerned here mainly with the issue of bus traffic and cache effects, the detailed issues of the processor architecture are largely irrelevant, as long as the processor used is similar in its address reference behavior to conventional machines. Our simulation has been so designed.

To illustrate the role cache plays in reducing bus traffic, a simple program was run on the system and bus speeds were varied. The program sorts elements in a matrix by calling several subroutines. This program was used because, in contrast to the Gaussian elimination program used later, it depends heavily on subroutines.

Figure 5 is a plot of average bus utilization versus bus speed for a two processor system with private cache. The higher curve is for the case that the cache is turned off completely. Note that saturation occurs at a much slower bus speed for the no cache case, indicating that the cache is effective in keeping the processors off the bus. It should be noted that the statistic given (bus utilization) is not a useful measure of performance since the amount of time a processor spends waiting due to conflicts is not indicated. It is not difficult to obtain conflict statistics from the simulator, but for this study it suffices to examine the execution time of the various configurations (see Figure 6 for example).

A more interesting example is shown in Figures 5 and 6 which show traffic and execution time respectively as a function of timeslice. The analysis of the cache behavior has been carried out on several variations of the following configuration.

(1) The cache is two way set associative and can hold 1Kb. Both code and data are typically cached, and the block size is 4 bytes.

(2) The cache is private to the processor, one cache per processor as in Figure 1.

(3) The memory is simplistic in that no requests are queued; if a word is written on a write through cycle the processor waits until that transaction is complete before proceeding.

(4) The bus is relinquished at the end of each cycle, so that if contention occurs, processors will alternate bus cycles.

The code in this case is a 30 X 30 Gaussian elimination program. The cache size is 256 32 bit words. This is intentionally somewhat small compared to the size of the code plus data for the program, which totals about 5600 bytes. The hit rate for this program is typically 95%.

The Gaussian elimination program does no I/O and is inherently free of subroutine calls. To simulate the effect of operation in a timesharing environment, a clock tick interrupt occurs at regular intervals corresponding to a transition from the running to the ready state of Figure 3. It is assumed that on return to the running state the cache appears empty and has to be demand loaded. While this is not necessarily the best way to design a system, it is common practice.

At context switch time the write back cache has to write any modified location into main memory. The write through cache has no such requirenment since written values have already been updated. Hence we expect that a very high context switch rate will cause the write back to suffer. As can be seen in Figures 5 and 6 this is indeed the case.

However, even going to the extreme of a context switch every two thousand instructions, the write back strategy is superior by a factor of 1.64 in terms of bus traffic and by a factor of 1.62 in execution time, for this program. While the times as shown in Figure 5 are accurate relative to the assumed speed of the components of the system, they depend on both the speed of the processor relative to the cache and the times for the instructions executed. The bus traffic in Figure 6 is independent of processor speed and we can state that assuming a context switch every eight thousand instructions we need to support 2.57 times as many bus transactions if write through is used.

On a large VAX system, the context switch rate under load is in the vicinity of sixty context switches per second [15]. With the timing used in our experiments this corresponds to about 16000 instructions per timeslice, which will give an even greater advantage to the write back policy. Furthermore, increasing the number of processors for a given multiprogramming load will decrease the context switch rate, further reducing overhead bus traffic. In the limit, assuming processes run to completion, the write back policy requires fewer bus transactions by a factor of 8. If the processor architecture is primarily memory to memory, as in the Intel iAPX 432 [17], the need for cache to reduce bus traffic is even greater. Under these assumptions, and assuming run to completion processing we find that the write through cache issues more bus transactions than the write back cache by a factor of 18 for this problem. This illustrates the need for careful cache design in such systems.

We have also investigated smaller programs and different cache organizations. Figure 8 is a graph of execution time versus bus speed for the matrix sorting problem. The cache in this case has a blocksize of 16 bytes, and we assume that any write back transaction must write a 16 byte block. Write through, of course, requires only a single transaction. The average bus utilization in this case ranges from .36 to .59, even though the bus speed is 2.2 micro seconds, which corresponds to approximately twice the instruction time for this processor. When the processor speed was changed to 30ns for all instructions, with a 100 ns cache, the bus utilization was still only .9 at a bus cycle time of 1.2 microseconds. Thus the adding a write back cache effectively more than doubles (700 ns versus 2000ns at the .55 saturation level) bus bandwidth as measured by the average saturation.

In Figure 9 the bus traffic for this program for the write through and write back case are plotted. The two are equal at a point well below realistic levels of context switch activity. As in the other cases, the write back policy is superior for reasonable context switch rates, though in this case, the improvement is only a factor of 1.3.

Conclusions

This study has shown that for a shared bus multiprocessor organization one can significantly increase the number of processors a given bus can support by using a write back rather than a write through policy for main memory update from cache.

For small to medium size machines, for which the cost of a processor is small compared to the cost of the rest of the system, this is an especially attractive means of improving multiuser performance without adversely affecting the efficiency or the functionality of the operating system.

There are many more parameters and design tradeoffs that we have not considered here. Currently we are investigating the benefits of having a relatively wide path from the cache to main memory, and having the I/O devices communicate using this bus. If we assume that the I/O devices are reasonably intelligent it is possible to move a large part of the file system and operating system services into the device controllers (the file and I/O handlers) and into the bus interface (interprocess communication and syn-

chronization). Current CPU chips are quite impressive in that they have reached the level of being viable for support of useful operating systems. VLSI techniques applied to considerations such as cache design and operating system support as described here will allow the construction of extensible systems that can be configured for an extremely wide range of performance while maintaining component commonality.

REFERENCES

[1] VERSAbus Specification Manual, 1981, Motorola Inc.

[2] Richard W. Boberg, "Proposed Microcomputer System 796 Bus Standard," Computer, Vol 13 No. 10, pp. 89-105, Oct. 1980.

[3] G. H. Goble and M. H. Marsh, "A Dual Processor VAX 11/780," 9th Annual Symposium on Computer Architecture, 26 April, 1982.

[4] VAX/VMX System Services Reference Manual, 1980, Digital Equipment Corporation.

[5] Unix Programmer's Manual, 7th edition, Vol 2b, Jan 1979, Bell Telephone Labs inc, Murry Hill, NJ.

[6] J. H. Patel, "Analysis of Multiprocessors with Private Cache Memories", IEEE Trans. Comput., vol. C-31, pp. 296-305, April 1982.

[7] J. H. Patel, "Performance of Processor-Memory Interconnections for Multiprocessors," IEEE Trans. Comput., vol c-30, pp 771-780 Oct 1981

[8] R. J. Swan, S. H. Fuller, and D. P. Sieworek, "Cm* - A modular, multi-microprocessor," Proceedings of the National Computer Conference, 1977.

[9] K. R. Kaplan and R. O. Winder, "Cache-Based Computer Systems", Computer, pp. 30-36, March 1973.

[10] G. S. Rao, "Performance Analysis of Cache Memories", J. ACM, vol. 25, No. 3, pp. 378-395, July 1978.

[11] C. J. Alsing, K. D. Holberger, et al. "Minicomputer fills mainframe's shoes," Electronics, pp. 130-137, May 22, 1980.

[12] VAX Architecture Handbook, 1981, Digital Equipment Corp.

[13] L. M. Censier and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems", IEEE Trans. Comput., vol. C-27, No. 12, pp. 1112-1118, December 1978.

[14] F. I. Parke, "An Introduction to the N.mPc Design Environment," The proceedings of the 16th Design Automation Conference, June, 1979.

[15] W. N. Joy, "Installing and Operating 4.1bsd" May 18,1981, Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of Calif., Berkeley.

[16] VAX Hardware Handbook, 1981, Digital Equipment Corp.

[17] P. Tyner, iAPX 432 GENERAL DATA PROCESSOR ARCHITECTURE REFERENCE MANUAL Intel Corporation, 1981.

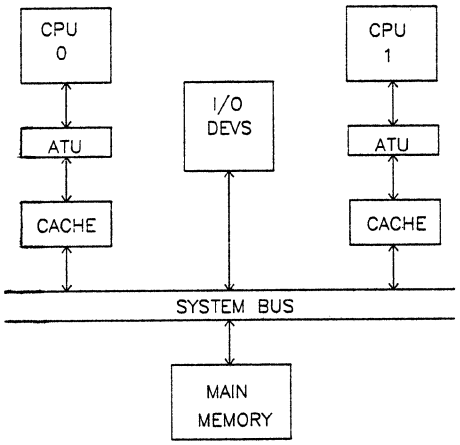


Figure 1. Private Cache System

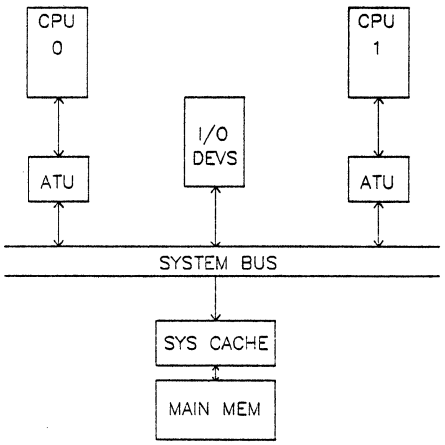


Figure 2. Shared Cache System

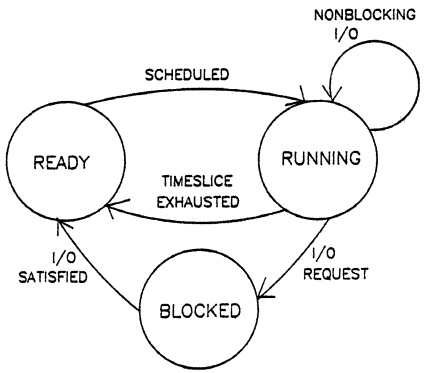


Figure 3. States of Process Execution

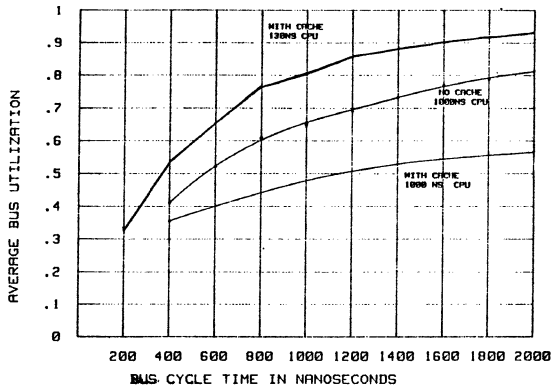
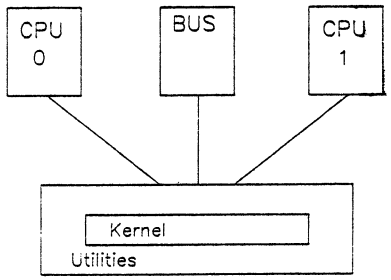


Figure 5. Bus Saturation



Kernel: Process management
Utilities: Instrumentation and kernel interface

Figure 4. Simulator Structure

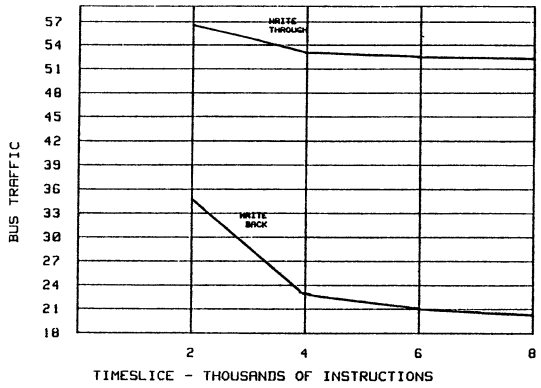


Figure 6. Bus Traffic VS Timeslice

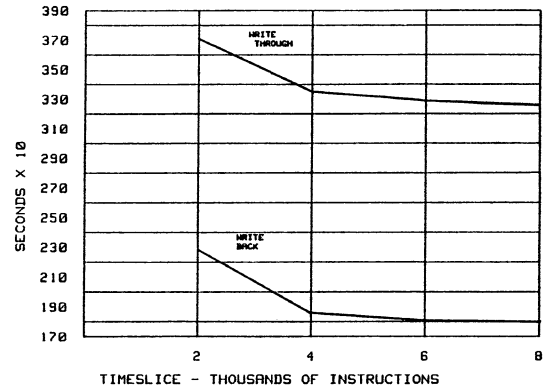


Figure 7. Execution Time VS Timeslice

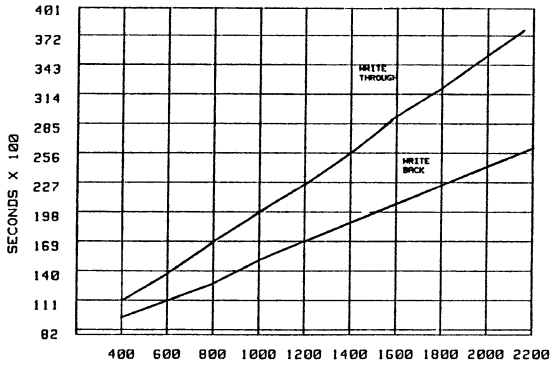


Figure 8. Execution Time VS Bus Speed

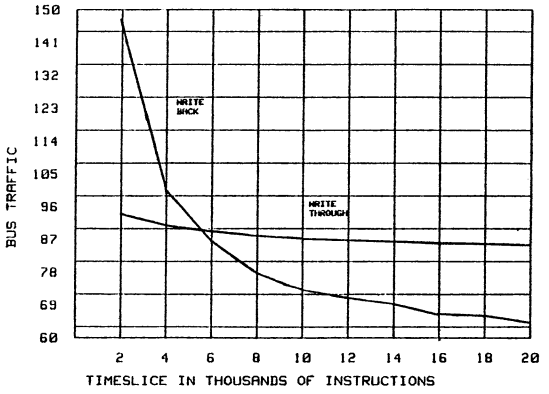


Figure 9. Traffic VS Timeslice

Coherence Problem in a Multicache System†

W. C. Yen‡
Fairchild Advanced Research and Development
Palo Alto, California 94304

and

K. S. Fu
School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

Abstract

Coherence problem occurs in a multicache system when data inconsistency exists in the private caches and the main memory. Without an effective solution to the coherence problem, the effectiveness of a multicache system will be inherently limited. These problems will be closely examined and treated in a systematic top-down manner. A new solution, LSCS (logical semi-critical section) scheme, in which the memory reference of a processor is made as fast as possible, is proposed.

1. Introduction

The architectures of a multiprocessor computer [8,12] are primarily characterized by three attributes: (1) multiple, not highly specialized processors are used, (2) all processors share most, and often all, of the main memory, and (3) each of the processors is able to do computation individually. Advantages offered by these attributes are so fruitful that these architectures will undoubtedly play an important role in the computer of the future.

The modularity and redundancy inherent in a multiprocessor computer offer the opportunity to build a more reliable system. In a carefully designed computer, a failure of a single module does not crash the entire system, instead only a graceful degradation of the system's performance is anticipated. It is also true that in a multiprocessor computer the sharing of resources and processing power tends to smooth out effects due to random variations in workloads. In return, the throughput/cost ratio is increased. This occurs even if each processor in a multiprocessor computer performs worse than when it is in a uniprocessor configuration [12].

The economical advantage of a multiprocessor architecture during the computer construction phase has also been noticed in [15]. The regularity of a multiprocessor computer allows duplication of modules of the same type. Both time and cost of design are thus reduced significantly. Furthermore, the designer of a multiprocessor computer may enjoy the freedom of choosing the most cost-effective uniprocessor element structure, independent of the processing speed of the element.

Due to physical limitation imposed by the existing technology, a uniprocessor computer may not be able to offer enough, or required, processing power. Thus, in spite of other advantages such as expandability, modifiability, etc., the construction of a multiprocessor computer seems to be necessary. Concurrent execution of a number of tasks on different processors which aim at a single computation objective can reduce the overall computation time to a certain degree depending on the nature of the computation and the specific architecture of the computer. To date, all or some of these advantages have been clearly demonstrated to a certain extent by a number of experimental and commercially available computers such as C.mmp, Cm*, PLURIBUS, S-1 Multiprocessor, IBM 370/168, CDC Cyber 170, Honeywell 60/66, Burroughs B7700, and Tandom Nonstop.

Although multiprocessor computers offer many potential advantages, they also generate many problems. In particular, many people have long believed that a multiprocessor computer composed of N processors always yields much less than N times the performance (throughput) of the corresponding single processor computer due to the substantial memory interference and synchronization overhead. Multiprocessor computers are, thus, doomed to waste substantial resources, especially when the number of processors is large, say greater than four.

The experimental data obtained from C.mmp [10] disproved the impression that synchronization overheads, also termed software lockout, is intolerably high in a multiprocessor computer. In fact, in the measurements involving 14 processors, idleness due to locking consumed less than 1 percent of the processors. On the contrary, the cost of memory interference is indeed high. Roughly a factor of 3 in performance degradation has been observed in C.mmp [10] if all 16 processors execute from a common memory. Consequently, the major threat to the performance of a multiprocessor computer is primarily due to the contention in the memory.

Numerous studies aiming at reducing the memory interference have been performed ever since the proposal of multiprocessor computers. They can be roughly grouped into three categories: (1) solutions resort to the static or dynamic memory allocation strategies [4,6,9,13,17], (2) solutions require data tagged by specially designed operating systems [5], (3) solutions assume dynamic hardware support independent from software environments [1,6,11,18]. Among them, we believe that the future general purpose multiprocessor computers will fall into the third category especially for high-performance systems. The architectures with fewer

† This work was supported by the NSF Grant ECS 80-16580.

‡ W. C. Yen was with the School of Electrical Engineering, Purdue University.

0190-3918/82/0000/0332$00.75 © 1982 IEEE

management problems and less special software assists will eventually dominate.

In a computer of the third category, each processor is associated with a private cache by which a certain amount of information is trapped in and retained. As we know, in addition to the usually faster memory cycle time, a cache serves as a lookahead and lookbehind buffer. The lookahead capability of a cache may actually increase memory interference unless the additional words brought along with the missing word into the cache do not introduce extra fetches to the main memory. Moreover, the cache capacity has to be large enough to insure that the utility of the lookahead is no less than that of the lookbehind. Nevertheless, a system with less memory interference does not necessarily result in a better performance. Good performance is a result of a balance between the degree of memory interference and the cache hit ratio. This subject has been studied in [18]. On the other hand, the information retained in a cache, termed lookbehind capability, usually has a high possibility to be reused several times before swapped back into the main memory so that the frequency of main memory references is highly reduced. In return, the memory interference is also reduced.

Unfortunately, such a multicache system, as shown in Figure 1, causes coherence problem because multiple copies of a main memory block may reside in multiple private caches at any given time. In general, a coherence problem occurs as soon as two or more access paths to a single data entry exist simultaneously. This problem is vital to the integrity of the system and is regarded as the major obstacle in the design of a multicache system. In order to eliminate such a coherence problem once and for all, an interesting shared-cache system, as shown in Figure 2, has been proposed and extensively studied by Yeh [16]. Each processor, instead of talking to its private cache, goes through the interconnection network and then talks to a cache which is shared by all processors. In principle, the philosophy behind this proposal is to apply the cache memory technology to the conventional main memory and omit all privately owned caches. As a result, the original shared main-memory now becomes the secondary memory. The level of boundary of memory hierarchy for context switching during a page fault, however, is pushed one level down to the boundary between the second and the third level of memory hierarchy in this case. For such a shared-cache system, the coherence problem is truly eliminated but all the original problems which lead us to put the cache into a multiprocessor computer still exist, such as memory interference and transmission delay of interconnection network.

In the following sections, we first illustrate coherence problems in detail and then discuss various solutions for them.

2. Coherence Problem

Coherence problem may occur in a multicache system when data inconsistency exists in the caches and the main memory. Multiple copies of a given main memory block may exist in several private caches. Modification of any copy of this shared block by a processor in its cache will cause an obsolete value of this shared data in every other cache. Data inconsistency thus occurs in the caches. To take specific examples, let us consider a multicache system with N caches, C_i for i=1,...,N, and a main memory being shared by all processors, P_i for i=1,...,N. Let X be the physical address of a main memory block issued by the memory mapping function of a processor. When a copy of block X resides in C_i, let the corresponding cache block address be y_i. Thus, "block X" is always used to denote a specific block in the main memory, and "y_i" implies that a copy of this main memory block resides in the cache block y of C_i. In the following two examples, coherence problems arise:

(E1) Data are assumed to be shared among processes. P_i reads block y_i without noticing that block y_j has been modified by P_j.

(E2) A process is allowed to switch among processors. Process A may be executed on two processors in a sequence such as $\cdots \rightarrow P_i \rightarrow P_j \rightarrow P_i$. As a result, process A may have a copy of block X in both C_i and C_j. If process A has modified block y_j, it will then read obsolete data from y_i after switching back to P_i.

Modification of a copy of block X by a processor in its private cache will result in data obsoleteness in the main memory if block X is not updated immediately. As a result, coherence problems may also arise under the following situations:

(E3) Data are assumed to be shared among processes. A copy of block X is brought into C_i upon a miss while another copy of block X has been modified in C_j and this modification has not yet been reflected in block X.

(E4) A process is allowed to switch among processors. Process A is running on P_i first and then switched to P_j. After process A has been switched, the most recently modified data of process A may still be in C_i. Hence process A running on P_j could read obsolete data from the main memory upon a miss.

These examples are expressed from a process' point of view. Viewing from a processor, in fact, the problems posed in (E2) and (E4) are exactly the same as the problems posed in (E1) and (E3). Since cache memory management is carried out in hardware it is much easier to deal with processors rather than with processes for a multicache system. Thus, restricting processes switching among processors to eliminate (E2) and (E4) does not actually simplify the problems. On the other hand, (E3) and (E4) can be eliminated if the main memory update policy is write-through instead of flag-swap. Nevertheless, without buffering, the rate of accessing the main memory can not be lower than the write rate of a processor in a write-through policy. In the next section, some previous solutions are described. We will then present a new

scheme in which the memory reference of a processor is made as fast as possible. In particular, this scheme is developed in a systematic top-down manner.

3. Previous Solutions

A commonly used coherence scheme in commercial computer systems with a small number of processors is to connect every cache to a high-speed bus on which the addresses of the block to be modified are sent. Each cache permanently monitors this bus and invalidates the affected block in case of a hit. In the mean time, the write-through policy is used to insure the update of main memory. This scheme has many weaknesses [2]. The invalidation traffic on the bus is often very high since the mean write-rate for most processors is between 10 and 30 percent. The peak rate is even much higher, and, a buffer may be needed for each cache to queue up the invalidated addresses. Moreover, a different coherence problem may occur due to these invalidation queues. Finally, the rate of cycle stealing of the cache directory to perform the search for those invalidated addresses is so high that only a small proportion of cache directory cycles is free for normal operations. All of these explain the reason that this scheme has been limited to systems with no more than two caches.

The caches in C.mmp [5] implement write-through in the main memory, but the contents of caches on other processors are not affected. The coherence problem is resolved by having the operating system to designate which pages are safe to cache via the cacheable bit in the relocation registers. Thus, all those shared writable pages have to be in the main memory only. In other words, the shared writable data are centrally managed. The drawbacks of this scheme are the need of a special operating system and the hit ratio of caches is inadequate for a high-performance computer. It should be noticed that the special assist required from the operating system may not be necessary in a capability-based system with architectural supports. In addition, for specific environments, the resulting cache hit ratio may be adequate for a low-budget multiprocessor computer as well. However, this solution obviously is an inherently limited approach.

More recently, three closely related schemes have been developed independently by Tang [14], Censier and Feautrier [2], and Widdoes [15]. They treat each block in the main memory as a semi-critical section [3]. This means that a block X can be shared among several readers but can only be accessed by one writer. Here, a reader stands for a cache C_i in which a copy of block X resides and in the mean time this copy has only been read. A writer represents a cache C_i in which a copy of block X resides while this copy has been written by processor P_i. All the readers or the writer of block X are recorded in a logically centralized map. This map is dynamically updated whenever the state of any semi-critical section is changed. In other words, such a map is designed to keep track of all the readers or the writer of each block X in the main memory. Hence, not only the irrelevant cache invalidation requests can be filtered out but the cache where the most recently modified data of block X reside, can be identified. As a result, the flag-swap policy is adopted in all three proposals.

This map-based approach certainly requires more hardware, but offers a much better performance especially when the multiprocessor computer contains more than two or four processors. It solves the coherence problem without knowing the semantics of the content of each main memory block. Thus, this is a totally transparent approach. However, this approach is based on the concept of semi-critical section not only logically but also physically. Consequently, when the first time a processor is trying to write into a cache block which was loaded upon a read miss, the processor can not execute this write until the state of the corresponding main memory block X is changed even if its cache owns the only copy of block X. We thus refer this approach as the PSCS (Physical Semi-Critical Section) scheme in contrast to our LSCS (Logical Semi-Critical Section) scheme presented in the next section. A digest of the PSCS scheme can be found in [7].

4. The LSCS Scheme

The purpose of using a cache is to feed the data to a processor as fast as the processor demands. Thus, the cache is often integrated into a processor unit and implemented by the same technology as the processor. Moreover, the management of cache memory is completely made by hardware and makes decisions locally as much as possible so that the response time to a processor's demand can be minimized.

The objective of the proposed LSCS scheme is to reduce the effective cache access time by making as many local responses as possible. There are a main memory controller MC and a cache controller CC_i for each cache C_i. These controllers run asynchronously. Commands are exchanged between the cache controller and the main memory controller. A local response means that a cache controller can permit a processor accessing its cache without interacting first with the main memory controller. To understand this proposed scheme clearly, let us define that the legal state of block X viewed from MC is X-state $\{a,b,c,d\}$, where

- a: no copy, or no valid copy, of block X is in caches,
- b: block X is updated and only a single copy of block X is in caches,
- c: block X is updated and multiple copies of block X are in caches,
- d: block X is obsolete and only a single copy of block X is in caches.

Let the legal state of a copy of block X in C_i viewed from CC_i be y_i-state $\{\alpha,\beta,\gamma,\sigma\}$ where

- α: a copy of block X may be in C_i, however, it is invalidated.
- β: an intact copy of block X is in C_i and it is the only copy of block X in caches.
- γ: an intact copy of block X is in C_i, howev-

er, there is one or more copies of block X in other caches.

σ: a modified copy of block X is in C_i and it is the only copy of block X in caches.

These states are illustrated in Figure 3. NP(X) represents the number of copies of block X which resides in caches. Obvious correspondences between b and β, c and γ, d and σ respectively can be recognized. Any change of y_i-state must be reflected in the X-state.

Figure 4 specifies the state-transitions required to maintain data coherence when a write operation is performed by processor P_i. In case of a cache hit on the write reference to C_i, there are four possible states for y_i-state. If y_i-state is α, the situation is the same as a cache miss except that the cache replacement algorithm does not have to be executed. If y_i-state is β, CC_i signals MC to declare a write into block X so that X-state has to be changed from b to d; in the mean time, P_i's write operation and the change of y_i-state from β to σ are carried out as well. This means that a processor's write operation is not delayed if a cache hit is in state β. However, if MC is invoked by another cache controller to inquire the state of block X at this time, there is a slight possibility that the X-state is still in state b but the corresponding y_i-state has been changed to σ. We call this problem "the uncertainty of state b", that is, when MC detects a block X in state b it can not be sure that the block X is indeed in state b or actually in state d. This problem, fortunately, does not complicate the cache control mechanism as much as it first appears to be and will be discussed and resolved later. If y_i-state is γ, CC_i signals MC to declare an exclusive writing in block X. Processor P_i can not write into the corresponding cache block y_i until a X-state-transition completion signal from MC is received. In order to change the X-state from c to d, MC has to inform every other cache controller which owns a copy of block X in its cache to invalidate that copy. In other words, one request on the X-state's transition from c to d will invoke one or more y-state's transitions from γ to α.

In case of a cache miss on the write reference to C_i, a replacement algorithm will be executed and a cache block y_i will be selected for the missing block X. However, if the copy of block X' which originally resides in the selected cache block y_i has been modified, it needs to be swapped back to update the main memory. MC will also check to see if there is only a single copy of block X' left in all other caches after C_i has replaced this copy of block X'. If it is the case, the corresponding cache controller, say CC_j, will be signaled to change the y'_j-state from γ to β to declare that the cache block y'_j owns the only copy of block X'.

Before a copy of block X being loaded into C_i, MC has to declare an exclusive reading of block X for C_i. Thus, all cache controllers which have a copy of block X in their caches will be signaled to invalidate those copies. In addition, if MC detects that a copy of block X has been modified, the main memory needs to be updated first. The uncertainty of state b of block X does not really complicate the declaration of exclusive reading. No matter what the X-state really is the cache controller which has the only copy of block X in its cache has to be signaled to invalidate this copy. Therefore, whether or not this copy has been modified can be checked at the same time.

Figure 5 specifies the state-transitions required to maintain data coherence when a read operation is performed by processor P_i. Nothing has to be done for reading block X if there is a valid cache hit in C_i. IF a cache miss occurs on the read operation to C_i, the same replacement algorithm as the one specified in Figure 4 will be executed. Before a copy of block X being loaded into C_i, however, MC declares a shared reading of block X for C_i. Now, an extra work has to be done by MC for resolving the uncertainty of state b of block X, it is not required otherwise. MC needs to signal CC_j to check the y_j-state if a copy, and the only copy, of block X is in C_j. This uncertainty checking can be done in parallel with loading a copy of block X into C_i. However, if unfortunately y_j-state is indeed d, this is rare, MC will be signaled by CC_j to update the block X and reload an updated copy of block X into C_i. Note that we have to pay attention to the timing of this uncertainty checking since it has to be completed before processor P_i reads block y_i.

5. Considerations of implementation

Sample implementations of the specifications in Figures 4 and 5 are illustrated in Figures 6 and 7 respectively. A 3-bit tag (2 bits if encoded) is associated with each cache block to represent the y_i-state. The tag is interpreted as follows if it is set.

$v_i[y]$: valid bit. The copy of block X in cache block y_i is valid.

$s_i[y]$: single bit. The copy of block X in cache block y_i is the only copy of block X in caches.

$m_i[y]$: modify bit. The copy of block X in cache block y_i has been modified by P_i.

A (N+1)-bit tag is associated with each block in

the main memory to represent the X-state. The tag is interpreted as follows if it is set.

P[X,i]: ith bit in the present array. A copy of block X is in C_i, where i=1,...,N.

M[X]: modify bit. Block X is obsolete.

In addition, a combinatorial logic circuit NP(X), which tells us how many bits are set in the present array of block X, is required in MC.

There is little problem with the cache tag organization. However, there is a variety of different ways to organize the main memory tag for each block, which directly affects the organization of MC. Two most intuitive approaches are available. One is to include the (N+1)-bit tag into each main memory block so that the tags are actually a portion of the main memory space and spread out all over the entire space. The other is to aggregate all tags into a dynamically managed bit map which can be implemented by a faster device. However, the former approach suffers a slow memory reference time; the latter approach has the problem of contention.

We suggest to physically distribute the MC (of course, all tags) as illustrated in Figure 8. Furthermore, the main memory is interleaved by both higher and lower order bits. Two levels of distribution are intended to obtain. The lower level is achieved by associating a module of MC with each main memory module. Thus, the contention in MC is highly reduced. On the other hand, the higher level is achieved by interleaving the memory in higher order bits so that the availability of the main memory is provided. Note that the interleaving on the lower bits is also necessary for reducing potential memory interference on shared code.

With regard to the position of the lower bits, it is dependent on that the main memory is interleaved by block, subblock, or word. This further depends on the ratio of interconnection network switch time (circuit switch) and main memory module access time. In other words, if the circuit switch set-up time is relatively long, interleaving by block would be a better choice. On the contrary, if the main memory access time is relatively long, to interleave the main memory by subblock or word would then be better. Such an architectural decision can be made in terms of manipulating the parameters of main memory access time t_{sc} and block transfer parameter y in our earlier models reported in [18].

Thus, each MC module actually contains a set of tags (a portion of the dynamically managed bit map) for the corresponding main memory module and a replica of MC logic circuit and microprogram. The MC module may be implemented by a faster device than the main memory module and the operations in MC may also be overlapped with those in the main memory module. As a result, the performance degradation due to the addition of coherence mechanism can be highly reduced. We can also take advantage of such an organization to include buffers in MC and CC_i. Because now these controllers are physically associated with each individual memory (main memory and cache) modules, the additional buffers do not affect the coherence mechanism much. The system performance can thus be further improved.

6. Performance Estimates

The use of coherence mechanism always degrades the system performance. The effect of the scheme based on the map-based approach appears in both cache hit ratio and effective memory access time. A lower cache hit ratio will be observed due to inevitable cache invalidations, while the need of interactions with memory controllers slows down the memory access. The cache invalidation rate will be the same for all map-based schemes; thus, the effective memory access time is used as the performance index. A rough comparison between PSCS and LSCS schemes is given in this section.

Let us assume that the probability of a valid hit in cache is h. The probability that a cache block contains the only copy of a main memory block is $(1-\rho)$, where ρ is the multi-copy coefficient. Furthermore, the mean time required for completing a memory access in case of a valid cache hit and without consulting with MC is t. The mean time required for completing a memory access in case of a valid cache hit but requiring a consultation with MC is t'. The mean time required for completing a memory access in case of a miss or an invalid cache hit is T. The effective memory access time may be obtained by assuming that the proportion of data will be brought in the cache by a read miss and be modified subsequently by a write is θ. Then, the effective memory access time for the PSCS scheme is

$$\omega_P = h[1-\theta(1-h)]t + h\theta(1-h)t' + (1-h)T,$$

and for the LSCS scheme is

$$\omega_L = h[1-\theta\rho(1-h)]t + h\theta\rho(1-h)t' + (1-h)T.$$

Thus, the difference is

$$\omega_P - \omega_L = h\theta(1-\rho)(1-h)(t'-t).$$

The amount of performance improvement is directly related to the specific organization and implementation of MC. Nevertheless, this gives a typical performance improvement of about 5 to 15 percent. In particular, the additional hardware overhead of the LSCS scheme is almost negligible.

7. Concluding Remarks

The coherence problem in a multicache system has been treated in a systematic top-down manner. The proposed LSCS scheme offers a better performance with negligible additional hardware overhead.

ACKNOWLEDGMENT

The authors wish to thank F. A. Briggs for providing the initial manuscript on a digest of some previous coherence schemes.

REFERENCES

[1] F. A. Briggs and M. D. Dubois, "Cache Effectiveness in Multiprocessor Systems with Pipelined Parallel Memories," Proc. Int. Conf.

Parallel Processing, pp. 306-313, Aug. 1981.
[2] L. M. Censier and P. Feautrier, "A new solution to coherence problems in multicache systems," IEEE Trans. on Comput., vol. C-27, no. 12, Dec. 1978.
[3] P. J. Courtois, F. Heymans, and D. L. Parnas, "Concurrent Control with "readers" and "writers"," Comm. ACM, vol. 14, no. 10, pp. 667-668, Oct. 1971.
[4] A. A. Covo, "Analysis of multiprocessor control organizations with partial program memory replication," IEEE Trans. on Comput., vol. c-23, no. 2, pp. 113-120, Feb. 1974.
[5] S. H. Fuller and S. P. Harbison, "The C.mmp Multiprocessor," Tech. Rep., Carnegie-Mellon Univ., Pittsburgh, Pa., Oct. 1978.
[6] C. H. Hoogendoorn, "Reduction of memory interference in multiprocessor systems," Proc. 4th Annual Symp. Comput. Arch., pp. 179-183, 1977.
[7] K. Hwang and F. A. Briggs, Computer Architecture and Parallel Processing, McGraw-Hill, Initial Manuscript, pp. 7.34-7.43, July 1981.
[8] A. K. Jones and P. Schwarz, "Experience Using Multiprocessor Systems - A Status Report," Computing Surveys, vol. 12, no. 2, pp. 121-166, June 1980.
[9] J. M. Kurtzberg, "On the memory conflict problem in multiprocessor systems," IEEE Trans. on Comput., vol. c-23, no. 3, pp. 286-293, March 1974.
[10] M. V. Marathe, "Performance Evaluation at the Hardware Level and the Operating System Kernel Design Level," Ph.D. Thesis, Carnegie-Mellon Univ., Pittsburgh, Pa., 1977.
[11] J. H. Patel, "A Performance Model for Multiprocessors with Private Cache Memories," Proc. Int. Conf. Parallel Processing, pp. 314-317, Aug. 1981.
[12] M. Satyanarayanan, "Commercial Multiprocessing Systems," Computer, vol. 13, no. 5, pp. 75-100, May 1980.
[13] A. J. Smith, "Multiprocessor memory organization and memory interference," Comm. ACM, vol. 20, no. 10, pp. 754-761, Oct. 1977.
[14] C. K. Tang, "Cache System Design in the Tightly Coupled Multiprocessor System," AFIP Proc. NCC, vol. 45, pp. 749-753, 1976.
[15] L. C. Widdoes, Jr., "S-1 Multiprocessor Architecture (MULT-2)," S-1 Project Report, 1979.
[16] C. C. Yeh, "Shared Cache Orgaization for Multiple-Stream Computer Systems," Tech. Rep., CSL, Univ. of Illinois at Urbana-Champaign, Urbana, Il., Jan. 1981.
[17] W. C. Yen and K. S. Fu, "Performance Analysis on Multiprocessor Memory Organization," Proc. ACM Pacific '80 Conf. on Dist. Processing, pp. 142-153, Nov. 1980.
[18] W. C. Yen and K. S. Fu, "Analysis of Multiprocessor Cache Organizations with Alternative Main Memory Update Policies," Proc. 8th Annual Int. Symp. Comput. Arch., pp. 89-105, May 1981.

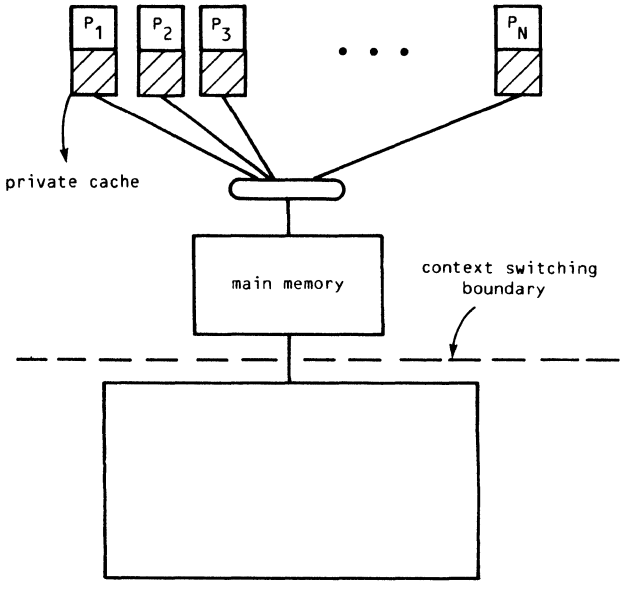


Figure 1. The multicache system

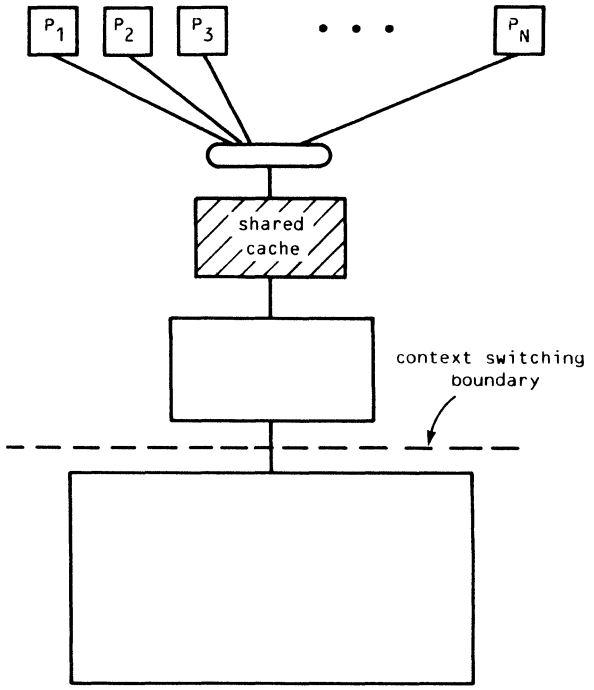
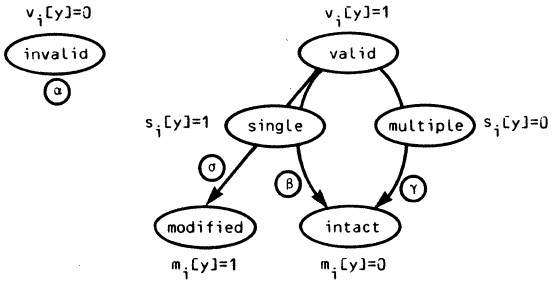


Figure 2. The shared cache system

- y_i-state: the legal state of a copy of block X in C_i viewed from CC_i.



- X-state: the legal state of block X viewed from MC

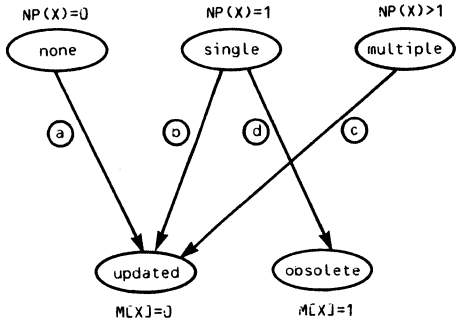


Figure 3. The states of block X

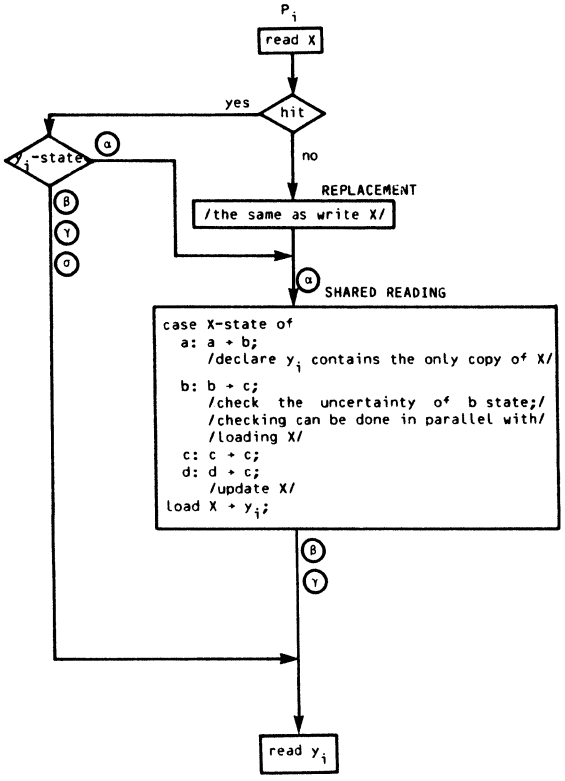


Figure 5. State-transitions for a read operation

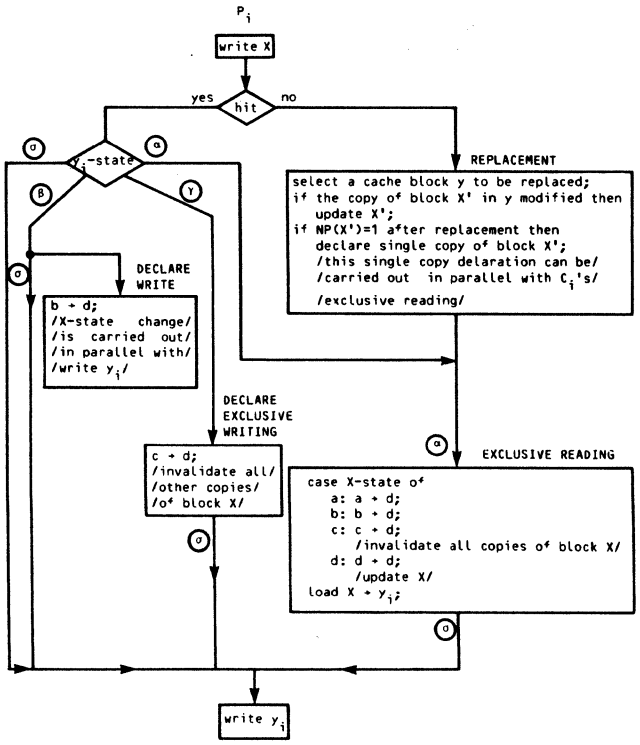


Figure 4. State-transitions for a write operation

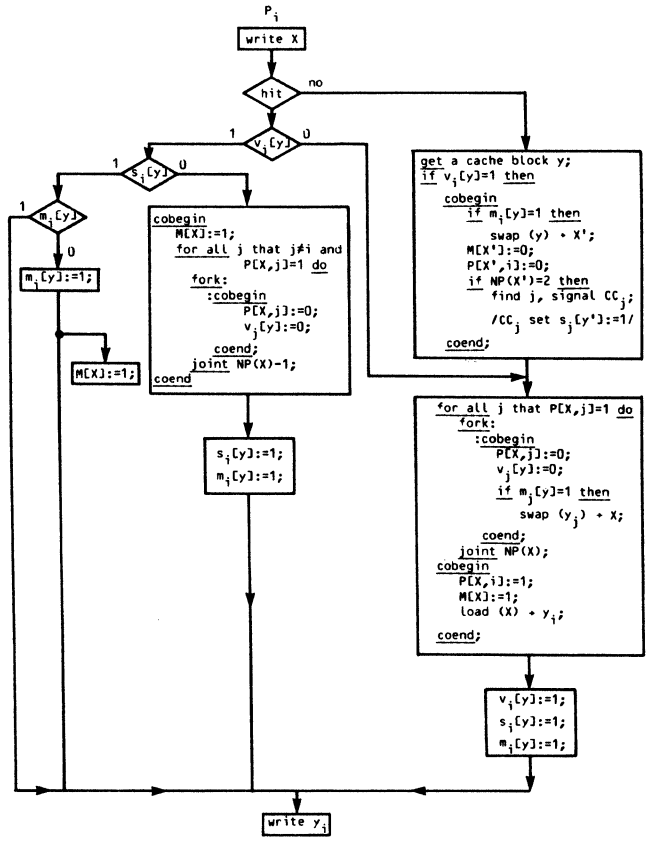


Figure 6. A sample implementation of a write operation

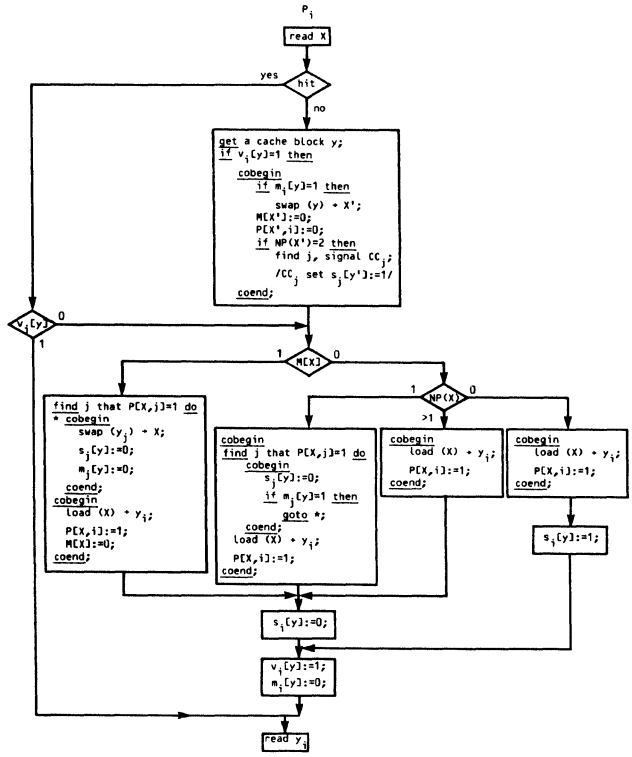


Figure 7. A sample implementation of a read operation

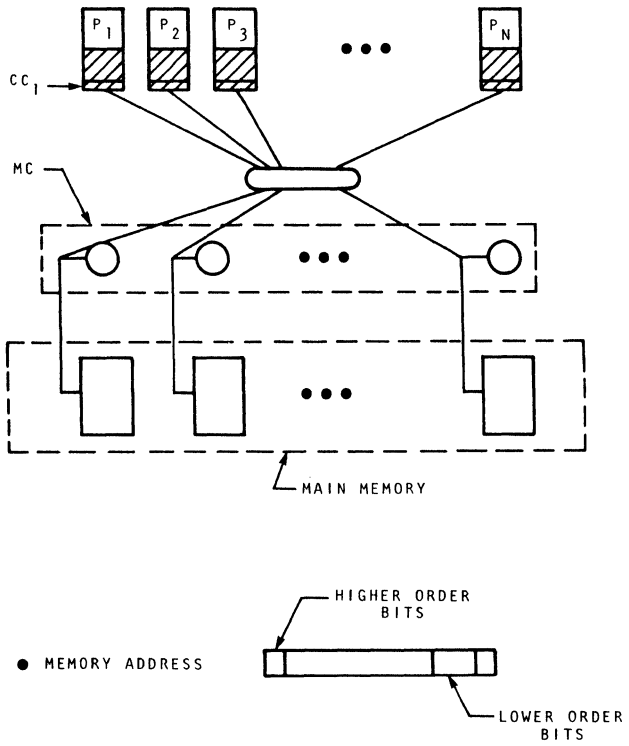


Figure 8. The organization of MC

CONSTRAINED EXPRESSIONS AND THE ANALYSIS OF DESIGNS FOR DYNAMICALLY-STRUCTURED DISTRIBUTED SYSTEMS

Jack C. Wileden*

Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

Abstract -- Designs for distributed systems with dynamic structure can be very difficult to understand and reason about. Constrained expressions, a closed form, non-procedural representation of all the possible behaviors of such a system, can help designers in analyzing a dynamically-structured distributed system's design. In this paper, the constrained expression formalism is introduced, an effective procedure for deriving constrained expressions from procedural design descriptions is outlined, and an example illustrating the use of this procedure in analyzing a design is presented.

Introduction

The Dynamic Process Modelling Scheme (DPMS) and its main descriptive component, the Dynamic Modelling Language (DYMOL) [1], were developed to provide a foundation for software design tools applicable to distributed systems with dynamic structure. A distributed system with dynamic structure is one in which some or all of the system's components may come into and/or go out of existence, and intercomponent communication paths may be established and/or closed, during the life of the system [2]. Systems of this kind are increasingly common; they include such things as process-based distributed operating systems, computer networks, and the tasking facility in Ada. The potential value of DYMOL as a language to aid designers of dynamically-structured distributed software systems has been demonstrated in [1]-[3].

Dynamically-structured distributed systems can be very difficult to understand and reason about, primarily due to the subtle interactions among system components that can arise due to dynamic structure and concurrent activity in such systems. Hence, it is useful to have techniques supporting the succinct description and rigorous analysis of the possible behaviors of a system of this kind. To this end, DPMS provides constrained expressions, a closed form, non-procedural representation for all the possible behaviors that could be realized by some dynamically-structured distributed system.

Constrained expressions are related to other regular expression-based description languages [4] such as event expressions [5], path expressions [6], flow expressions [7] and counter expressions [8]. Constrained expressions are more general than any of these related languages, however. Constrained expressions also permit the description of the behavior of dynamically-structured distributed systems, which these other languages do not.

In this paper, we introduce the constrained expressions formalism, outline the effective procedure for deriving constrained expressions from DYMOL design descriptions and give an example illustrating the use of this derivation procedure in informally analyzing the DYMOL design of a dynamically-structured distributed system. We also outline our current and future work on formal analysis techniques for dynamically-structured distributed system designs based upon the constrained expressions formalism.

*Supported in part by the National Aeronautics and Space Administration under grant NAG1-115

Constrained Expressions

Informal Description

Constrained expressions are a closed form, non-procedural representation of concurrent behavior in the same sense that regular expressions are a closed form, non-procedural representation of the behavior of finite state machines. In fact, the operators used in constrained expressions include the standard regular expression operators (concatenation, alternation, transitive closure) as well as two operators (interleaving and its transitive closure) used to represent concurrent activity. A constrained expression is formed by using these operators to combine symbols from an alphabet of events in the system being described into a collection of subexpressions, one subexpression for each component in the modelled system. The interleave of these subexpressions then represents the unconstrained set of possible system behaviors, ignoring such fundamental properties as the necessity of a message's being sent before it can be received or an intercomponent communication channel's being opened before it can be used in message transmission. The required fundamental properties are formally described by a second collection of subexpressions, called the constraint set. Then the set of behaviors (or, in formal terms, the language over the event alphabet) described by the overall constrained expression is just what remains after the unconstrained set of behaviors is filtered by the constraint set. This filtering process can be formally defined as a set intersection.

Formal Definitions

Constrained expressions define languages over an alphabet, E, of distinguished events. The expressions are composed of symbols from E, symbols from an auxiliary alphabet S of constraint symbols, the special symbols λ (null event sequence) and $\emptyset$ (empty set of event sequences), and a set of operator symbols. The constrained

0190-3918/82/0000/0340$00.75 © 1982 IEEE

expression operators include the familiar operators of regular expressions -- alternation (represented by U), concatenation (represented by juxtaposition), and transitive closure (represented by *) -- plus two operators, Δ and +, used to represent concurrent activity. The Δ operator signifies the shuffling or interleaving of the two strings that are its operands. The unary operator + denotes the interleaving of zero or more copies of its operand.(a)

In defining the language represented by a particular constrained expression, we begin with a representation over an augmented alphabet and use an interpretation rule to produce a set of strings over the actual alphabet of interest. In particular, we let E and S be two disjoint, finite sets called the event alphabet and the constraint alphabet, respectively. A constraint set, CS, consisting of n constraining languages, C_i for $1\leq i\leq n$, can then be defined with respect to n disjoint subsets, S_i, of S. Each such C_i is represented by an expression over S_i, $ex(S_i)$, formed using any of the event expression operators, and interleaved with E* and S_j* for $j\neq i$. That is,

$$C_i=ex(S_i)\Delta S_1^*\Delta\ldots\Delta S_{i-1}^*\Delta S_{i+1}^*\Delta\ldots\Delta S_n^*\Delta E^*$$

for each constraining language C_i, $1\leq i\leq n$. A constrained expression with respect to CS is then defined to be any expression over (E U S) that can be formed using the event expression operators other than +. This expression thus represents a regular language L', which is a subset of (E U S)* and which we call the uninterpreted language of the constrained expression.(b) We also define a homomorphism $H:(E\ U\ S)^*\rightarrow E^*$ by:

$$H(e)=e \quad \text{for all } e \text{ in } E$$
$$H(s)=\lambda \quad \text{for all } s \text{ in } S$$

Finally, for a given constrained expression with respect to CS which represents the uninterpreted language L', we define the interpreted language, L, represented by the expression to be the set of strings over E (i.e., subset of E*) described by:

$$L=H(L'\cap C_1\cap\ldots\cap C_n) \qquad \text{for } C_i \text{ in CS} \quad (1)$$

This definition generalizes the definition given for counter expressions in [8] by allowing for the application of multiple constraints in determining the acceptability of a string.

Analyzing Designs

For an important subset of dynamically-structured distributed systems, DYMOL and constrained expression descriptions are related by an effective procedure for deriving the constrained expressions describing the potential behavior of any given DYMOL description of a system. This effective procedure is similar to the syntax-directed translation scheme commonly employed in compiler construction [9]. By using this procedure, the designer of a dynamically-structured distributed software system can derive a succinct representation of the possible behaviors of a system whose design is described in the more natural, procedural language DYMOL. This succinct, constrained expression representation provides the basis for an informal analysis of the DYMOL design, since it can expose intercomponent synchronization or communication anomalies. It also serves as the starting point for more formal analysis methods, based upon the derivation of systems of inequalities from a constrained expression description, which are currently being developed [10].

For the subset of DYMOL-described systems that we are considering here, the constraint set CS consists of three constraining languages. The first, C_1, describes the necessary restriction on transmission of messages in a distributed system. C_1 is expressed using subset S_1 of the constraint alphabet S, where $S_1=\{@_i,@_i'\}$. The special symbol $@_i$ can be thought of as corresponding to the sending of a particular message along a particular message transmission channel, with the subscript i indexing the specific message type and channel pair. Similarly, the symbol $@_i'$ may be thought of as corresponding to the receipt of a particular message on a particular channel. The constraining language C_1 is represented by the expression

$$C_1=\bigtriangleup_i(@_i^*\Delta(@_i@_i')+)\Delta S_2^*\Delta S_3^*\Delta E^*$$

The event expression (bc)+ represents a set that may be described as "all strings containing an equal number of b's and c's such that any prefix of any string always contains at least as many b's as c's". Thus, C_1 describes the requirement that the reception of a message is always preceded by the sending of a corresponding message, although more messages may be sent than are ever received (due to the interleaved $@_i$*). This precisely captures the message transmission semantics of DYMOL.

Constraining languages C_2 and C_3 are formed using subsets $S_2=\{\$_i,\$_i',\&_i,\&_i'\}$ and $S_3=\{\#_i,\#_i'\}$ of S, respectively. For brevity and simplicity, we omit their detailed definitions. C_2 describes a constraint on the use of interprocess communication channels in a dynamically-structured distributed system. Specifically, it stipulates that message transmission may only take place along channels that are currently operational. C_3 similarly governs the use of message contents in determining the flow of control within a process in a DYMOL model. (See [2] for details on these constraining languages.)

In the setting of the constraint set $CS=C_1\ U\ C_2\ U\ C_3$, analysis of a design modelled in DPMS proceeds in two stages. First, the effective procedure is applied to the model, translating it from a DYMOL description into a constrained expression. This procedure, defined in detail in

(a) The shuffle operation was first defined and studied by Ginsburg [11], while Riddle presents a thorough formal discussion of both of these concurrency operators in [5].

(b) Proofs that expressions of this form represent regular languages may be found in [11], [5], and [8].

[2], is very similar to the translation performed by a compiler, being based upon translation rules associated with each syntactic construct of DYMOL. The result of this translation is an expression consisting of message type names (the elements of the event alphabet E in this setting), symbols from S and the constrained expression operators. The second stage of analysis involves inspection of the language represented by the derived constrained expression. Specifically, strings in the language that correspond to either desirable or undesirable system behaviors are sought. While no completely general algorithmic approach to this search is possible, it nevertheless can often result in useful information to guide the designer of a dynamically-structured distributed system. Several examples, such as those in [1]-[3], have demonstrated the value of this technique.

Examples

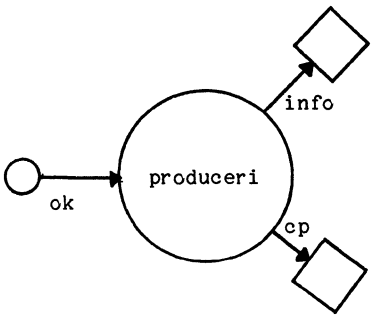
Although space limitations preclude a fully detailed treatment, we offer the following excerpts from an example in [2] to illustrate various facets of the preceding discussion.

This example concerns a DPMS model of a distributed system with dynamic connectivity, namely a producer-consumer situation in which a producer generates information that can be processed by either of two consumers. The producer in the modelled system generates a stream of variable-length information packets, each packet postfixed with a termination indicator. It is intended that each packet constitute a single complete, coherent set of data for a consumer. Therefore, proper processing requires that each complete packet, including its termination indicator, be received by exactly one consumer. Each time that the producer is prepared to generate an information packet, a manager process called c_pool selects a consumer to receive the information. When the producer has completed the generation of a packet, c_pool is notified to disconnect the producer from the most recently active consumer.

Figure 1 gives the DYMOL description of the producer process of this model. (See [1] for details on DYMOL and [2] for a complete version of this example model and its analysis.) Following each sending of the 'ready' message indicating its intention to generate an information packet, the producer (at p4) awaits an indication that a consumer is prepared to receive it before actually commencing generation of the information. The consumer's confirmation of each reception of a 'goods' message is used by the producer (at p8) to ensure that the items in an information packet are consumed in the same order in which they were produced. The producer's 'term' message (p9 and p10) is the packet termination indicator sent to the consumer, while the 'done' message (p11 and p12) informs the consumer pool manager, c_pool, that a complete packet has been generated.

The result of applying the constrained expression translation procedure to the DYMOL description of the producer process is shown in Figure 2. This is the subexpression describing the unconstrained possible behavior of this process. Figure 3 shows the complete constrained expression for the full modelled system, including

```
producer:  p1:  WHILE INTERNAL TEST DO
                   BEGIN
           p2:       SET BUFFER := ready;
           p3:       SEND cp;
           p4:       RECEIVE ok;
           p5:       WHILE INTERNAL TEST DO
                       BEGIN
           p6:           SET BUFFER := goods;
           p7:           SEND info;
           p8:           RECEIVE ok;
                       END;
           p9:       SET BUFFER := term;
          p10:       SEND info;
          p11:       SET BUFFER := done;
          p12:       SEND cp
                   END.
```



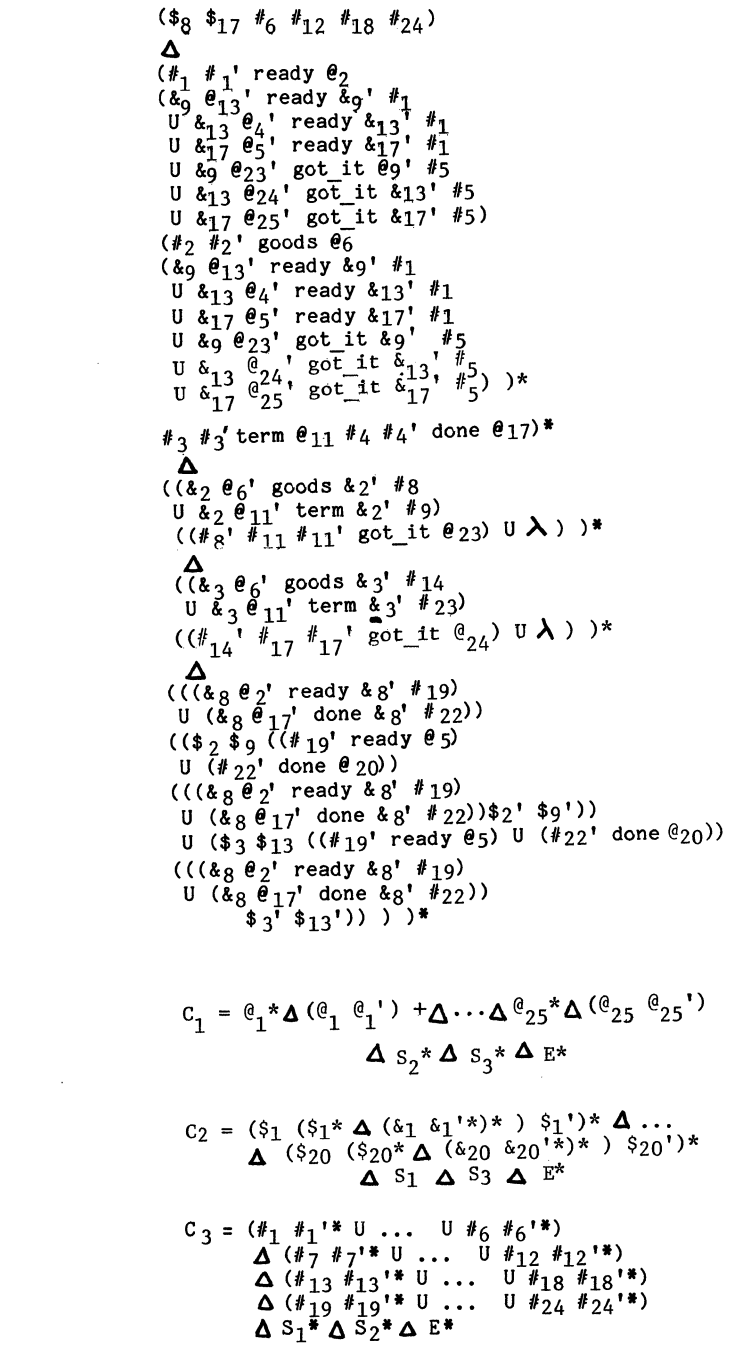
Producer Process

Figure 1

$(\#_1\ \#_1'\ \text{ready}\ @_2$
$(\&_9\ @_{13}'\ \text{ready}\ \&_9'\ \#_1$
$\cup\ \&_{13}'\ @_4'\ \text{ready}\ \&_{13}'\ \#_1$
$\cup\ \&_{17}\ @_5'\ \text{ready}\ \&_{17}'\ \#_1$
$\cup\ \&_9\ @_{23}'\ \text{got\_it}\ @_9'\ \#_5$
$\cup\ \&_{13}\ @_{24}'\ \text{got\_it}\ \&_{13}'\ \#_5$
$\cup\ \&_{17}\ @_{25}'\ \text{got\_it}\ \&_{17}'\ \#_5)$
$(\#_2\ \#_2'\ \text{goods}\ @_6$
$(\&_9\ @_{13}'\ \text{ready}\ \&_9'\ \#_1$
$\cup\ \&_{13}\ @_4'\ \text{ready}\ \&_{13}'\ \#_1$
$\cup\ \&_{17}\ @_5'\ \text{ready}\ \&_{17}'\ \#_1$
$\cup\ \&_9\ @_{23}'\ \text{got\_it}\ @_9'\ \#_5$
$\cup\ \&_{13}\ @_{24}'\ \text{got\_it}\ \&_{13}'\ \#_5$
$\cup\ \&_{17}\ @_{25}'\ \text{got\_it}\ \&_{17}'\ \#_5)\ )^*$
$\#_3\ \#_3'\ \text{term}\ @_{11}\ \#_4\ \#_4'\ \text{done}\ @_{17})^*$

The Producer Process Subexpressions

Figure 2



Complete Derived Constrained Expression

Figure 3

the producer, the consumers, and c_pool. Now, using the interpretation rule for constrained expressions (1) it can be determined that one string contained in the language described by the Figure 3 constrained expression is:

ready ready ready ready goods goods got_it got_it
term done done ready ready ready ready goods
goods got_it got_it term done done term

The message sequence represented by this string corresponds closely to the expected sequence of message transmissions in the modelled system in a case where the producer sends two single-item information packets. A sequence of four[c] 'ready' messages would naturally appear as the producer process signalled c_pool of its intention to generate a packet and c_pool responded by indicating that a consumer was prepared to receive the information. The transmitted information ('goods') and the confirmation of its reception ('got_it') follow as expected, and finally the 'term' and 'done' messages indicate the completion of a packet transmission. The intermixing of the final four symbols of the string (in fact, the derived constrained expression permits any ordering of the last three symbols in this behavioral representation) indicates that the receiving of the 'term' and 'done' messages by a consumer and the c_pool process, respectively, are potentially concurrent events.

Examination of the string reveals one unexpected aspect of the modelled system's message transmission behavior, however. The string contains only three 'term' symbols, indicating that one information packet termination indicator was not received by a consumer process, although it was sent by the producer. This is an unacceptable situation under our previously-stated assumption that a complete packet, including the termination indicator, should be received by a single consumer each time such a packet is generated. Yet the fact that this string is a string in the interpreted language of the derived constrained expression indicates that this unacceptable behavior can be realized by the system as currently modelled. To the software system designer contemplating this DPMS model as a possible design for the producer-consumer system, this would presumably indicate that the proposed design was faulty. In fact, examination of the Figure 3 constrained expression can lead to discovery of the source of the difficulty. This is done in [2], where a revised version of the DYMOL design is then presented. Performing the constrained expression analysis on the revised design demonstrates that the difficulty has indeed been corrected.

[c] Each completed communication of the modelled system generates two symbols in the message sequence describing its behavior. One represents the movement of the message from sender into the message channel while the other signifies the message's movement from the message channel to receiver. Under the semantics of DPMS, these events can be arbitrarily separated in time, hence the corresponding symbols need not, in general, occur adjacent to one another in the string.

Conclusion

In this paper we have introduced the formalism of constrained expressions and suggested how constrained expressions can be used in analyzing designs of dynamically-structured distributed systems. As our example indicates, however, the derived constrained expressions can be long and unwieldy. Therefore, analysis based upon manually manipulating and inspecting the derived expressions is likely to be incomplete and error prone. At the very least, one would like to have automated tools for generating example strings from the language represented by a given constrained expression. Such tools would simplify the inspection approach and reduce the tedium and likelihood of errors. Fortunately, such tools are quite straightforward to build. Ideally, one would like formal techniques, more powerful than simple inspection, for uncovering anomolies in the behavior described by a constrained expression. We are presently working on developing such formal techniques [3]. Our approach involves deriving a system of inequalities from a constrained expression description, then attempting to solve the system of inequalities. This process formally demonstrates the presence or absence of certain types of behaviors in the modelled distributed system. Examples of this technique appear in [3] and [10].

REFERENCES

[1] J. Wileden, "Techniques for Modelling Parallel Systems with Dynamic Structure," Journal of Digital Systems, 4,2, (Summer 1980), pp.177-197.

[2] J. Wileden, "Modelling Parallel Systems with Dynamic Structure," Department of Computer and Information Science, University of Massachusetts, COINS Technical Report 78-4, (January 1978).

[3] G.S. Avrunin and J.C. Wileden, "Algebraic Techniques for the Analysis of Concurrent Systems," Department of Computer and Information Science, University of Massachusetts, COINS Technical Report 81-11, (May 1981).

[4] A. Shaw, "Software Specification Languages Based on Regular Expressions," in Software Development Tools, Springer-Verlag, Heidelberg, (1980), pp.148-175.

[5] W. Riddle, "An Approach to Software Behavior Description," Journal of Computer Languages, 4, (1979), pp.29-47.

[6] R. Campbell and A. Habermann, "The Specification of Process Synchronization by Path Expressions," in Lecture Notes in Computer Science, 16, Springer-Verlag, Heidelberg, (1974), pp.89-102.

[7] A. Shaw, "Software Descriptions with Flow Expressions," IEEE Transactions on Software Engineering, SE-4, 3 (May 1978), pp.242-254.

[8] M. Welter, "Counter Expressions," RSSM/24, Dept. of Computer and Communication Sciences, University of Michigan, Ann Arbor, Michigan (October 1976).

[9] A. Aho and J. Ullman, The Theory of Parsing, Translation and Compiling, Prentice-Hall, Englewood Cliffs, N.J., (1972).

[10] G.S Avrunin and J.C. Wileden, "Describing and Analyzing Distributed System Designs," Department of Computer and Information Science, University of Massachusetts, COINS Technical Report 82-2, (January 1982).

[11] S. Ginsburg, The Mathematical Theory of Context-Free Languages, McGraw-Hill, New York, (1966).

ANALYSIS AND MODELING OF A SPLITTED-BUS DISTRIBUTED MULTIPROCESSOR SYSTEM

Lan Jin Wei-min Zheng
Department of Computer Engineering and Science
Tsinghua University, Beijing, China

Summary

A two-dimensional distributed multiprocessor system structure based on the concept of splitted bus was proposed in [1]. It was noted that introduction of switches into bus structure makes the multiprocessor systems highly flexible and cost-effective. The probability of bus contention can be reduced, thus resulting in better line utilization and shorter message delay time. The ease of routing control by means of switches helps in organization of reconfigurable and partitionable distributed systems.

The one-dimensional structure based on splitted-bus concept proposed in this paper looks like a wheel, as shown in Fig.1. It consists of a data loop and star-shaped control links between the centralized routing controller and processor nodes. The data loop is fully duplex and can be splitted into segments by the routing switches, so that separate communication paths can be established at once for several distinct pairs of nodes to transfer variable-length messages so long as these paths do not conflict with one another. Three switches are needed for each processor node: one central switch for segmentation of the data loop, and two side switches for connecting the two ports of the processor to the loop across the central switch. Such an arrangment makes it possible to realize different modes of interconnection: broadcasting (one-to-all), shifting (every node to its neighbor by any modulo count), and random (multiple one-to-one). All these modes of interconnection are useful for parallel computations on a multiprocessor system. Furthermore, the installation of three switches for each node permits all or any number of processors to be isolated and operate independently without affecting the normal operation of the rest of the system.

For purpose of analysis, we establish two models for the system: one for the control processor, and the other for the data loop.

The job arrival process is modeled by two waiting queues in the control processor. The queue A accepts the message-communication requests from all nodes and delivers them one by one to find the desired interconnection paths on the basis of First-come-first-serve discipline. If the control processor fails to find path for some job, then this job enters the second queue B, which is given a higher priority than the queue A. The arrival of jobs to the queue A assumes a Poisson distribution with mean arrival rate $N\lambda$ packets per second, where N is the total number of processor nodes on the loop, and λ is the mean delivering rate of packets from each node. All the nodes are assumed to be identical. The message length per packet assumes a negative exponential distribution with mean length 1/u bytes per packet.

The model of the data communication loop consists of N separate models of the nodes, each of which is composed of receiving buffers, transmitting buffers, and the data link i connecting two adjacent nodes i and (i-1). Let R_k be the mean arrival rate of messages delivered from all nodes and passing through data link k. It can be shown that

$$R = R_k = \frac{\lambda}{u}\frac{N^2}{4}\frac{1}{N-1} \text{ bytes/sec.}$$

and is independent of k for all $0 \leq k \leq N-1$.

Let the maximum transfer rate of the communication line be V bytes/sec, then the bandwidth utilization factor is equal to

$$BU = \frac{R}{V} = \frac{\lambda N^2}{4uV(N-1)}$$

The inverse ratio of the number of node pairs which occupy the data link k during message transfers between them to the total number of node pairs which require communications in general is

$$n = \frac{N(N-1)}{N^2/4} = \frac{4(N-1)}{N}$$

which represents the number of communication jobs that can be served simultaneously by the loop in the limit of its maximum bandwidth. It is a measure of parallelism [2] of the system, and is nothing else but the number of servers of the system viewed from queueing theory.

To estimate the total message delay time T, we neglect the time spent in repeatedly examining the queue B, and thus obtain

$$T = \frac{\rho_1 Z_1}{2(1-\rho_1)} + Z_1 + \frac{\rho_2 Z_2}{1-\rho_2} + Z_2$$

where Z_1 is the constant service time of queue A,
Z_2 is the mean service time of data loop and equals to 1/uV sec,
$\rho_1 = N\lambda Z_1$, and
$\rho_2 = N\lambda Z_2/n$

The formulae obtained above for calculating can also be applied to the cases of single-port splitted-bus systems as well as integrated-bus systems. The only difference exists in the value of n. For integrated-bus systems, n is always equal to 1, whereas for single-port splitted-bus systems, the value of n can be found as

$$n = \frac{N(N-1)}{(N^2/4)+2N-2} = \frac{4N(N-1)}{N^2+8N-8}$$

Different values of n for the systems under comparison are listed in the following table:

0190-3918/82/0000/0345$00.75 © 1982 IEEE

Number of nodes	2-port splitted-bus loop system	1-port splitted-bus loop system
6	3.33	1.58
8	3.5	1.87
10	3.6	2.09
12	3.67	2.28

The formula for bandwidth utilization is rewritten in terms of n for all cases as below:

$$BU = \frac{N\lambda}{u\sqrt{n}}$$

A series of simulation experiments have been performed for 2-port splitted-bus, 1-port splitted-bus, as well as integrated-bus systems. The same assumptions and conventions were made as in theoretical analysis. The calculated and the experimental curves are shown in Fig.2. They coincide rather satisfactorily.

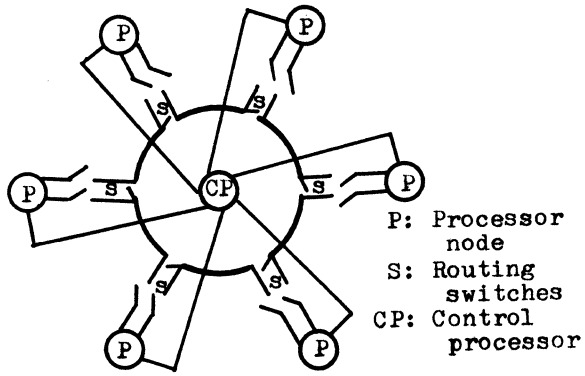


Fig.1 Splitted-bus wheel-structured system

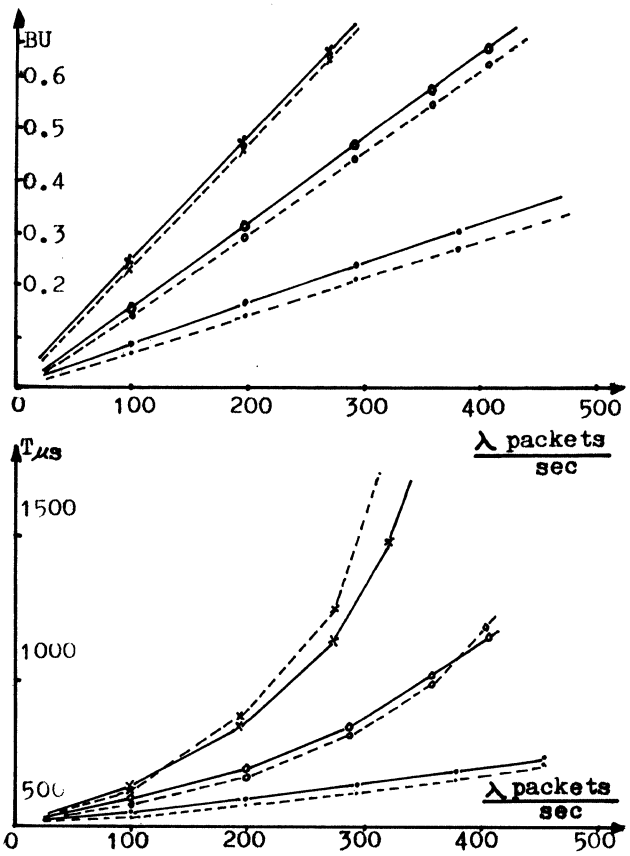


Fig.2 Bandwidth utilization factor BU and total transmission time T vs mean message arrival rate curves
N=6, 1/u=50 bytes/packet, V=125 Kbytes/sec.
—— experimental ------ theoretical
×: integrated-bus system
∘: single-port splitted-bus system
•: double-port splitted-bus system

References

1. Lan Jin, "A New General-purpose Distributed Multiprocessor System Structure", Proc. of the 1980 International Conference on Parallel Processing, pp.153 - 154.

2. Lan Jin, D. Wang, and M. Sheng, "Parallel Processing Computer Architecture", (in Chinese) Guofang Gongye Publisher, Beijing, China, 1982

3. H. Kobayashi and A. G. Konheim, "Queueing Models For Computer Communications System Analysis", IEEE Trans. on Comm., COM-25 (Jan. 1977), pp. 2 - 29.

4. W. W. Chu and A. G. Konheim, "On The Analysis and Modeling of a Class of Computer Communications System", IEEE Trans. on Comm., COM-20 (June 1972), pp.645 -660.

5. J. Spragins, H. Jafari and T. Lewis, "Some Simplified Performance Modeling Techniques with Applications to a New Ring-structured Microcomputer Network", Proc. of the 6th Annual Symposium on Computer Architecture (April 1979), pp.111 - 116.

6. H. Jafari, J. Sprins, "Simulation of a Class of Ring-structured Networks", IEEE Trans. on Computers, vol.C-29 No.5 (May 1980), pp. 385 - 392.

7. H. Jafari, and T. Lewis (1977). "A New Loop Structure for Distributed Microcomputing Systems". 1st Ann. Rocky Mountain Symp. on Microcomputers: Systems, Software, Architecture, pp. 121 - 141.

LOGIC PROGRAMMING ON ZMOB: A HIGHLY PARALLEL MACHINE

U.S. Chakravarthy, S. Kasif, M.Kohli, J. Minker, D. Cao
Department of Computer Science
University of Maryland
College Park, MD, 20742

Abstract -- This paper proposes a framework for implementing a logic programming environment on a distributed system. ZMOB is one such multiple microprocessor architecture where the individual microprocessors are connected by a high speed conveyor belt. The paper describes how the logic programming environment is created on ZMOB. This enables us to exploit the high level of parallelism possible in logic programming. The approach and preliminary design raises several relevant issues in distributed parallel processing environments which are currently being investigated.

1. Introduction

The introduction of logic as a conceptual and applicative aid in the derivation of programs ([8]) has resulted in the implementation of the logic language PROLOG ([2], [6], and [7]). PROLOG interpreters have been designed to be executed on a sequential machine i.e., on single processor architectures. However the complete separation of logic (the specification of the statements of the problem to be executed) and control (the order of execution of statements) allows several degrees of freedom during the interpretation of the program, thus providing a natural parallelism that could be implemented on a parallel architecture system ([4]).

The inherent nondeterminism available during a logic program execution can be exploited in the following directions:

1. At any time during execution more than one goal node may be selected.
2. Many literals in a clause can be expanded upon.
3. Many procedures can be invoked for a procedure call.

Logic programming is distinguished from other applicative programming languages such as LISP due to the fact that more than one procedure can match a procedure call. This seeming disadvantage on a sequential machine can be exploited in a highly parallel environment, as some or all matching procedures can be invoked simultaneously.

We shall discuss the design considerations for implementing a PROLOG-like language on a highly parallel system called ZMOB, at a high level. In our initial design the problems associated with the parallel execution environment are made transparent to a class of users who do not want to know about it. For a detailed functional specification and design considerations of ZMOB as a parallel problem solving system see [1] and [2]. Because of length restrictions we assume that the reader is familiar with Horn clauses and logic programming. See [4] for details.

2. ZMOB Configuration and Architecture

ZMOB is multi-microprocessor, distributed memory architecture with a high speed "conveyor belt" communication facility among the microprocessors, and a host mini-computer. There can be a maximum of 256 microprocessors in the system each with a memory capacity of 64K bytes. There are 257 bins (including one for the host machine) which go around on the conveyor belt carrying information from processors to which they are attached. Ideally speaking the conveyor belt is fast enough to service individual microprocessors at the speed at which they can access their core memory. For more details refer to [5].

Communication among the processors can take place either on a point to point basis or by broadcast, wherein a single microprocessor can communicate with a set of microprocessors on the belt. This broadcast mode of message passing is achieved by a pattern matching capability at each processor instead of a destination oriented transmission. An exclusive source mode is available for communicating on a point to point basis without repeated handshaking. Each processor has a mail stop, which handles the interrupt, receiving and transmission of information to and from the bins.

3. ZMOB as a Problem Solver

In this section we first describe the decomposition of ZMOB into its functionally independent components. Then we present the rationale for this decomposition and describe the conceptual specification of each component.

There are five clusters of microprocessors and the host machine in the ZMOB problem solving system. They are:

1. The VAX-host machine.
2. A set of machines dedicated to problem solving, termed Problem Solving Machines (PSMs).
3. A set of machines dedicated to answering function free ground assertions, the Extensional Database (EDB) machines.
4. A set of machines dedicated to servicing general rules or axioms, termed Intensional Database (IDB) machines.
5. An IDB monitor supervising the IDB machines.

The assertions and the axioms of the logic program will be distributed among the IDB and EDB microprocessors and the actual problem solving carried out on several PSM microprocessors simultaneously. Figure 1 shows a small logic program and its distribution among different clusters of ZMOB.

3.1 Problem Solving Machines (PSMs)

The role of the PSMs is central to the entire problem solving process. The capabilities of the PSM permit the inherent parallelism available in a logic program to be exploited in its entirety.

The PSM manages the search space. Initially the goal node is placed in a PSM. In the process of attempting to solve the goal node, a PSM has the capability to dynamically create new PSMs which can independently develop and manage the subtrees of the search space. New PSMs are initiated if there is non-determinism associated with solving an atomic goal. Each PSM is autonomous except for the knowledge of the parent-child relationship and can, in turn, perform the same operations. In addition to managing the goal tree and dynamically initiating the new PSMs, the PSM selects a subgoal of a conjunction of goals to be solved, and selects a clause in the goal tree to be expanded next. When a goal assigned to a PSM is completely solved it transmits the solution to its parent PSM. The parent of the initial PSM is the VAX machine.

Each PSM interacts with IDB machines to obtain procedure bodies to generate new nodes in the goal tree. It also interacts with EDB machines to solve atomic goals and to generate new nodes in the goal tree. A PSM can work on more than one path of the search tree while it is waiting for a unifier and procedure body to be returned on some other path from either the EDB or the IDB.

3.2 Intensional Database (IDB) Machines

We assume that the IDB is relatively small and that each Z80A can effectively contain the set of all procedure clauses of the program. This assumption allows us to replicate the IDB on several machines referred to as IDB machines. Each IDB machine contains the same information, thus making the existence of several IDB machines transparent to the PSMs. This replication of IDB machines is advantageous on account of the

0190-3918/82/0000/0347$00.75 © 1982 IEEE

"broadcast by pattern" facility available on ZMOB.

Whenever the PSM sends an atom for expansion to the IDB, the first "idle" IDB machine could pick up the request and process it. In an extension to this system we plan to consider the handling of the IDB when the set of procedure clauses exceed the capacity of any one machine in the system. The IDB could then be distributed on a set of machines in much the same way as is done for the EDB, still making it transparent to PSMs. An IDB machine finds all matching procedures requested by the PSMs, returning all matching procedure bodies at once, or one at a time.

3.3 Extensional Database (EDB) Machines

The EDB is the union of all relational tables containing ground, function-free assertions.

We identify a relation with an unique integer number obtained from the relation name. This permits a relation to be readily relocatable onto any EDB machine, thus creating a virtual addressing facility. The atoms that belong to the EDB in the procedure definition are specially marked to denote they may be found in the EDB, the IDB or in both. Thus PSMs send only the valid requests to the EDB and IDB machines. When the request for matching some atom P(...) is sent to the EDB, one of the machines that contains the relation "P" picks up the request. When a relation is distributed among several EDB machines, the operation is still similar, as one of the EDB machines containing the relation acts as a supervisor, thus making it transparent to the PSMs.

3.4 The Host Machine and the IDB Monitor

The VAX 780 serves as the host machine and provides an interface between the user and the ZMOB. The user loads ZMOB executable code via the VAX and initiates processing. As noted earlier, VAX shields the user from the parallel environment of ZMOB.

The IDB monitor is a simple version of a monitor and exists in the system to avoid certain deadlocking and overloading conditions. When all IDB machines have their buffers full, no more requests may be sent to the IDB machines. The IDB monitor keeps track of the availability of buffers in IDB machines and when an overload condition is detected, it informs all PSMs to take corrective action. PSMs, in turn, request responses from the IDB machines without making new additional requests until buffers become available in IDB machines. This situation is detected by the IDB monitor and broadcast to all PSMs.

3.5 The Rationale for the Approach

There are many possible ways in which one could design a parallel logic problem solving system on ZMOB. The design of the system that we have outlined above was arrived at through several iterations and has numerous advantages.

The system is modular in design in that each processor is dedicated to a specific task: problem solving, procedure management, assertion management, or monitoring. Hence each microprocessor can operate independently of other microprocessors.

The autonomous nature of each PSM in developing its search tree enables it to use necessary heuristics in the generation of the search tree.

The conceptual treatment of procedures and assertions is handled uniformly. There is no essential difference between the EDB and the IDB, as they both serve as distributed memory to the system. The distinction was made because of different unification algorithms employed and potential size differences. In database applications the size of the IDB is likely to be significantly smaller than the EDB portion. IDB procedures may contain functions, whereas EDBs contain only function-free ground atomic formulae.

The fact that an unique operation is performed by each processor leads to speed up possibilities such as preprocessing, and preparation for next request at idle time.

The IDB features the principle, that information is not returned unless requested. This allows us to use the IDB as a temporary buffer for the PSM and to perform local heuristic ordering on the procedures returned by the IDB machine.

The system is inherently flexible and allows the user to choose a particular configuration (i.e., the number of EDB, IDB and PSMs) especially suited for his purpose.

The relocatable nature of the usage of EDBs and IDBs permits them to be loaded onto any processor in the system.

The functional independence of the system components increases the reliability of performance.

4. Communication Among ZMOB Components

Any distributed processing environment requires that information be exchanged between its elements, and hence communication is needed between the elements. In an environment like ZMOB where the same problem is being distributed among several processors there is a need for information exchange to carry out the problem solving activity. Hence the need for communication primitives and message formats to keep communication overhead low and to maintain a flexible system.

Two features of the ZMOB architecture, namely the broadcast facility using a pattern, and the exclusive source mode of communication, have been useful in designing the primitives necessary for problem solving. The broadcast facility allows a single PSM to converse with a set of IDB and EDB machines and the exclusive source mode permits large amounts of data to be transferred between two machines without handshaking overhead.

The ZMOB components constantly interact among themselves and the VAX for solving a problem. Communication primitives have been defined to carry on the interaction in an efficient and flexible manner.

5. User Interface and Control

Facilities for the user to interact with the problem solver (machine) is an essential and integral part of the overall design aspect of any system.

VAX acts as the external interface to ZMOB. The user accesses ZMOB as a resource from the host machine and VAX provides a smooth interface relieving the user of the details of ZMOB aspects of problem solving.

The user can be in three different states of interaction as a logic problem solver. They are: the off-line mode, the active mode, and the execution mode.

In the off-line mode the user is, in general, creating a logic program (or deductive relational database as the case may be) by using the facilities available on the host machine. These operations are performed outside the ZMOB utilization, as they do not need the ZMOB capabilities and during these operations ZMOB is potentially free to be used by others.

The active mode of operation contrasts with the off-line mode in the usage of specific support software developed for ZMOB and is executed on the host machine. This phase is employed in preparing for the execution mode by compiling and converting the source code to ZMOB compatible version. The user has control over the configuration and other aspects of ZMOB in this mode.

The execution mode is the phase in which the user is actually using ZMOB for problem solving in

its full capability (though transparent to the user) with VAX acting as the interface. In this mode the VAX channels the user query (or the goal) and other user interaction to ZMOB components after checking for syntax and sequencing and formatting them if necessary. Also all communication from the ZMOB components to the user are channeled through VAX.

The user is to be provided with facilities to trace/debug his logic program, as well as to gather specific statistical information from the system. The statistical information gathered can be used to fine-tune the system to specific needs and evaluate the performance of the system under various conditions.

Apart from the interaction to create, compile, execute and debug logic programs, the user is to be provided with the facility to specify and guide the control aspects of logic problem solving. The user can exercise several degrees of freedom in the choice of the control of a logic program. Within a clause literals can be selected ranging from left to right literal selection (i.e., depth first tree generation) as in PROLOG, to a completely arbitrary literal selection. The user can specify these through the syntax of the axioms of the logic program.

6. Summary and Scope

The ZMOB configuration permits a high degree of parallelism to take place in solving a problem. The use of logic programming as a formalism permits the exploitation of the parallel capability without forcing the user to rewrite his program to account for the inherent parallelism that can take place. The separation of the logic of the specification of the program from the control permits this to be achieved. Thus the system being designed will permit a problem to be executed on a single machine or on multiple machines within the ZMOB configuration.

We know of no attempts to exploit parallelism in programs based upon a specification of the program in logic. We know of no other approaches similar to ours using other computer configurations.

The work described is of considerable interest for problems which inherently have a high degree of parallelism. These problems arise in artificial intelligence and in database systems. Sequential computations, although possible to run in the system would undoubtedly be executed much slower in our approach.

Once the system is implemented, there is much that must be done. Its effectiveness must be evaluated. Simplifications made for this first system must be removed. A more flexible control capability should be provided to the user so that advantage can be taken of his knowledge of the problem. Modifications to PROLOG-like languages will be necessary to enable communications between machines to take place. Considerations must also be given to operating system problems and to understanding the optimum configuration needed to solve a problem. Work on large artificial intelligence problems and databases would be achieved by the availability of disks and drums on the Z80A machines. Thus a wealth of research topics remain to be accomplished. Work on parallelism and logic programming is in its beginning stages.

7. Acknowledgements

Work on this subject was supported by NASA grant NAG 1-51 and by the NSF grant MCS-7919418. We gratefully appreciate their support which made this work possible. We also would like to express our appreciation to Charles Asper and Norbert Eisinger who have also contributed to the effort.

References

[1] C. Asper, D. Cao, U.S. Chakravarthy, S. Kasif, M. Kohli, and J. Minker, Functional Specifications of ZMOB Problem Solver, Department of Computer Science, University of Maryland, Under Preparation.

[2] K.L. Clark, and F. McCabe, Programmers' Guide to IC-PROLOG, Computer Science Department, Imperial College, London, CCD Rep. 79/7, (1979).

[3] N. Eisinger, S. Kasif, and J. Minker, Logic Programming: A Parallel Approach, Department of Computer Science, University of Maryland, College Park, Technical Report 1124, (December,1981), 44 pp.

[4] R.A. Kowalski, Logic for Problem Solving, North-Holland, (1979), 287 pp.

[5] C. Rieger, J. Bane, and R. Trigg, ZMOB: A Highly Parallel Microprocessor, Department of Computer Science, University of Maryland, Technical Report 911, (May,1980), 22 pp.

[6] P. Roussell, PROLOG: Manuel de Reference et d'Utilisation, Groupe d Intelligence Artificielle, Universite d'Aix-Marseille, Luminy, (September,1975).

[7] D.H.D. Warren, Implementing PROLOG: Compiling Predicate Logic Program, Department of Artificial Intelligence, University of Edinburgh, Research Reports 39 and 40, (1977).

[8] M.H. Van Emden, and R.A. Kowalski, "Logic as a Programming Language", J. ACM, Vol 23, No 4, (1976), pp. 733-742.

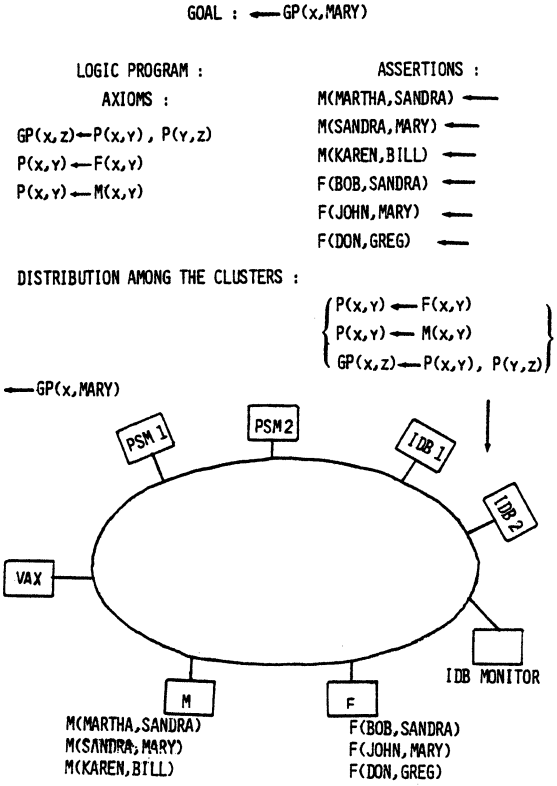


FIGURE 1.

SYSTEM ARCHITECTURE OF A RECONFIGURABLE MULTIMICROPROCESSOR RESEARCH SYSTEM

Vito A. Trujillo
Computing Division
Los Alamos National Laboratory
Los Alamos, New Mexico 87545

Summary

This paper presents the architecture of an experimental multiprocessor system that incorporates a reconfigurable array of microprocessing and memory elements. The system is designed specifically as a research tool for implementing and evaluating parallel-processing algorithms on various multiprocessor architectures. Consequently, the principal design objective is to provide a multiprocessor with fully reconfigurable processor-to-memory and processor-to-processor interconnections in order to allow direct comparison of algorithms for a wide range of multiprocessor architectures. Basically, the system is a tightly coupled, shared-memory MIMD machine [1-2] that supports reconfiguration between processor and memory nodes to permit experimentation with common memory architectures and with various processor network structures such as rings, trees, and stars. This experimental computer system is currently under development within the Computing Division at Los Alamos National Laboratory.

As illustrated in Figure 1, the Multimicroprocessor Research System consists of numerous processor and memory nodes that are directly interconnected using multiple processor-to-memory buses and multiported global memory nodes. The multiple bus/multiported memory arrangement functionally implements a full crossbar switch between the processor and memory nodes [3]. This multiple-bus architecture allows processor-to-processor communications to occur concurrently

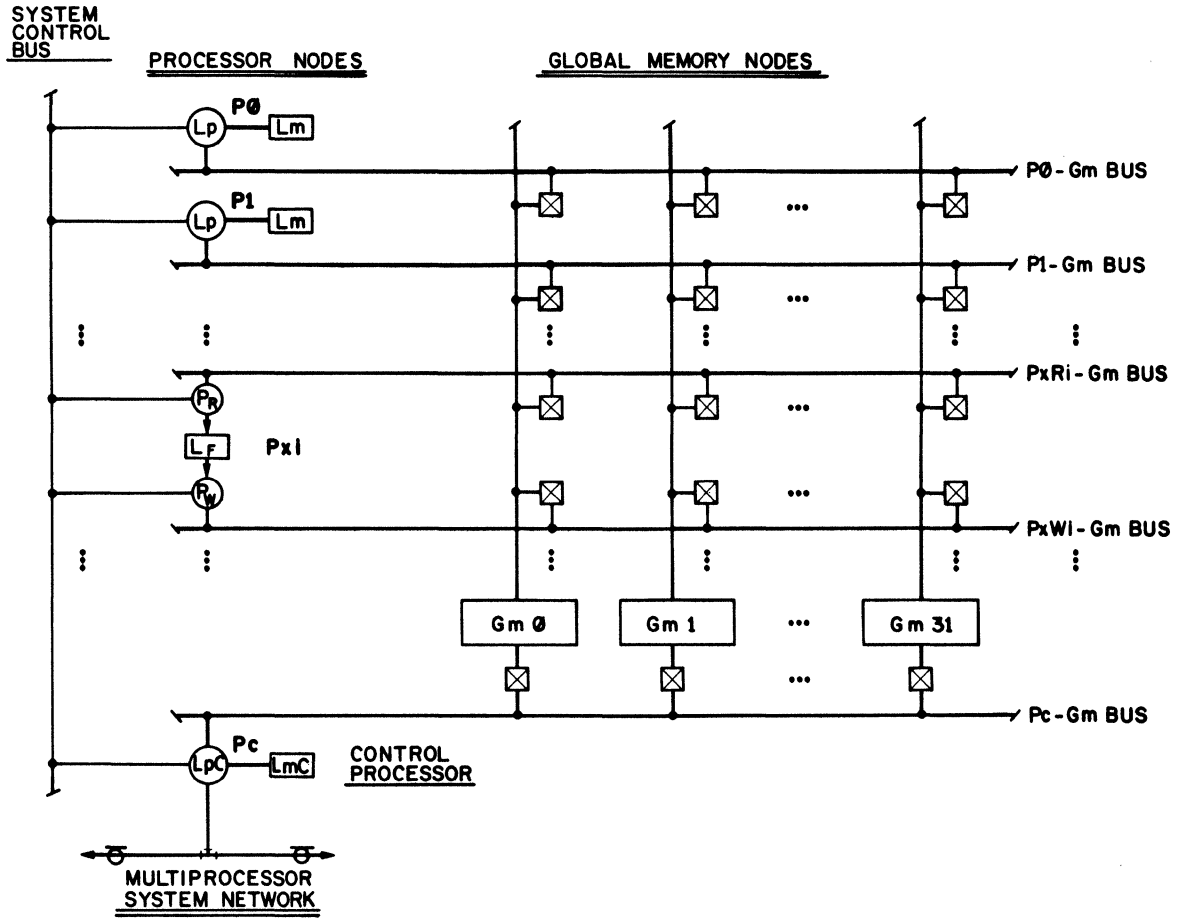


Fig. 1. Multiprocessor Research System system architecture.

with processor execution from either local memory or global memory.

Three types of processor nodes are included within the system: (1) system control processor, (2) general floating-point processors, and (3) dedicated data transfer processors. The system control processor performs system initialization (downloading of global memory, configuration control, etc.), initiation of parallel processing applications code, performance measurements, memory error processing, etc. In addition, because the multiprocessor is strictly an execution environment, the system control processor provides communication with an external local area network that includes development workstations. Each general floating-point processor includes Intel iAPX 86/87 microprocessing elements [4], 48k bytes of local dedicated ROM/RAM, real-time interrupt facility, and memory mapping logic that allows sixty-one 16k-byte memory segments to be permanently and/or dynamically allocated within the system global memory. Each data transfer processor is a high-speed controller specifically designed for implementing processor-to-processor communications by performing data movement between global memory segments.

The system global memory consists of multiple memory nodes, each having a 256k-byte RAM array accessible from the system control processor and a multiported memory controller. The port for the system control processor supports downloading and memory error reporting functions. The multiported memory controller includes interface logic for 20 ports, memory arbitration logic that implements a last-granted-lowest-priority algorithm, and a high-speed memory access controller. Memory mapping logic within each processor node allows each memory node segment to be allocated as either private or public memory for each processor node.

The processor-memory interconnection is accomplished with memory mapping logic at each processor node, a multiported memory controller at each global memory node, and a multiple bus interconnection backplane that allows an orthogonal arrangement of processor and memory boards. As illustrated in Figure 2, an orthogonal packaging scheme uses minimal bus lengths in providing complete physical interconnection between processor and memory nodes. Basically, the processor-memory interconnection provides fully reconfigurable processor-to-memory connections, resolves access arbitration when multiple processors are simultaneously accessing a common global memory node, and supports mutual exclusion to shared memory. Control of shared memory is accomplished through an extension of the LOCK mechanism available with the iAPX 86/87 microprocessor [4].

Processor-to-processor communication is implemented indirectly through the processor-memory interconnection by specialized data transfer processors that perform data movement between global memory nodes. Each data transfer processor includes memory mapping logic similar to the general floating-point processors; consequently, these nodes can access any segment within system global memory. In addition, the data transfer processor nodes include high-speed control, buffer, and translation logic that permit both contiguous and noncontiguous memory block transfers. The data transfer processors are controlled by linked structures within global memory and include maskable interrupt capability indirectly through the

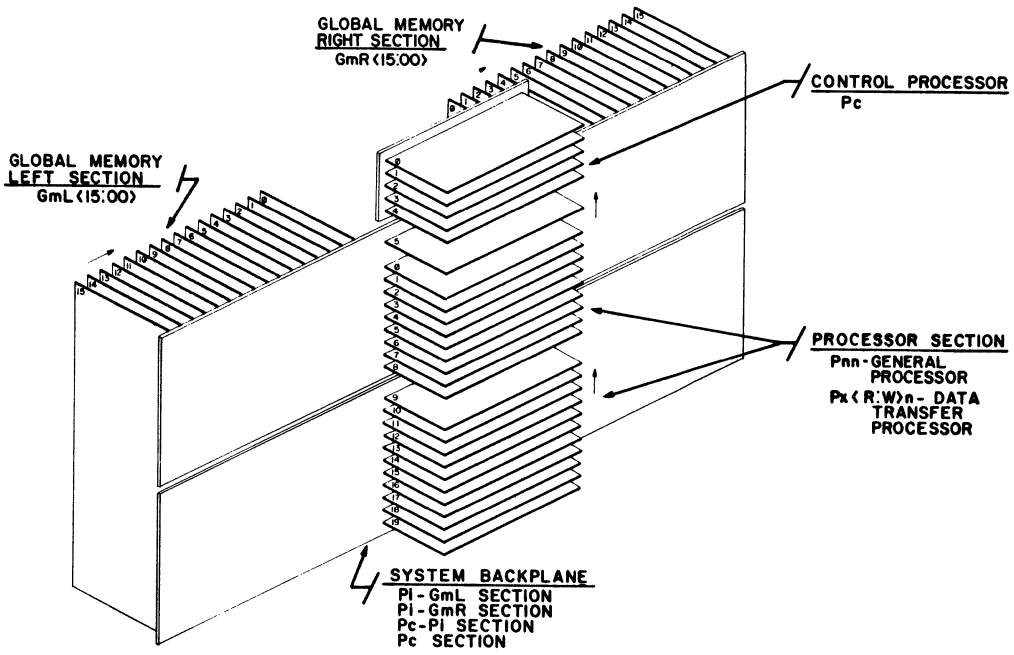


Fig. 2. Multiprocessor Research System orthogonal packaging diagram.

system-control processor. Functionally, the data-transfer processors can be visualized as multiple intelligent buses for interprocessor communications.

Currently, the Multimicroprocessor Research System accommodates a single system control processor, 20 processor nodes that can include either general floating-point processors or data transfer processors, and 32 global memory nodes. However, a typical maximum configuration consists of 16 general floating-point processors and 2 data transfer processors, which are sufficient for handling 16 iAPX 86/87 interprocessor communications.

References

1. M. Satyanarayanan, Multiprocessors: A Comparative Study, Prentice-Hall, Inc., Englewood Cliffs, N.J., (1980).

2. P. H. Enslow, Jr., "Multiprocessor Organization - A Survey," ACM Computing Surveys (March 1977), Vol. 9, pp. 103-129.

3. J. H. Patel, "Processor-Memory Interconnections for Multiprocessors," Proc. 6th Annual Symposium on Computer Architecture (April 1979), New York, N. Y., pp. 168-177.

4. Intel iAPX 86,88 User's Manual (August 1981), Intel Corporation, Santa Clara, CA.

DESIGN AND SIMULATION OF AN MC68000-BASED MULTIMICROPROCESSOR SYSTEM

James T. Kuehn
Howard Jay Siegel
Peter D. Hallenbeck

Purdue University
School of Electrical Engineering
West Lafayette, IN 47907, USA

Abstract -- The design of a multimicroprocessor system for image processing and pattern recognition applications utilizing the 16-bit Motorola MC68000 and other off-the-shelf components is described. This system can be dynamically reconfigured to operate in either SIMD or MIMD mode and can be used as a building block for the PASM partitionable SIMD/MIMD machine. The results of simulations of SIMD operation that were used to guide the design of the MC68000-based system are discussed. The possibilities for overlapped operation of the SIMD control unit and processors are examined. The system architecture, including hardware to interface the off-the-shelf components needed for SIMD/MIMD processing, is given. Finally, simulation studies of the performance of the proposed MC68000-based system are presented.

I. Introduction

The demand for higher throughput and very large database handling capabilities is forcing computer system designers to consider nontraditional architectures, notably distributed/parallel systems. Architects have proposed microprocessor-based large-scale parallel processing systems with as many as 2^{14} and 2^{16} processors [e.g., 9, 17] that show promise in meeting these data-handling and throughput demands.

Two types of parallel processing systems are SIMD and MIMD [4]. SIMD (single instruction stream - multiple data stream) machines (e.g., Illiac IV [3], STARAN [1]) typically consist of a set of N processors, N memories, an interconnection network, and a control unit. The control unit broadcasts instructions to the processors, and all enabled ("turned on") processors execute the same instruction at the same time. Each processor executes instructions using data from a memory with which only it is associated. The interconnection network allows interprocessor communication.

An MIMD (multiple instruction stream - multiple data stream) machine also typically consists of N processors and N memories, but each processor can follow an independent instruction stream (e.g., C.mmp [22], Cm* [18]). As with SIMD architectures, there is a multiple data stream and an interconnection network.

A Multiple-SIMD machine is a parallel processing system that can be structured as one or more independent SIMD machines of varying sizes (e.g., MPP [8]). A partitionable SIMD/MIMD machine can be structured as one or more independent SIMD and/or MIMD machines of varying sizes (e.g., PASM [13]).

SIMD and MIMD parallelism has been shown to be applicable to a wide variety of image processing tasks [13, 14, 15]. In this paper, the SIMD mode is emphasized; however, the use of full processors and the overall organization of the system will also allow MIMD operation. The system to be presented could be used as a single SIMD machine, or as a building block for a multiple-SIMD machine, or a partitionable SIMD/MIMD machine (using the techniques described in [13]).

SIMD algorithm simulations for several machine configurations have been performed [5, 11]. These studies have examined the possibilities for overlapped operation of the control unit and processors. Overlapping can be improved as additional hardware (e.g., latches, buffers) is added at the interfaces of these components. Each hardware configuration represents a "case" for which relative run time performance of assembly language test algorithms was measured. The results of these simulations were used as a basis for the control unit/processor organization described in this paper. A design based on this organization and employing currently available off-the-shelf components is described and simulated.

This design work is motivated by two research projects at Purdue. One is the development and implementation of the PASM (partitionable SIMD/MIMD) multimicroprocessor system. The other is the study of the use of parallel processing for mapping applications.

In Section II, a model of the proposed SIMD/MIMD system is given. A summary of our earlier SIMD algorithm simulation studies and overlapping schemes is presented in Section III. In Section IV, the design of a multimicroprocessor system which incorporates Motorola MC68000 processors is described. The hardware organization of the control unit, processors, and additional support components is discussed in detail in Section V. It is shown that the interface logic for the microprocessors necessary for SIMD/MIMD processing will be minimal; thus the high cost of a custom VLSI design can be saved. Ideas for a prototype patterned on this design are given. Finally, results of simulation studies of the proposed MC68000-based machine are summarized in Section VI.

This research was supported by the Air Force Office of Scientific Research, Air Force Systems Command, USAF, under Grant No. AFOSR-78-3581, and by the Defense Mapping Agency, monitored by the United States Air Force Systems Command, Rome Air Development Center, under Contract No. F30602-81-C-0193. The United States Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation hereon.

0190-3918/82/0000/0353$00.75 © 1982 IEEE

II. Model of the Proposed SIMD/MIMD System

The basic system components of the proposed machine are a Control Unit (CU) (including its own memory), $N=2^n$ processors, N memory modules, and an interconnection network. The processors are microprocessors that perform the actual SIMD and MIMD computations. A memory module is connected to each processor to form a processor/memory pair, called a processing element (PE). The PEs are numbered (addressed) from 0 to N-1. The interconnection network provides a means of communication among the PEs.

In SIMD mode, the CU fetches instructions from its memory, executes the control flow instructions (e.g., branches), and broadcasts the data processing instructions to the PEs. The CU may coordinate the activities of the PEs in MIMD mode.

"Functional-block" models of the interactions of the CU, PEs, and network will now be presented. Later, the hardware used to implement each function will be described.

The CU's functions may be classified into six areas (consult Figure 1). The numbers in parentheses in Figure 1 correspond to the component classifications given below.

(1) The CU execution unit performs program flow operations (e.g., loop counting, branching).
(2) CU memory contains the SIMD instruction stream. It also provides data storage for the CU execution unit.
(3) The fetch unit fetches instructions from CU memory and routes them to the CU execution unit, the PEs (via the CU/PE interface), or to other specialized CU hardware.
(4) The CU/PE interface collects PE instructions and enable signals and broadcasts these to the PEs.
(5) The masking operations unit decodes and manipulates masks. Masks specify which PEs are to be enabled or disabled.
(6) Microprogrammed logic directs the operations of the fetch unit, masking operations unit, and other specialized CU hardware. Signals are generated for system control functions (e.g., "bringing up" the CU and PE execution units, initializing I/O devices).

A PE's functions include the following (consult Figure 2).

(7) In SIMD mode, the PE execution unit accepts instructions broadcast by the CU and performs computations that process the local (PE memory) data stream. In MIMD mode, instructions and data are fetched from PE memory.
(8) PE memory contains data for the SIMD mode operations of the PE execution unit. It also contains instructions and data for MIMD mode operations.
(9) The PE/network interface sends data and routing information to and accepts data from the interconnection network.
(10) The condition codes register stores the PE execution unit condition codes. The data condition select lines specify which bit or boolean function of bits in the register will represent the status of the PE.
(11) Logic controlled by the PE's enable/disable signal ensures that the PE executes no instructions and generates no network conflicts while disabled.

The interconnection network has the single task of transferring data among the PEs. It accepts data from the "source" PEs at its N input ports and routes the data to its N output ports, where it is accessible to the "destination" PEs. The Generalized Cube network, a network being considered for use in PASM for reasons discussed in [12], is assumed in the simulations. This network consists of n stages of switches and is controlled by routing tags.

In a serial processor, components 4, 5, 6, 9, 10, and 11 are either unnecessary or are meaningless. These functions comprise what is known as the "overhead due to parallelism." Well-designed CU/PE and PE/network interfaces can minimize this overhead by overlapping the operations of the CU, PEs, and network. Overlapping allows the CU, the set of PEs, and the network to perform their own tasks, synchronizing only when there is some information to be exchanged. Examples of overlapping are:

(1) the CU fetching the next instruction in the stream or executing CU instructions while the PEs are executing an instruction,
(2) the PEs executing an instruction while a set of data items is passing through the network, and
(3) the network passing more than one set of data items from input to output simultaneously.

In this paper, (1) is analyzed and simulated. (Aspects of (2) and (3) are discussed in [11], but are beyond the scope of this paper.)

In SIMD mode, all of the enabled PEs will execute instructions broadcast to them by the CU. A masking scheme is a method for determining which PEs will be active at a given point in time. An SIMD machine may have several different masking schemes.

The general masking scheme uses an N-bit PE enable vector to determine which PEs to activate. PE i will be active if the i-th bit of the PE enable vector is a 1, for $0 \leq i < N$. A mask instruction is executed whenever a change in the active status of the PEs is required. The Illiac IV, which has 64 processors and 64-bit words, uses general masks [16]. However, when N is larger, say 1024, a scheme such as this becomes less appealing.

The PE address masking scheme [10] uses a 2n-bit mask to specify which of the N PEs are to be activated. PE address masks are fetched from the instruction stream and sent to the masking operations unit to be decoded into a PE enable vector [13]. This vector is passed to the CU/PE interface to effect the change in status of the PEs. General masks are passed to the CU/PE interface unchanged by the masking operations unit.

PE address masks may be decoded and then manipulated by the masking operations unit. For example, decoding two PE address masks, "or"-ing them together, and using the result as the PE enable vector activates the union of the sets of PEs activated by each individual mask [13]. This implies that the masking operations unit can perform basic boolean operations on masks and can temporarily store a number of general and decoded PE address masks.

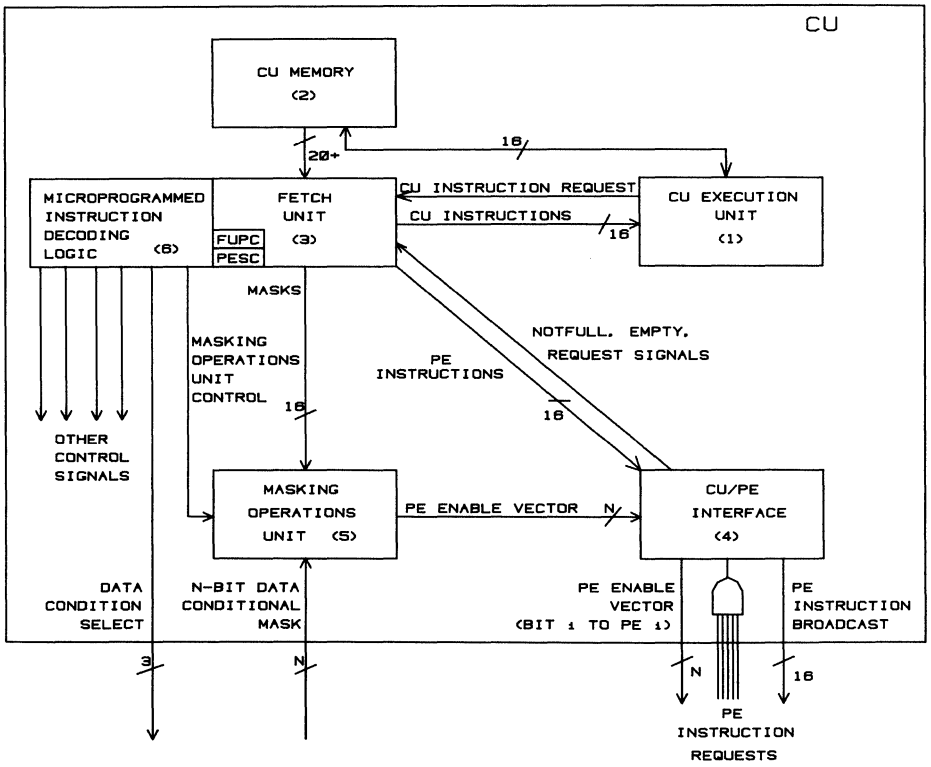


Figure 1. Model of the Control Unit (CU).

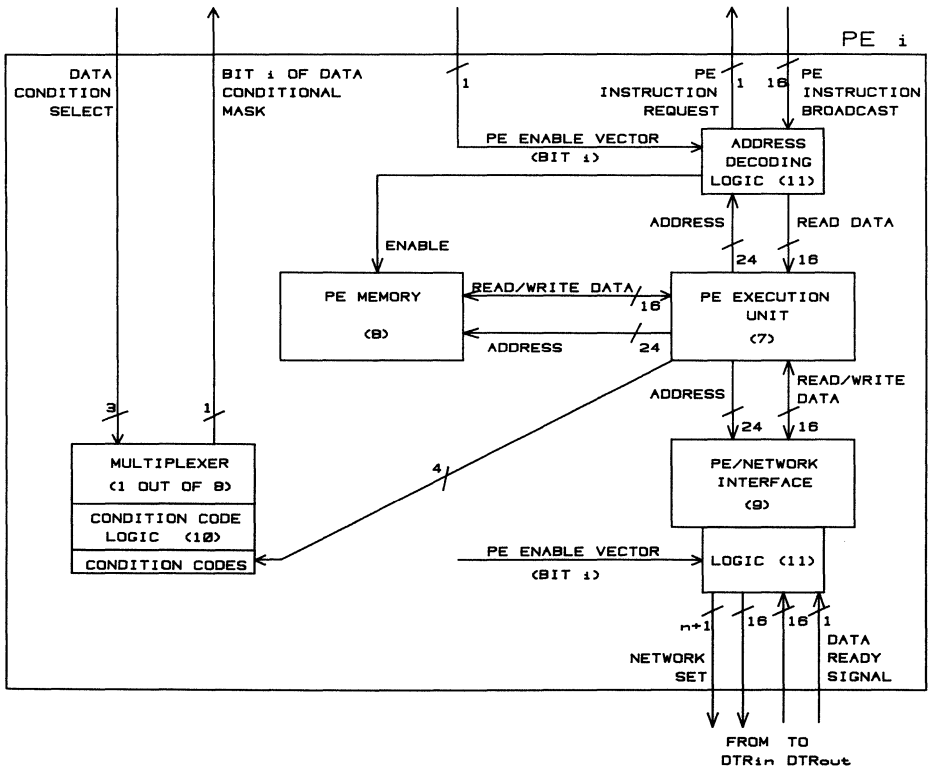


Figure 2. Model of a Processing Element (PE). The three ADDRESS buses shown coming from the PE execution unit are physically the same bus. Similarly, the three READ (/WRITE) buses are physically the same.

Data conditional masks are the result of performing a test on local (PE) data in an SIMD machine environment, where the results of different PEs' evaluations may differ. As shown in Figure 1, the CU receives an N-bit data conditional mask comprised of N one-bit "true/false" data conditional results, one result from each PE's condition code register. The "true/false" data conditional results are stored in the masking operations unit for use in activating or deactivating the PEs. For example, this type of data conditional masking was used in PEPE to implement the "where" conditional tests [21].

Certain CU execution unit instructions cause a branch based on data conditional mask information. For example, "if any" PE meets some criteria (a bit in the data conditional mask is "true"), the CU execution unit would execute a branch to a different part of the program. The masking operations unit uses the data conditional mask results from the PEs to evaluate the "if any," "if all," etc., conditions.

III. SIMD Simulation Overview

A. Introduction

Our earlier SIMD algorithm simulation studies have examined the possibilities for overlapped operation of the control unit, processors, and interconnection network [5, 11]. Overlapping can be improved as additional hardware (e.g., latches, buffers) is added to the CU/PE and PE/network interfaces. Six hardware configuration "cases" were identified and the relative run time performance of assembly language test algorithms was measured for each. A summary of the four cases from [5] and the two cases using tagged instruction words from [11] appear in Subsection B. In Subsection C, simulation results from the six cases will be summarized and compared. Based on the results, one of the cases will be chosen for the MC68000-based design.

B. Summary of Cases

In case 1, the CU and PEs are forced to operate in lock-step fashion. That is, while the CU is fetching or executing an instruction, the PEs are idled, and vice-versa. The STARAN system operates in a case 1 mode since there is a single instruction register in the control unit which contains the currently executing instruction [19].

Case 2 allows the CU to fetch instructions or execute CU instructions while the PEs are executing. However, the CU must wait until the PEs have completed their operation before broadcasting the next PE instruction.

A FIFO instruction queue shared by the PEs is added in case 3. This allows the CU to send PE instructions (opcodes and operands) to the queue without having to wait for the PEs to complete their current instruction. Associated with each opcode/operand pair in the queue, the N-bit enabled/disabled status associated with that instruction (the PE enable vector at fetch time) is stored. The PE enable vector must be stored since CU masking operations (changing the PE enabled/disabled status) might be performed before the queued PE instruction is actually executed.

The Illiac IV and MPP control units and PEPE arithmetic control unit use the case 3 overlap processing method [2, 19, 21]. All three machines employ data conditional masking, but the resulting masks are stored in the PEs themselves.

For case 4, a CU instruction buffer is added to the case 3 configuration. An implicit assumption for this case is that the fetch unit and CU execution unit are independent processors. (For cases 1-3, the fetch and execution units are not necessarily distinct.) The fetch unit classifies instructions as CU or PE instructions and sends them to the appropriate instruction buffer. The fetch unit must distinguish branch operations (including "if any/if all" branches) by stopping the fetching process when these instructions are encountered. Branch instructions affect the program counter (which the fetch unit maintains to know the "next" instruction), so the fetch unit must wait until the CU has emptied its instruction buffer and adjusted the program counter based on the result of the branch before continuing. Fetching is also discontinued during masking operations to allow the masking operations unit to associate the new PE enable signals with subsequently fetched PE instructions.

For the previous cases, the fetch unit decoded each instruction in order to determine where (the CU or the PEs) the instruction would be executed and the size (number of operands) of the instruction. This scheme required that the fetch unit have full knowledge of CU and PE instruction types and formats. This adds considerable complexity to the fetch unit, and would necessitate changes to it if either the CU or PE execution units were changed. An effective solution is to associate a tag with each CU memory word, specifying which component (CU execution unit, CU microprogrammed logic, or PEs) is to interpret the word. Cases 5 and 6 correspond to cases 3 and 4, but with fetching and buffering by words rather than by instructions. Each word (as opposed to each instruction, including both opcode and operands, as in cases 1-4) sent to the PE instruction buffer will have associated with it an N-bit enable vector.

The tag scheme just described has several advantages. First, the PEs will only be idled when the instruction queue becomes empty; an unlikely event since the instructions are delivered to the queue at the rate of the CU memory access time. The time needed to decode the tag is negligible in comparison to the time necessary to fully decode each instruction and determine how many operand words are associated with that instruction. The fetch unit no longer requires knowledge of the specifics of the PE instruction set since PE instruction words are treated as data. Furthermore, a 16-bit line connecting the CU to the processors is needed, as opposed to the 80 bit line if complete instructions, including operands, were sent to the PEs (assuming MC68000 instructions require 1 to 5 16-bit words). However, the instruction opcode must be decoded by the PE execution units to determine if "immediate" data operands or address fields are present. This step was previously done by the control unit; data operands were associated with the instruction opcode before being broadcast to the PEs.

C. Simulation Results

During simulations performed for several assembly language test algorithms, the relative run time performance of the 6 cases was measured. The results of cases 1-4 were presented in [5] but are summarized here for comparison with cases 5 and 6.

The assembly language instruction set that was used for the simulations is of our own design. It is similar to instruction sets supported by sophisticated current microprocessors, but augmented by instructions for the control unit operations, masking operations, and network data transfers.

The test algorithms are two versions of an image smoothing algorithm for a 16-PE system smoothing a 16x16 pixel image [13]. For these algorithms, each PE contains a subimage of 4x4 pixels. In the "original" version of the algorithm, a PE's subimage pixels and "border" pixels from adjacent PEs are copied to a 6x6 pixel "work area" array. Smoothing operations are performed on the pixels in the work area. For the "improved" version of the algorithm, the "border" pixels and a subset of the subimage pixels are copied to the work area. In this version, both the work area array and the subimage array are accessed during the smoothing operations. As will be shown, the original algorithm performs better for small images, while the improved algorithm performs better for large (more realistically-sized) images. Some parameters of the algorithms are shown in Table 1.

Table 1. Test algorithm characteristics. The "TOTAL CU" and "TOTAL PE" columns indicate the percentages of CU and PE instructions executed. "CU IF ANY" is a subclass of "CU BRANCH," which is a subclass of "CU TOTAL." Similarly, "PE NETWORK" instructions are included in the "TOTAL PE" classification.

ALGORITHM	INSTRUCTIONS EXECUTED	PE/CU INSTRUCTION RATIO	INSTRUCTIONS EXECUTED (PERCENT)				
			TOTAL CU	CU BRANCH	CU IF ANY	TOTAL PE	PE NET.
ORIGINAL	649	5.01	17	12	0	83	4
IMPROVED	680	4.23	19	15	0	81	4

Each test algorithm was assembled using a special assembler supporting the augmented instruction set and simulated using our Purdue SIMD Simulation and Timing (PSST) system. An instruction execution trace for each simulated algorithm was generated to be used later as input to the timing algorithms. A small number of PEs and small image sizes were used since the simulations of the SIMD system are performed comparatively slowly on a serial host computer. Details of the algorithms, instruction set, assembler, simulations, and timing routines are presented in [11].

In preparation for timing the simulations, each instruction in the instruction set was classified by its constituent operations and characteristics. These characteristics include the number of operand words to be fetched, the CU execution time (for CU instructions), the CU to PE transfer time (for PE instructions), the PE execution time and network execution time (for PE instructions), flags to indicate data conditional mask instructions, branch instructions, network instructions, and so on. A table of instructions and their characteristics was prepared.

The timing algorithms reference the instruction set characterization table and accept input of relevant timing information (e.g., opcode load time (1 cycle), 16-bit operand load time (1 cycle), buffer enqueue or dequeue time (1/2 cycle), mask decoding time (1 cycle)). The interconnection network set-up time and network propagation delay time were 1 cycle each. Finally, the instruction trace output from the test algorithms was used as input to evaluate the timing for cases 1-6. Note that the same instruction execution trace can be used repeatedly for many combinations of cases and timing assumptions. For these simulations, a circuit-switched network whose ports are directly connected to PE execution unit registers was assumed. No PE/network overlap was considered.

The run time results shown in Table 2 are normalized such that the case 1 timing = 1.00. As shown, the run time of case 3 is significantly less than those of case 2 and case 1. This was expected since the instruction "mix" for these algorithms is such that PE instructions greatly outnumber CU instructions and PE instructions occur in large groups, allowing the buffer to do its intended function. The case 4 run time falls somewhere between the case 2 and case 3 timing. The fact that case 4 performs worse than case 3 for these algorithms is not surprising since CU instructions rarely occur in groups (thus underutilizing the CU instruction buffer). Further, the percentage of branch instructions performed ranges from 70 to 80 percent of the CU instructions, thus preventing the filling of the CU instruction buffer in the case 4 configuration.

In cases 5 and 6 (fetching and buffering by words), the time needed to fetch and enqueue instructions, including their operands, is proportional to their length (cases 1-4 had a constant time). However, the simplified tag decoding for these cases might offset the overhead of the extra enqueue/dequeue operations. Comparisons made between cases 1-4 and 5-6 may be influenced

Table 2. Normalized run times for cases 1-6.

ALGORITHM	CASE 1	CASE 2	CASE 3		CASE 4		CASE 5		CASE 6	
			a	b	a	b	a	b	a	b
ORIGINAL	1.00	.73	.67	.66	.71	.70	.76	.73	.84	.81
IMPROVED	1.00	.74	.68	.68	.72	.72	.77	.74	.86	.83

(a) "Enqueue" and "dequeue" operations may not overlap each other.
(b) "Enqueue" and "dequeue" operations may overlap each other.

strongly by the simpler (and potentially faster) case 5-6 hardware. For example, enqueue and dequeue times may be shorter for cases 5 and 6 since all queuing functions involve a shorter, fixed-size word. The very wide bus assumed in cases 1-4 may in reality be a smaller, time-multiplexed bus, thus increasing the CU/PE instruction transfer time for those cases. If the fetch, decode, enqueue, dequeue, and execution times are assumed to be the same as for cases 1-4, cases 5 and 6 perform somewhat worse than the case 2 configuration because of the aforementioned factors. Case 3 is faster than case 5 because enqueuing and dequeuing operations are not done word-by-word. The speed advantage of case 3 would be negated if it used a 16-bit time-multiplexed bus and a slightly slower fetch/decode unit. The percentage of instructions with operands and the average operand length (both algorithm-dependent parameters) also influence the relative performance of the cases greatly.

The instruction queue sizes chosen for the case 3-6 configurations also have an effect on the algorithms' run time. The minimum size needed was seven words for case 3, six for case 4, and three for cases 5 and 6. A detailed analysis of the minimum PE instruction buffer sizes required to get the same overall execution time the infinite buffer (assumed in Table 2) would provide is given in [11].

Based on the simulation results obtained for the SIMD mode, the case 5 configuration has been chosen. Case 3 was not chosen because of the more complex fetch unit design and the very wide CU/PE bus width requirement. Assuming the use of standard microprocessors, the case 3 configuration unnecessarily duplicates the instruction decoding function of the PEs. A narrower, time-multiplexed CU/PE bus could be implemented with case 3, but this approach would likely negate the speed advantage gained by buffering instructions as a unit. Furthermore, standard microprocessors accept instructions word-by-word. Case 5 simplifies the design of the fetch unit considerably since tags associated with each memory word indicate that word's destination. The fetch unit requires no knowledge of either the CU or PE execution unit's processor instruction set. The tagged memory scheme also allows the instruction complement of the microprogrammed hardware to be developed independently. The PE instruction queue and bus width of 16 bits is quite manageable. The case 5 queue may be longer since it is word-by-word, but has a much narrower width that is always fully utilized. Simulation results of the MC68000-based system are presented in a later section.

IV. The MC68000-Based PE

Referring to the model of a PE (Figure 2), consider incorporating the Motorola MC68000 processor as the PE execution unit. The processor itself, 256K-bytes of PE memory, and some simple latches (PE/network interface, condition code register) and logic can easily fit on a single physical board. The organization of the model was chosen carefully so that the number of wires running between the CU and PEs is minimized. The consolidation of specialized hardware in the CU makes each PE board simpler and cheaper to construct.

The MC68000 is a state-of-the-art 16-bit microprocessor [20, 7]. Internally, it can operate on bit, byte, word (16-bit), and long (32-bit) data formats. Its fast cycle time and large address space (currently 24-bit addresses) make it ideal for image processing applications where speed and large data set handling capabilities are a must. Its very regular instruction set, many addressing modes, and suitability to high-level language operations make it easy to program. While some of the MC68000's functions go unused when it operates in SIMD mode (e.g., branch and control operations), these functions are essential for MIMD "stand-alone" processing. While the MC68000 is not quite as "powerful" as the Illiac IV [3] or PEPE [21] PE, it is considerably more complex than the STARAN [1] or MPP [2] processors.

Each PE will be able to address any of three logical address spaces. Physical PE memory addresses (both ROM and RAM addresses) will represent one space. Addresses of I/O ports will be contained in the second space. The PE instruction queue (for the case 5 configuration) will have addresses in the third space. Initially, all PEs will be enabled, and have their internal program counter set to the address of the beginning of the PE instruction queue space. When the PEs try to fetch the first SIMD instruction, the address sent out by each of the PE execution units will be decoded by the "address decoding logic" as a reference to the PE instruction queue space. This logic will send an "instruction request signal" to the FIFO instruction queue. When all PEs request an instruction, the buffer acknowledges the requests and puts an instruction word on all the PE data buses. Each PE decodes the instruction and performs the operation or requests additional operand words. If the logic determines that a PE memory or I/O device address is being referenced, the operation is performed normally.

In SIMD mode, the PE program counter serves only to identify a request for an instruction word. The actual value of the PE program counter is irrelevant, as long as it references an instruction in the PE instruction queue space. However, the program counter is incremented automatically upon receiving an instruction from the PE instruction queue. Eventually, the program counter will near the end of the instruction queue space and will need to be reset. The instruction queue address space is made large so that the overhead of resetting the program counter is minimal.

When the PE enable vector specifies that a PE is to be disabled, the address decoding logic in that PE continues to send an instruction request signal to the PE instruction queue. However, the acknowledgement and data word from the queue is intercepted by the logic so that the PE execution unit never "sees" the instruction. When the PE execution unit is re-enabled, processing can continue.

In order to avoid internal modifications to the PE execution unit, PEs will communicate via the interconnection network using a sequence of I/O port read and write operations. A PE specifies where its data is to be routed by computing the address of the destination processor (PEs are addressed 0 to N-1). The address is written to an

external (n+1)-bit "network set" latch (the "extra" bit will be described later). This action instructs the network to set switches to make a connection with the destination address [6]. Data transmissions will occur through two 16-bit external data latches called Data Transfer Registers (DTRs) [13]. One latch is connected to the network input (DTRin), and the other to the network output (DTRout). The data to be transmitted is written to the DTRin latch. Finally, a control word is written to an external 1-bit "network transfer" register, signaling to the network that the transfer should be made. Subsequent transfers route items to the same destination until the "network set" latch is modified. In SIMD mode, all PEs do these operations at the same time. In MIMD mode, PEs use the network asynchronously.

At the destination PE, the network sets a flag indicating that the DTRout contains newly-transferred data and may be read. When the PE attempts to read DTRout, specialized logic examines this flag. If the PE attempts to read DTRout prematurely (the flag is not yet set), the PE is made to wait until the network has passed the data. For this reason, other processing is often done during network transfers to maximize overlapped operation of the PEs and network. In MIMD mode, a PE might send data faster than the destination PE requires it as input. In this situation, the network-generated signal flag might be used to interrupt the receiving PE and instruct it to buffer the incoming data.

When a PE is disabled, logic insures that the "network set" data does not create "conflicts" in the network switch settings. The "extra" bit in the "network set" latch is used to indicate that this network input should be ignored. A disabled PE may receive data normally since the DTRout is unaffected by the enabled/disabled status of the PE execution unit. When re-enabled, the PE can read DTRout.

When a data conditional mask is needed, PE instructions to evaluate the PE data condition are executed. Then the PEs write their status register (which contains the processor condition codes) to the condition codes register. Logic associated with the condition codes register can generate eight different conditional tests (e.g., equal, not equal, positive, overflow). Data condition select lines from the CU specify which of the conditional tests is to be returned to the masking operations unit as that PE's component of the data conditional mask.

From time to time, the CU fetch unit will enqueue a JUMP instruction to the beginning of the PE instruction queue space. This is to prevent the PE program counters from entering a different address space. The mechanism that the fetch unit uses to perform this function will be described later. When the machine is to change from SIMD to MIMD mode, the fetch unit broadcasts a JUMP instruction to some address within the PE memory space. Typically, this would be the beginning of a program stored in ROM that would initialize the PE operating system for MIMD processing. While in MIMD mode, PEs do not access the PE instruction queue space since MIMD instructions and data are contained entirely within the PE memory. When the PE is ready to revert to SIMD mode, it jumps to the beginning of the PE instruction queue space. When all the PEs have done this, SIMD processing continues.

V. CU Architecture Details

There exist no off-the-shelf processors that can perform all of the functions of the control unit at a speed sufficient to keep the PE execution units busy. Therefore, fast microprogrammable bit-slice components will be used for all of the CU specialized functions. These functions include the operations of the fetch unit, masking operations unit, and CU/PE interface. In order to simplify the programming of the system and to make data formats uniform throughout, the CU execution unit will also be an MC68000 processor. For comparison, the execution component of the Illiac control unit is a powerful 64-bit integer/floating point processor [3]. The MPP "main control" and the PEPE arithmetic control unit are also quite sophisticated [21, 2]. By contrast, the STARAN execution unit and MPP "PE control" unit consist of only a few dedicated registers for loop counting and handling of "associative array field pointers" [19, 2].

The speed at which the bit-slice fetch unit can fill the PE instruction queue to capacity, and the large ratio of PE to CU instructions in algorithms programmed so far indicates that the MC68000 will be an acceptable CU execution unit. When the subset of MC68000 instructions actually used in normal CU execution unit operations is defined through actual use and further simulation, and if there is a need for more speed, the MC68000 could be replaced with a bit-slice machine.

CU memory will be comprised of 20-bit words. The most significant four bits will be interpreted by the microprogrammed logic portion of the fetch unit as a destination for the remaining 16.

The CU fetch unit will contain two registers: the Fetch Unit Program Counter (FUPC) and the PE Space Counter (PESC). The FUPC gives the address of the next instruction to be fetched from CU memory. The CU execution unit program counter serves only to identify a request for an instruction word. The actual value of the CU execution unit program counter is irrelevant, except when branch instructions are executed. The FUPC and the CU execution unit program counters must be equal before a branch instruction is executed since computations using the program counter will be done (e.g., relative branches).

The PESC begins at zero and is incremented each time a word is enqueued in the PE instruction queue. When the PESC reaches a threshold value close to the size of the PE instruction queue space, the fetch unit enqueues a JUMP instruction before the next PE instruction. (The first word of a PE instruction has a special tag.) The JUMP instruction causes the PE program counter to be reset to the beginning of the PE instruction queue space (see Section IV). When the JUMP instruction is enqueued, all PEs are temporarily enabled. The PESC register is also reset to zero.

The 4-bit memory word tags will specify what sequence of actions the microprogrammed logic is to take. Examples of these actions are enqueuing a PE instruction opcode or operand, sending a CU

instruction to the CU execution unit, mask decoding, and-ing and or-ing of masks, PE data condition selection, initialization of the CU execution unit, masking operations unit, PEs, or I/O devices, etc.

The CU fetch unit never operates at the same time the CU execution unit is performing branch instructions or while the masking operations unit is operating. The CU execution unit may modify the program counter which the fetch unit maintains to know the "next" instruction. The masking operations unit may modify the PE enable vector which must be associated with each enqueued PE instruction.

The masking operations unit maintains a stack of N-bit masks generated by nested "where" conditionals and PE address masks [11]. The PE enable vector that is currently on the top of the stack is enqueued whenever a PE instruction word is enqueued. The details of the stack operations, stack hardware, and the interplay between SIMD programs and masks are discussed in [11].

The PE instruction queue (CU/PE interface) is a high-speed I/O buffer N+16 bits wide and 32 words long. This length allows about ten average PE instructions to be queued. A head and tail pointer indicate the position of the next word to be dequeued or enqueued, respectively. The buffer dequeues a word if nonempty and when all PEs make the request (inactive PEs are always "requesting"). The fetch unit may enqueue a word provided the queue is nonfull.

In order for the instruction queue to be useful, the total time to fetch an average instruction, decode its tags, and enqueue its constituent words should not exceed the time needed by the PE to execute that instruction. Given that 2900-series microprogrammable bit-slice components have a cycle time of 200 nanoseconds vs. the MC68000 basic memory cycle time of 500 nanoseconds, there should be no problem in filling the PE instruction queue to keep the PEs "satisfied." If the queue is sufficiently large, the execution of several consecutive CU execution unit, masking, or control instructions should not empty the queue and "starve" the PEs.

For a prototype system of size N = 16 or 32 PEs, the MC68000 execution unit could be used to simulate some of the CU operations in software and monitor the PEs. For example, the masking operations unit and CU/PE interface could be implemented in software (but at a cost in system speed). The large address space of the MC68000 could be used to access any part of up to thirty-two 256K-byte PE memories if the hardware is so arranged. This scheme would be most useful in a prototype: the CU execution unit could load and unload PE memories, monitor the behavior of individual PEs, and so on. (A real system would not use this scheme because of speed and memory contention problems.)

VI. MC68000 Simulation Results

The simulation of the MC68000-based system was carried out using the same techniques as described earlier. However, these simulations required the writing of new SIMD algorithms in the MC68000 instruction set, a specialized version of an MC68000 assembler, and new PSST simulation programs. The PSST timing algorithms were largely unchanged, but a new table of instruction timing characteristics had to be prepared.

The PSST simulator consists of two main coroutines: the simulation of an MC68000 processor and the simulation of the CU microprogrammed logic. The actions of the fetch unit and masking operations unit are included in the CU microprogrammed logic simulation. When the CU execution unit is to be activated, a copy of the "CU data area" is passed to the MC68000 simulator and processing is initiated. When a PE is to be activated, a copy of the appropriate "PE data area" is passed to the MC68000 simulator. The action of the CU/PE interface (case 5: overlapping of the instructions) is simulated by the timing routines.

The PSST simulator for the MC68000 system is largely complete although it lacks BCD arithmetic operations, trap and exception processing, interrupts, and MIMD operation (the asynchronous interaction of the PEs). It also cannot detect interconnection network "conflicts." Major effort will be required to implement interrupts and MIMD operation in both the simulation and timing routines.

Two versions of the SIMD image smoothing algorithm for a 16-PE system were simulated. The algorithms are identical to those described in Section III. Simulations of both algorithms were performed for a variety of image sizes ranging from 16x16 to 128x128 pixels. The complete image can be superimposed onto an array of 4x4 (=16) PEs such that each PE processes a subimage of 4x4 to 32x32 pixels.

Table 3 compares the simulation and timing

Table 3. Comparison of smoothing algorithm simulation and timing characteristics. The "original" algorithm run time results are normalized to 1.00. The internal cycle time is 250ns. All of the simulations are performed with N=16 PEs.

TOTAL IMAGE SIZE (PIXELS)	SUBIMAGE SIZE (PIXELS PER PE)	ORIGINAL ALGORITHM			IMPROVED ALGORITHM		
		INSTRUCTIONS EXECUTED	TIME (CYCLES)	TIME (NORMALIZED)	INSTRUCTIONS EXECUTED	TIME (CYCLES)	TIME (NORMALIZED)
16x16	4x4	729	4002	1.00	796	4370	1.09
32x32	8x8	2011	12588	1.00	2089	13079	1.04
48x48	12x12	4101	26628	1.00	4060	26362	0.99
64x64	16x16	6005	42196	1.00	5615	39875	0.94
128x128	32x32	18443	146476	1.00	15493	123040	0.84

results for the two smoothing algorithms. The run time has been normalized such that the original algorithm run time=1.00. These results indicate that the original algorithm is more efficient for small subimages (fewer than 12x12 pixels per PE) than the "improved" algorithm. The improved algorithm would be used for real-world-size problems.

The actual algorithm execution time can be calculated for a given algorithm/image size pair by multiplying the number of cycles by the cycle time. Assuming a standard 8MHz MC68000 processor, the internal cycle time is two clock cycle times, or 250ns. Thus, a 128x128 (=16K) pixel image can be smoothed by the 16-PE system in about 31ms. Note that this is algorithm execution time. The simulations do not include data load/unload time between primary and secondary memory (which will be highly implementation dependent, e.g., see [13]).

The 128x128 pixel simulation required about 16 minutes of VAX cpu time. This corresponds to an average execution rate of over 19 SIMD instruction per second of cpu time. Recall that the simulator executes a single PE instruction 16 times, once for each PE. Somewhat less than half of the cpu time may be saved if the "PE memory dump" following the simulation is inhibited. The writing of 128^2 numbers to disk files (for verification of the smoothed output) takes a considerable amount of time.

A "serialized" (single PE) algorithm was constructed from the original parallel algorithm to determine the "speedup." The serial algorithm operates on the entire image (rather than a subimage) and thus does not need to perform masking or inter-PE transfer operations. When the number of masking and transfer operations per pixel processed (parallel overhead) is high, the parallel algorithm will not be very efficient. If no overhead is involved, the parallel algorithm should execute 16 times faster on a 16-PE machine than on a 1-PE machine. As shown in Table 4, the parallel algorithm performs relatively poorly for small subimage sizes, but approaches a perfect speedup for "real-size" tasks.

Table 4. Determination of the speedup factor of the original parallel algorithm. All of the simulations are performed with N=16 PEs.

TOTAL IMAGE SIZE (PIXELS)	SUBIMAGE SIZE (PIXELS PER PE)	SERIAL TIME / PARALLEL TIME
16x16	4x4	9.52
32x32	8x8	13.16
48x48	12x12	13.64
64x64	16x16	14.32
128x128	32x32	15.56

It was observed that the MC68000 divide instruction, which is executed once per pixel processed to scale the result, accounts for roughly 35% of the total run time. The divide instruction requires about 75 machine cycles as compared to a typical add instruction requiring about 4 cycles. If better run times were necessary, the algorithm could be restructured to smooth a window of eight nearest-neighbor pixels (as opposed to nine) and scale the data by shifting the result right by three bits. A typical 3-bit shift requires 7 cycles, or about 10% of the divide cycle time. However, a load and add cycle (about 7 cycles) is saved since only eight pixels are used in the window. Thus a 35% improvement can be gained by replacing the divide instruction.

The 75 cycles for a divide instruction is the maximum instruction time; the actual time required is data-dependent and is not considered by the PSST timing routines. If some PEs finish the instruction before others, they will be made to wait until all the PEs have finished. Recall that a PE instruction is dequeued from the FIFO buffer only when all PEs make the request for the next instruction. However, if all of the PEs finish the division before the 75 cycle maximum, the hardware will be able to exploit this and release the next instruction to the PEs.

The simulation results presented may be extrapolated to determine timings and speedups for other machine and/or image sizes. The run time of an algorithm depends on the relative sizes of the machine and the image, or equivalently, the subimage size in pixels per PE. For the smoothing examples, a minimum machine size of 4 PEs is necessary and sufficient so that all relevant inter-PE transfer and masking instructions are included. For example, a 4-PE SIMD machine can smooth an 8x8 pixel image in the same amount of time as a 16-PE machine can smooth a 16x16 pixel image. In each case, a PE operates on a subimage of 16 pixels. Similarly, since 16 PEs can smooth a 128x128 (=16K) pixel image in 31ms, a 64-PE system of the same design and using the same algorithm could smooth a 256x256 (=64K) pixel image in 31ms. (For larger machines, the number of stages in the Generalized Cube network will increase; however, assuming that the propagation delay of the network is overlapped with PE operations, the impact of the added stages is negligible.) In general, increasing the number of PEs by a factor of four allows four times as many pixels to be processed in the same amount of time. However, this does not mean that processing four times as many pixels will take four times as long for a fixed machine size. In the latter case, the fixed and variable costs of performing the particular algorithm must be taken into account.

VII. Conclusions

Based on the results of past simulation studies, the design of an extensible SIMD/MIMD machine based on state-of-the-art microprocessors and off-the-shelf components was developed. The interface logic necessary for SIMD/MIMD processing was found to be minimal. Thus the high cost of designing and fabricating a custom VLSI PE has been saved. The architecture could be used as a single SIMD/MIMD machine, or as a building block for a larger multiple-SIMD or partitionable SIMD/MIMD system using the techniques described in [13]. Also, the design presented is easily modified even after it is constructed since the CU does not decode any PE instructions. This is

especially important since the MC68000 processor is not yet in the final stages of its evolution. The use of an MC68000-based control unit in a prototype has also been shown to be highly desirable. In a final design however, many of the CU functions will have to be implemented using bit-slice technologies.

Given these considerations, it appears that a powerful SIMD/MIMD system having at least 128 processors could be built without encountering severe physical hardware restrictions (e.g., space, power, and cooling requirements, bus length restrictions), and at a reasonable cost using current technology. Further, we have working SIMD machine simulators and trace-driven timing analysis algorithms that can be used to evaluate additional SIMD programs for image processing and pattern recognition in order to study various system architecture features.

References

[1] K. E. Batcher, "STARAN parallel processor system hardware," AFIPS 1974 Nat'L. Comp. Conf., May 1974, pp. 405-410.

[2] K. E. Batcher, "Design of a massively parallel processor," IEEE Trans. Comp., Vol. C-29, Sept. 1980, pp. 836-844.

[3] W. Bouknight et al., "The Illiac IV system," Proc. IEEE, Vol. 60, Apr. 1972, pp. 369-388.

[4] M. Flynn, "Very high speed computing systems," Proc. IEEE, Vol. 54, Dec 1966, pp. 1901-1909.

[5] J. T. Kuehn and H. J. Siegel, "Simulation studies of PASM in SIMD mode," IEEE Computer Architecture for Pattern Analysis and Image Database Management Workshop, Nov. 1981, pp. 43-50.

[6] D. H. Lawrie, "Access and alignment of data in an array processor," IEEE Trans. Comp., Vol. C-24, Dec. 1975, pp. 1145-1155.

[7] Motorola Semiconductor, MC68000 16-bit Microprocessor User's Manual, Motorola IC Division, Austin, TX, 78721.

[8] G. J. Nutt, "Microprocessor implementation of a parallel processor," 4th Symp. Comp. Arch., Mar. 1977, pp. 147-152.

[9] M. Pease, "The indirect binary n-cube microprocessor array," IEEE Trans. Comp., Vol. C-26, May 1977, pp. 458-473.

[10] H. J. Siegel, "Analysis techniques for SIMD machine interconnection networks and the effect of processor address masks," IEEE Trans. Comp., Vol. C-26, Feb. 1977, pp. 153-161.

[11] H. J. Siegel and J. T. Kuehn, Parallel Image Processing/Feature Extraction Algorithms and Architecture Emulation: Interim Report for Fiscal 1981, Volume II: Architecture Emulation, School of Electrical Engineering, Purdue University, Technical Report, Oct. 1981.

[12] H. J. Siegel and R. J. McMillen, "The multistage cube: a versatile interconnection network," Computer, Vol. 14, Dec. 1981, pp. 65-76.

[13] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller, Jr., H. E. Smalley, Jr., and S. D. Smith, "PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition," IEEE Trans. Comp., Vol. C-30, Dec. 1981, pp. 934-947.

[14] L. J. Siegel, "Image processing on a partitionable SIMD machine," in Languages and Architectures for Image Processing, M. Duff and S. Levialdi, ed., Academic Press, London, 1981.

[15] L. J. Siegel, E. J. Delp, T. N. Mudge, and H. J. Siegel, "Block truncation coding on PASM," 19th Ann. Allerton Conf. on Communication, Control, and Computing, Oct. 1981, pp. 891-900.

[16] K. G. Stevens, Jr., "CFD - A FORTRAN-like language for the Illiac IV," Conf. Programming Languages and Compilers for Parallel and Vector Machines, ACM, Mar. 1975, pp. 72-76.

[17] H. Sullivan, T. R. Bashkow, and K. Klappholz, "A large-scale homogeneous, fully distributed parallel machine," 4th Symp. Comp. Arch., Mar. 1977, pp. 105-124.

[18] R. Swan, S. Fuller, and D. Siewiorek, "Cm*: a modular, multimicroprocessor," AFIPS 1977 Nat'L. Comp. Conf., June 1977, pp. 637-644.

[19] K. J. Thurber, Large Scale Computer Architecture: Parallel and Associative Processors, Hayden Book Co., Rochelle Park, NJ, 1976.

[20] H-m. D. Toong and A. Gupta, "An architectural comparison of contemporary 16-bit microprocessors," IEEE Micro, Vol. 1, May 1981, pp. 26-37.

[21] C. R. Vick and John A. Cornell, "PEPE architecture-present and future," AFIPS 1978 Nat'L. Comp. Conf., June 1978, pp. 981-992.

[22] W. Wulf and C. Bell, "C.mmp--A multiminiprocessor," 1972 Fall Joint Computer Conf., Dec. 1972, pp. 765-777.

ANALYSIS OF THE PASM CONTROL SYSTEM MEMORY HIERARCHY

David Lee Tuomenoksa
Howard Jay Siegel

Purdue University
School of Electrical Engineering
West Lafayette, IN 47907

Abstract - Many proposed large-scale parallel processing systems (e.g., PASM) can operate in multiple-SIMD mode. The multiple control units in such a system share a common secondary storage for programs. The control units use paging to transfer programs to their primary memories. One design problem is determining the optimal service rate for the secondary storage, where the "optimal" is characterized by maximum processor utilization. The problem is approached by developing a queueing network model for the PASM control system memory hierarchy. Based on assumed values for parameters which characterize the expected task environment, an optimal service rate is derived from the model. The values of the parameters in the model can be varied to determine the impact these changes would have on system performance. Simulation results verifying various aspects of the model are presented. The results are shown to apply to the general model for a multiple-SIMD machine.

I. Introduction

A multiple-SIMD system (e.g., [9]) is a parallel processing system which can be dynamically reconfigured to form one or more independent SIMD (single instruction stream - multiple data stream) [5] machines of varying sizes. Handling the memory management problem for the multiple control units is an issue which must be considered in the design of multiple-SIMD systems. One possible solution to the problem is the use of virtual memory [1]. If virtual memory is used in a multiple-SIMD system with common secondary storage for the multiple control units it is necessary to determine the optimal page request service rate for the secondary storage. The optimal is characterized by maximum utilization of the processors.

PASM is a multimicrocomputer system being designed at Purdue University for a variety of image processing and pattern recognition problems [10]. It is the use of PASM in the multiple-SIMD mode of operation which motivates this study. In this paper a queueing network model is developed for the memory hierarchy of the multiple control units in PASM and is analyzed to determine the optimal page request service rate for the secondary storage. The optimal service rate for the secondary storage can be determined from the average system page request rate using heuristics for serial multiprogrammed systems as a guideline. The average system page request rate can be determined by making assumptions about the task environment (e.g., number of processing elements which tasks require and the estimated execution time of the tasks). The system idle time which results from the multiple control units waiting for page requests to be serviced is determined for the case where the system page request rate deviates from the average rate which was used to determine the optimal secondary storage service rate. The values of the parameters in the model can be varied to determine the impact these changes would have on system performance. Simulation results verifying various aspects of the model are presented. The model and analysis is related to a general model for a multiple-SIMD machine.

Section II is an overview of the PASM multimicrocomputer system. Terminology is defined in Section III. In Section IV a queueing network model for the PASM control system memory hierarchy is developed. The average system page request rate for PASM is determined in Section V. In Section VI the optimal service rate for the secondary storage is determined. Operational analysis [3] is used to determine the average idle time for the multiple control units in Section VII. Simulation results are presented in Section VIII. In Section IX the analysis is related to the general model for a multiple-SIMD machine.

II. PASM Overview

PASM, a *pa*rtitionable *S*IMD/*M*IMD machine, is a large-scale dynamically reconfigurable multiprocessor system [10]. It is a special purpose system being designed to exploit the parallelism of image processing and pattern recognition tasks. PASM can be partitioned to operate as many independent SIMD and/or MIMD (multiple instruction stream - multiple data stream) machines of varying sizes. *PASMOS* is the operating system for PASM.

A block diagram of the basic components of PASM is given in Figure II.1. The *Parallel Computation Unit* contains $N = 2^n$ processors, N memory modules, and an interconnection network (see Figure II.2). The *Parallel Computation Unit processors* are microprocessors that perform the actual SIMD and MIMD computations. The *Parallel Computation Unit memory modules* are

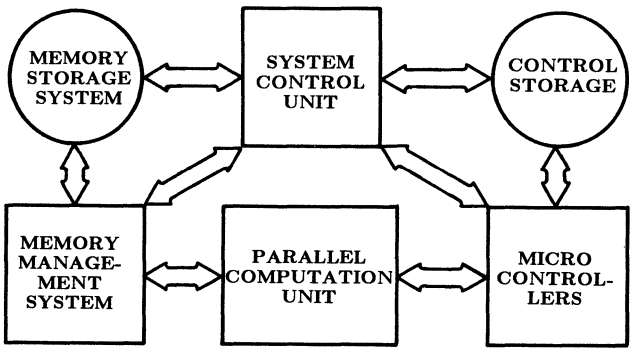


Figure II.1: Block diagram overview of PASM.

This research was supported by the Air Force Office of Scientific Research, Air Force Systems Command, USAF, under grant number AFOSR-78-3581 and by a Purdue University Graduate Fellowship. The United States Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation hereon.

0190-3918/82/0000/0363$00.75 © 1982 IEEE

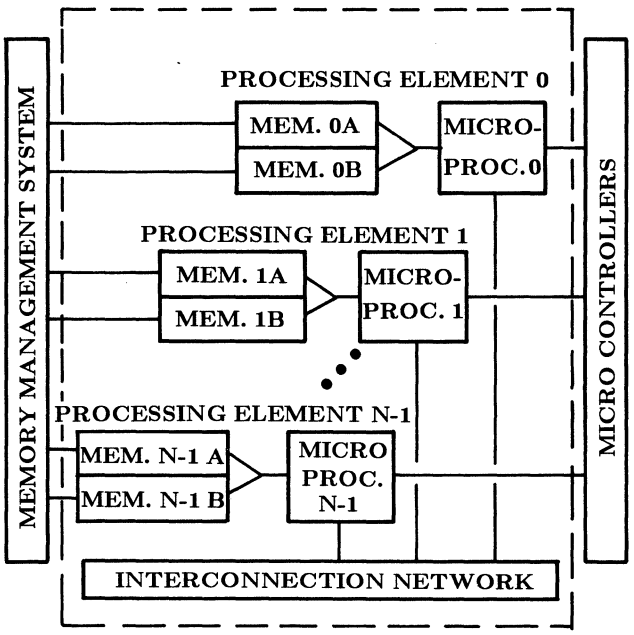


Figure II.2: PASM Parallel Computation Unit.

used by the Parallel Computation Unit processors for data storage in SIMD mode and both data and instruction storage in MIMD mode. The *interconnection network* provides a means of communication among the Parallel Computation Unit processors and memory modules. The *System Control Unit* is a conventional machine, such as a PDP-11, and is responsible for the overall coordination of the activities of the other components of PASM.

The *Memory Storage System* provides secondary storage space for the Parallel Computation Unit data files in SIMD mode, and for both the Parallel Computation Unit data and program files in MIMD mode. The *Memory Management System* controls the transferring of files between the Memory Storage System and the Parallel Computation Unit memory modules. It employs a set of cooperating dedicated microprocessors. Multiple storage devices are used in the Memory Storage System to allow parallel data transfer.

The *Micro Controllers (MCs)* are a set of microprocessors which act as the control units for the Parallel Computation Unit processors in SIMD mode and orchestrate the activities of the Parallel Computation Unit processors in MIMD mode. There are $Q = 2^q$ MCs. Each MC controls N/Q Parallel Computation Unit processors [7]. A virtual SIMD machine (partition) of size RN/Q, where $R = 2^r$ and $1 \leq r \leq q$, is obtained by loading R MC memory modules with the same instructions simultaneously. Similarly, a virtual MIMD machine of size RN/Q is obtained by combining the efforts of the Parallel Computation Unit processors and R MCs. Q is therefore the maximum number of partitions allowable, and N/Q is the size of the smallest partition. Possible values of N and Q are 1024 and 16, respectively.

Each MC processor is attached to a memory module which consists of a pair of memory units. The second memory unit may be used to load the initial pages of the next task while the current task is executing instructions from the first memory unit. In this analysis the steady-state condition is considered, i.e., the effect of preloading is ignored. Since a task which is executing uses only one memory unit, the paging analysis dose not consider the double-buffering. In SIMD mode, each MC fetches instructions from its memory module, executing the control flow instructions (e.g., branches) and broadcasting the data processing instructions to its Parallel Computation Unit processors. In MIMD mode the MCs may be used to help coordinate the activities of their Parallel Computation Unit processors.

SIMD programs are stored in the *Control Storage* which is the secondary storage for the MCs (see Figure II.1). The loading of SIMD programs from the Control Storage into the MC memory units is controlled by the System Control Unit and *Control Storage Controller.* The Control Storage Controller is a dedicated microprocessor which manages the Control Storage file system. When large SIMD tasks are run, i.e., SIMD tasks which require more than N/Q processors, more than one MC executes the same set of instructions. Therefore each of the MC memory units must be loaded with the same set of instructions. The fastest way to load several MC memory units with the same set of instructions is to load all of the memory units at the same time. This can be accomplished by connecting the Control Storage to all the MC memory units via the MC Memory System Switch. A block diagram of the control system memory hierarchy is given in Figure II.3. The MC Memory System Switch is controlled by the Control Storage Controller. All interaction between the Control Storage and the System Control Unit is done through the Control Storage Controller. (In [10] an enhanced scheme for connecting the MC processors to the MC

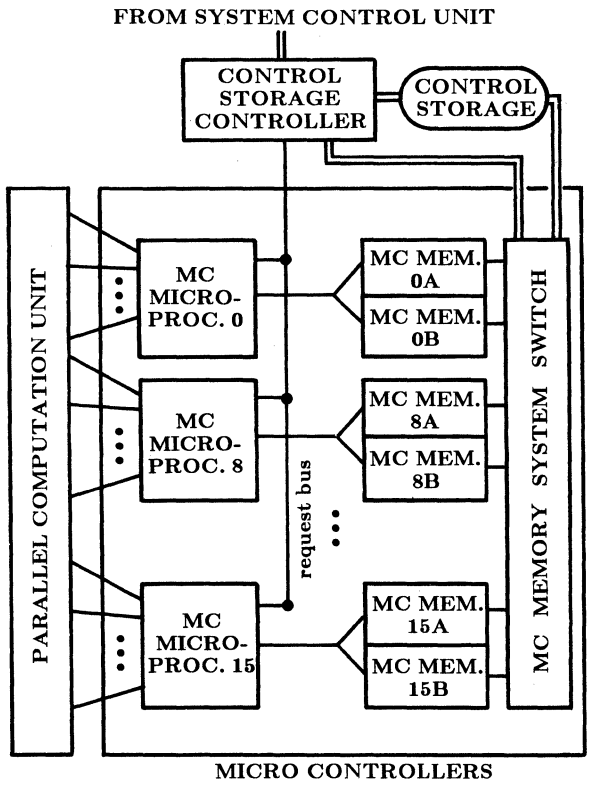


Figure II.3: Overview of PASM control system memory hierarchy for Q=16.

memory modules is also considered. The analysis in this paper also applies directly to that scheme.)

For some applications of PASM it is possible that the SIMD programs may be too large to fit into the primary memory (memory unit) of a given MC. Virtual memory may be used to give the programmer the illusion that the primary memory is much larger than in reality. There are two methods for implementing virtual memory: paging and segmentation [1]. In this paper, paging is considered. To implement paging as a part of the PASMOS operating system, the system must provide a translation mechanism to map the virtual address, which is used by the programmer, to a physical address, which is used by the system. In PASM, the translation is done by the MCs. When the page is not in the MCs primary memory (memory unit), it has a page fault. When an MC has a page fault, the MC sends a request on the request bus (see Figure II.3) to the Control Storage Controller which then services the request by locating the page in the Control Storage and sending it to the appropriate MC memory units through the MC Memory System Switch.

Consider the case where an SIMD task requires more than one MC. When a page fault occurs for the task, all of the MCs which are executing the task have a page fault. Since the faulted page is the same for all of the MCs which are executing the task, the page may be broadcast to all of the MC memory units simultaneously through the MC Memory System Switch. Hence, only one of the MCs must report the page fault to the Control Storage Controller, i.e., only one page request is generated. The MC which reports the fault can always be the same (e.g., logical 0 in the virtual SIMD machine) or may vary from one page fault to the next.

When PASM is operating as a number of independent virtual SIMD machines of varying sizes, the MCs are in effect a virtual MIMD machine. The secondary storage to this MIMD machine is the Control Storage. The model for the control system memory hierarchy is developed in Section IV.

III. Terminology

In this section the terminology which is used in the analysis is defined. *Virtual time* is defined to be the time that a processor is executing a task not including the time that it is idle waiting for page faults to be serviced or the time which a task is not assigned to it. *Real time* includes all time. The *real page fault rate,* referred to as the "page fault rate," is defined to be the rate at which page faults occur over real time, i.e., the number of page faults for the processor divided by the real time. The *virtual page fault rate* is defined to be the rate at which page faults occur over virtual time, i.e., the number of page faults for the processor divided by the virtual time. The *real page request rate,* referred to as the "page request rate," and *virtual page request rate* are the rates which pages are requested from the secondary storage over real time and virtual time, respectively.

For PASM, the virtual page request rate for MC_i is ν_i. The MC utilization is the fraction of time an MC is executing. The utilization of MC_i is U_i. The (real) page request rate for MC_i is $\lambda_i = U_i\nu_i$. The (real) system page request rate is the combined page request rates of all the MCs and is denoted by λ_{sys}.

IV. Model

In this section a queueing network model is developed for the PASM control system memory hierarchy. The interaction between the MCs and the Control Storage can be modeled by the two-station cyclic network in Figure IV.1 [3]. The MC subsystem contains the Q MC processor-memory unit pairs. Since only one of the MCs in a group of MCs executing a task makes page requests, there are only T MCs making requests where T is the number of tasks executing. Hence, the network is closed with T customers. The average time between page requests for each of the MCs making page requests is $1/\Lambda$, where Λ is assumed to be the virtual page fault rate for all tasks. The page request rate of the MC subsystem is the system page request rate λ_{sys}.

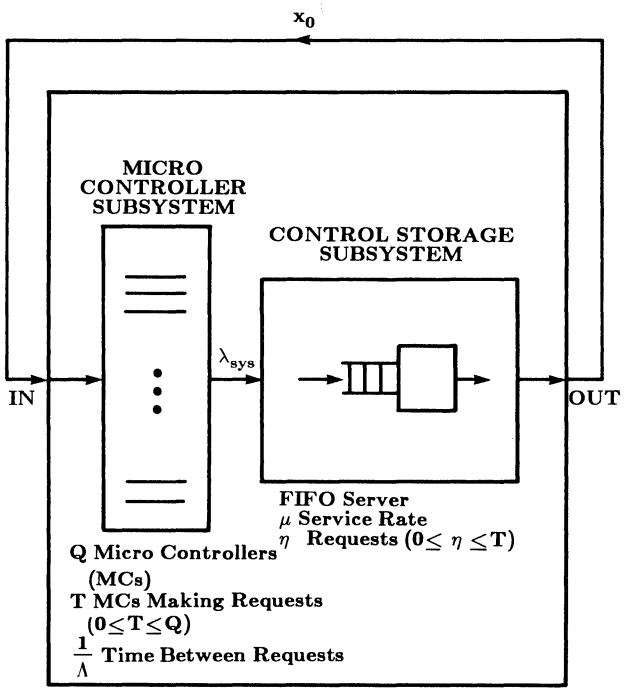


Figure IV.1: Two-station cyclic network which describes the interaction between the MCs and the Control Storage. The combined page request rates of the MCs is λ_{sys} and the throughput of the network is X_0.

The Control Storage subsystem services the page requests made by the MC subsystem. The service rate of the Control Storage subsystem is μ. The service queue at the Control Storage uses a FIFO queueing discipline. The number of requests in the Control Storage subsystem at a given time is η, where $0 \leq \eta \leq T$. The throughput of the network is X_0.

V. System Page Request Rate

In this section the queueing network model is analyzed to determine the average system page request rate. If all of the MCs are executing a different task (Q tasks executing), then the virtual page request rate for each of the MCs is Λ, i.e., $\nu_i = \Lambda$, where $0 \leq i < Q$. Therefore, the system page request rate is:

$$\lambda_{sys} = \sum_{i=0}^{Q-1}\lambda_i = \sum_{i=0}^{Q-1}U_i\nu_i = \sum_{i=0}^{Q-1}U_i\Lambda = \Lambda\sum_{i=0}^{Q-1}U_i,$$

Using the simplifying assumption that $U_i = U_{mc}$, a constant MC utilization, where $0 \leq i < Q$, then $\lambda_{sys} = QU_{mc}\Lambda$.

However, in the case of PASM, there are not usually Q independent tasks executing. For example, if an SIMD task of size RN/Q is being executed by the Parallel Computation Unit, the same instruction stream is being used by each of the R MCs which are controlling the task. The virtual page request rate to the Control Storage by the R MCs can be reduced from $R\Lambda$ to Λ by having one MC make the page requests and having the Control Storage broadcast the page to all R MC memory units simultaneously through the MC Memory System Switch (see Figure II.3).

To determine the actual average system page request rate it is necessary to determine the average number of independent instruction streams or tasks being executed by the MCs. This discussion will be limited to the execution of SIMD tasks. There are q+1 different sizes of SIMD tasks which can be controlled by the Q MCs, where $q = \log_2 Q$. A task may require 2^i MCs, where $0 \leq i \leq q$. Let the probability that an SIMD task requiring 2^i MCs is created be P_i. Let $\overline{E}_i$ be the average execution time for tasks which require 2^i MCs. The average value of the processor-time product for a task which requires 2^i MCs is defined as the product of the average execution time and the number of MCs required, $\overline{E}_i\, 2^i$. The processor-time product can then be used to weight the P_i distribution to determine R_i, the probability that a task requiring 2^i MCs will be executing on any MC which has a task assigned to it. R_i is defined as:

$$R_i = \frac{P_i \overline{E}_i 2^i}{\sum_{j=0}^{q} P_j \overline{E}_j 2^j} .$$

The average execution time parameters may be varied based on system use experience. For this analysis it is assumed that tasks of all MC requirements have equal execution time, so $\overline{E}_i = \overline{E}$. Therefore,

$$R_i = \frac{P_i \overline{E} 2^i}{\sum_{j=0}^{q} P_j \overline{E} 2^j} = \frac{P_i 2^i}{\sum_{j=0}^{q} P_j 2^j} .$$

In the analysis in this paper, a PASM with 16 MCs is assumed, i.e., q=4. For this analysis it is also assumed that distribution of the number of MCs required by a task is uniform, i.e., $P_i = 1/5$, where $0 \leq i \leq 4$. Once again, this assumption can be varied based on system use experience. The probability that a task requiring 2^i MCs is executing on a given assigned MC is $R_i = 2^i/31$, where $0 \leq i \leq 4$. The following theorem uses the above result for the probability that a task requiring 2^i MCs will be executing on any given assigned MC to determine the average system page request rate.

Theorem 1: The average system page request rate, $\overline{\lambda}_{sys}$, is:

$$\overline{\lambda}_{sys} = QU_{mc}\Lambda \sum_{i=0}^{q} \frac{1}{2^i} R_i ,$$

where Λ is the virtual task page fault rate, U_{mc} is the MC utilization for all MCs, and R_i is the probability that a task which requires 2^i MCs will be executing on any given assigned MC.

Proof: Consider an SIMD task which requires 2^i MCs. The set of MCs which is executing this task will be denoted by S_i. The virtual page fault rate for the task is Λ. When a page fault occurs, only one of the MCs in the set S_i reports the fault to the Control Storage Controller. If MC_j, $j \epsilon S_i$, is reporting the page faults, $\nu_j = \Lambda$ and $\nu_k = 0$ for all $k \epsilon S_i$ and $k \neq j$. Therefore, from the set S_i of 2^i MCs, only one page request is generated for each task page fault. Thus,

$$\sum_{k \epsilon S_i} \nu_k = \Lambda .$$

The average virtual page request rate for MC_j, where $j \epsilon S_i$ is defined as:

$$\mathrm{Avg}[\nu_j \mid j \epsilon S_i] = \frac{1}{2^i} \sum_{k \epsilon S_i} \nu_k = \frac{1}{2^i} \Lambda .$$

The notation Avg[x] denotes the average value of x. The average of the virtual MC page request rates, $\overline{\nu}$, can then be calculated by taking the average value of ν_j over all possible task sizes. Hence,

$$\overline{\nu} = \mathrm{Avg}[\nu_j] = \sum_{i=0}^{q} R_i \, \mathrm{Avg}[\nu_j \mid j \epsilon S_i] = \sum_{i=0}^{q} R_i \frac{\Lambda}{2^i} .$$

The system page request rate, λ_{sys}, is defined as:

$$\lambda_{sys} = \sum_{j=0}^{Q-1} \lambda_j = \sum_{j=0}^{Q-1} U_j \nu_j .$$

Assuming that the utilization for all of the MCs is the same, i.e., $U_j = U_{mc}$, where $0 \leq j < Q$, the average value of the system page request rate, $\overline{\lambda}_{sys}$, is:

$$\overline{\lambda}_{sys} = \mathrm{Avg}[\lambda_{sys}] = \mathrm{Avg}[\sum_{j=0}^{Q-1} U_j \nu_j]$$

$$= \sum_{j=0}^{Q-1} \mathrm{Avg}[U_j \nu_j] = \sum_{j=0}^{Q-1} U_{mc} \overline{\nu} = QU_{mc}\overline{\nu} .$$

Substituting in the equation for $\overline{\nu}$,

$$\overline{\lambda}_{sys} = QU_{mc} \sum_{i=0}^{q} R_i \frac{\Lambda}{2^i} = QU_{mc}\Lambda \sum_{i=0}^{q} \frac{1}{2^i} R_i ,$$

which is the desired result.

□

It is noted that the system page request rate is dependent on Q, the number of Micro Controllers and is independent of N, the number of Parallel Computation Unit processors. Theorem 1 is generalized to account for the fact that the task virtual page fault rate Λ may vary for tasks requiring different numbers of MCs in the following corollary.

Corollary 1: The average system page request rate, $\overline{\lambda}_{sys}$, is:

$$\overline{\lambda}_{sys} = QU_{mc} \sum_{i=0}^{q} \frac{\Lambda_i}{2^i} R_i ,$$

where Λ_i is the virtual page fault rate for the instruction stream of a task which requires 2^i MCs.
Proof: Follows directly from the proof of Theorem 1. □

When an MC is not executing, it is either waiting for a page request to be serviced by the Control Storage or it does not have a task assigned to it to execute. The *virtual utilization* is the utilization of the MC while it has a task assigned to it and is denoted by U'_{mc}. The *assignment ratio* is the fraction of time an MC has a task assigned to it. If $\overline{A}$ is the average MC assignment ratio, then $U_{mc} = \overline{A}\, U'_{mc}$. Note that if the MCs are always assigned tasks, then $U_{mc} = U'_{mc}$. The (real) page fault rate for a task may now be defined as $U'_{mc}\Lambda$ since the virtual utilization only accounts for the time that a task is assigned to a group of MCs.

The multitasking level, T, is defined to be the number of tasks which are executing on the system at a given time. The average system page request rate may be defined in terms of $\overline{T}$, the average multitasking level, and $U'_{mc}\Lambda$, the (real) task page fault rate, to be: $\overline{\lambda}_{sys} = \overline{T}U'_{mc}\Lambda$. Combining this with the result of Theorem 1, the average multitasking level is determined to be:

$$\overline{T} = \frac{\overline{\lambda}_{sys}}{U'_{mc}\Lambda} = \frac{\overline{A}\,\overline{\lambda}_{sys}}{U_{mc}\Lambda} = \overline{A}\sum_{i=0}^{q} R_i \frac{Q}{2^i}.$$

Hence, instead of Q tasks executing, the average multitasking level is $\overline{T}$, which for the uniform distribution case with all MC assigned tasks ($\overline{A} = 1$) would be: 80/31 and the average system page request rate is:

$$\overline{\lambda}_{sys} = \overline{T}U'_{mc}\Lambda = \frac{80}{31} U'_{mc}\Lambda = 2.58\, U'_{mc}\Lambda,$$

where $U'_{mc}\Lambda$ is the task page fault rate.

In conclusion, in the case where all 16 MCs are executing tasks it might be expected that the average system page request rate would $16U'_{mc}\Lambda$, where $U'_{mc}\Lambda$ is the page fault rate for the task running on each MC. In this section it has been determined that the average page request rate for the system is only $2.58U'_{mc}\Lambda$ when there is a uniform distribution of task sizes. Hence, the average system page request rate is only 16.1 per cent of what might be expected when all 16 MCs are executing tasks. The worst case system page fault rate is $16U'_{mc}\Lambda$ which occurs when each MC is executing an independent task. On the other hand, when all MCs are executing the same task, the system page fault rate is $U'_{mc}\Lambda$. The average multitasking levels for a variety of P_i distributions are given in Table V.1.

Table V.1: Average multitasking level, $\overline{T}$, for a variety of task size distributions. The task size is the number of MCs a task requires. P_i is the probability that a task which requires 2^i MCs is created.

P_0	P_1	P_2	P_3	P_4	$\overline{T}$
0.20	0.20	0.20	0.20	0.20	2.58
0.00	0.25	0.25	0.25	0.25	2.13
0.00	0.00	0.33	0.33	0.33	1.72
0.00	0.00	0.00	0.50	0.50	1.33
0.00	0.00	0.00	0.00	1.00	1.00
1.00	0.00	0.00	0.00	0.00	16.00
0.50	0.50	0.00	0.00	0.00	10.67
0.33	0.33	0.33	0.00	0.00	6.86
0.25	0.25	0.25	0.25	0.00	3.75
0.23	0.23	0.23	0.23	0.08	3.38
0.50	0.00	0.00	0.00	0.50	1.88
0.00	0.50	0.00	0.00	0.50	1.78
0.00	0.00	0.50	0.00	0.50	1.60
0.00	0.00	0.00	0.50	0.50	1.14
0.00	0.00	0.50	0.50	0.00	2.67
0.00	0.00	0.00	1.00	0.00	2.00
0.00	0.00	1.00	0.00	0.00	4.00
0.00	1.00	0.00	0.00	0.00	8.00
0.00	0.33	0.33	0.33	0.00	3.34
0.10	0.25	0.30	0.25	0.10	2.96

VI. Optimal Control Storage Service Rate

Criterion for optimal memory management in multiprogrammed systems have been given in [2,4,8]. The optimum is characterized by maximal system service rate, and in turn by maximal processor utilization and minimal response time [4]. One such criterion is the 50% criterion. The *50% criterion* for optimal memory management states that in a multiprogrammed system with page request rate λ, the use of the CPU is "optimized" when the disk service rate $\mu = 2\lambda$ so that the disk is 50% utilized [8]. So for $\mu \leq 2\lambda$, as μ is increased, the system service rate is increased significantly, and for $\mu > 2\lambda$, as μ is increased, the system service rate does not increase significantly. Thus, $\mu = 2\lambda$ is considered "optimal."

For the class of systems studied in [8], it was determined that the utilization of the secondary storage which resulted in optimal memory management was 50%. In order to determine the appropriateness of the 50% criterion for the PASM MC secondary storage, MC utilization was used as a performance measure. Figure VI.1 is a graph of the MC utilization as a function of the Control Storage utilization which was generated from simulation data (details of simulation are in [11]). There are three optimal values for the Control Storage utilization in PASM: 32.5%, 50%, and 62.5%. They are all considered optimal since the increase in MC utilization resulting from a small decrease in Control Storage utilization is much less the the decrease in the MC utilization resulting from a small increase in the Control

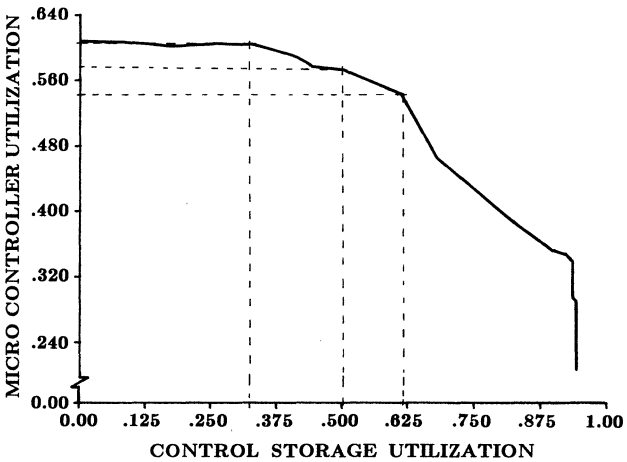


Figure VI.1: MC utilization as a function of Control Storage Utilization.

Storage utilization. The selection of the optimal Control Storage utilization to be used may depend on factors such as desired speed and cost of available secondary storage devices (e.g., disks).

It is noted that even when the Control Storage utilization is 0% (all page faults are serviced instantaneously), the MC utilization in Figure VI.1 is not 100%. This is due to other factors which impact the MC utilization besides the Control Storage utilization, such as availability of tasks to be scheduled [12] and fragmentation of MCs (i.e., available MCs do not form allowable group). For example, the MC utilization in Figure VI.1 could be increased if the task interarrival rate was increased, but its shape would remain similar.

To apply the optimal result to PASM, the Control Storage service rate μ must be selected so that $U_{cs}\,\mu = \overline{\lambda}_{sys} = \overline{T}\,U'_{mc}\,\Lambda$, where U_{cs} is the Control Storage utilization. Hence, $\mu = (\overline{T}\,U'_{mc}\,\Lambda) / U_{cs}$. If the optimal Control Storage utilization of 50% is selected, $\mu = 2\,\overline{T}\,U'_{mc}\,\Lambda$. In the case where there is a uniform distribution of the sizes of tasks created (derived in the previous sections), $\mu = 5.16\,U'_{mc}\,\Lambda$. If the virtual MC utilization is assumed to be one, then $\mu = 5.16\,\Lambda$. In actuality, U'_{mc} would be less than one, and a value other than one could be used here. Therefore, based on the assumption of a uniform distribution of the sizes of tasks created, the Control Storage service rate should be set to 5.16 times the virtual task page fault rate.

VII. Micro Controller Idle Time

Since the MCs in PASM are not multiprogrammed, there is not another task for an MC to execute while it is waiting for a page request from its current task to be serviced by the Control Storage. In this section, operational analysis is used to determine the MC idle time which results from an MC waiting for a page request to be serviced by the Control Storage. Note that this does not include the time that the MC is idle while it does not have a task assigned to it. This derivation makes use of Little's Law and is similar to that of the "Interactive Response Time Formula" for a terminal system in [3]. Let $\overline{m}$ be the mean queue length for a device (including the request which is being serviced); let X_0 be the throughput of the device; and let R be the accumulated time at that device per request (time spent by the request in the queue of the device while waiting for service plus the service time of the device). Then Little's Law [3] is: $\overline{m} = X_0\,R$.

Let I denote the average MC idle time and Z denote the average time interval between when an MC resumes execution after a page request is serviced and when its next page fault occurs. Hence, Z is the average execution time or busy time between page faults for a given MC. Each task is executed by a group of one or more MCs. Since each MC can have at most one instruction stream associated with it, the system has a finite customer population [6] (i.e., there is a finite number of page requests waiting to be serviced by the Control Storage at any given time since each MC cannot have another page fault while it is waiting for its current request to be serviced).

A task repeats "busy-idle cycles" while it has a group of MCs assigned to it. A *busy-idle cycle* has two phases: the busy phase, when the group of MCs which is assigned to the task is executing, and the idle-phase, when the group of MCs is waiting for a page fault to be serviced. The mean time for a task to complete one busy-idle cycle on a group of MCs is I + Z. Note that Z is the average time spent in the MC subsystem and I is the average time spent in the Control Storage subsystem during each busy-idle cycle (see Figure IV.1). Since all tasks which are assigned to MCs are repeating busy-idle cycles, $\overline{m}$ is equal to $\overline{T}$, the average multitasking level. Let X_0 be the throughput of the Control Storage. Applying Little's Law, $\overline{T} = X_0(I + Z)$.

The throughput, X_0, is the product of the utilization, U_{cs}, and the service rate, μ. So the throughput of the Control Storage is $U_{cs}\mu$. The average execution time between page faults, Z, is $1/\nu$ where ν is the MC virtual page fault rate. Therefore, the average MC idle time is:

$$I = \frac{\overline{T}}{X_0} - Z = \frac{\overline{T}}{U_{cs}\mu} - \frac{1}{\nu}\,.$$

This maps to the "Interactive Response Time Formula" in [3] by letting the MC busy time correspond to the user think time, the MC idle time correspond to the user wait time, the number of tasks executing (i.e., multitasking level) corresponds to the number of terminals, and the Control Storage throughput corresponds to the throughput of the central server.

Suppose the results for the optimal service rate μ from the previous section are used. Then the Control Storage service rate $\mu = 5.16\Lambda$ and

$$I = \frac{\overline{T}}{U_{cs}5.16\Lambda} - \frac{1}{\Lambda} = \frac{\overline{T} - U_{cs}5.16}{U_{cs}5.16\Lambda}\,.$$

Next the worst case situation is considered. In the worst case the Control Storage is completely utilized, so $U_{cs} = 1$ and

$$I = \frac{\overline{T} - 5.16}{5.16\Lambda}\,.$$

If the Control Storage is completely utilized, the system page fault rate λ_{sys} must be greater than or equal to the Control Storage service rate μ, so $\lambda_{sys} \geq \mu$. Hence, based on the 50% criterion and the assumptions used to select the Control Storage service rate μ, $\lambda_{sys} \geq 5.16\Lambda$ and $\overline{T} \geq 5.16$. If during some time interval there are 16 tasks executing (i.e., a multitasking level of 16), the MC idle time during that time interval would be:

$$I = \frac{16 - 5.16}{5.16\Lambda} = \frac{10.84}{5.16\Lambda} = 2.1\,\frac{1}{\Lambda}\,.$$

The time which an MC is busy executing a task is the time between page faults, $1/\Lambda$. The time which an MC is waiting for a page request to be serviced is its idle time, I. During a time interval when there are 16 tasks executing, the fraction of time which a given MC would be idle is:

$$\frac{I}{I + (1/\Lambda)} = \frac{2.1(1/\Lambda)}{3.1(1/\Lambda)} = 0.677\,.$$

So, if during some interval of time the average multitasking level was 16 (i.e., worst case level), the MCs would be idle 67.7% that time interval. Based on the simplifying assumptions in Section V used to compute the optimal Control Storage service rate μ, the probability of the worst case occurring is less than 0.1%. Note that if the probability had been greater, it would have impacted the calculation for the average system page fault rate $\overline{\lambda}_{sys}$ which would have resulted in a faster

Control Storage service rate μ.

VIII. Simulation Results

In this section results from the PASMOS simulator [11] are given. The simulator has been run with a variety of average execution times and distributions for the number of MCs a task requires. The results of all runs agree with the analytical result of Section V. As an example, consider the following simulation run where a random number generator was used to produce a uniform distribution (i.e., $P_i = 0.2$) for the number of MCs a task requires and the expected execution time for the tasks was fifteen seconds. The simulation ran for twenty thousand simulation seconds and over two thousand tasks were executed. The resulting distribution for the number of MCs required by a task for the simulation was: $P_0 = 0.191$, $P_1 = 0.204$, $P_2 = 0.198$, $P_3 = 0.208$, and $P_4 = 0.199$. The resulting average execution time for a task which requires 2^i MCs for the simulation was: $\bar{E}_0 = 15.006$, $\bar{E}_1 = 15.518$, $\bar{E}_2 = 14.215$, $\bar{E}_3 = 14.715$, and $\bar{E}_4 = 15.869$. The average MC assignment ratio $\bar{A}$ was 0.606 and the average multitasking level $\bar{T}$ was 1.531 streams. (Further details of the simulator are beyond the scope of this paper and are given in [11]. A description of the task scheduling algorithm which was used by the simulator is given in [12].)

Using the equation from Section V for R_i, the probability that a task which requires 2^i MCs is executing on a given assigned MC, with the $\bar{E}_i$s and P_is from the simulation, the results are found to be: $R_0 = 0.030$, $R_1 = 0.066$, $R_2 = 0.118$, $R_3 = 0.256$, and $R_4 = 0.529$. Substituting the MC utilization and the R_is into the equation for the average multitasking level:

$$\bar{T} = \bar{A}\sum_{i=0}^{q} R_i \frac{Q}{2^i} = (0.606)\sum_{i=0}^{4} R_i \frac{16}{2^i} = 1.530 ,$$

the average multitaking level in found to be 1.530.

Hence, by using the analytical method of Section V with the system characteristics from the simulation it has been determined that the average number of independent instruction streams is 1.530. The simulation results give the average number of instruction streams to be 1.531. Therefore, the simulation results support the analysis in Section V.

The simulator may also be used to confirm the worst case MC idle time result from Section VII. The Control Storage service rate and task page fault rate were selected so that $\mu = 5.16\Lambda$. To create the worst case situation, the distribution of the number of MCs required by a given task was adjusted so that all tasks would require one MC. The average execution time was adjusted so that the assignment ratio for all MCs would be one and the average number of tasks executing, $\bar{T}$, would approach sixteen. The resulting average multitasking level was 16.0; the Control Storage utilization, U_{cs}, was 1.0; and the average MC was idle for 67.5% of the simulation time. This result agrees with the expected result from the analysis in Section VII, where in the worst case (i.e., the multitasking level is 16) the average MC was idle 67.7% of the time. Again it is noted that the probability that this worst case would occur is very small.

IX. Relation to the General Multiple-SIMD Model

A general model of a multiple-SIMD system is shown in Figure IX.1. There is a pool of control units with a common secondary storage, a pool of processing elements, a switch which is used to connect a control unit to a group of processing elements, and an interconnection network for communication among the processing elements. In the case of PASM, the switch is fixed in that each processing element is connected to exactly one control unit (MC), and large machines are created by combining control units (MCs). Other MSIMD systems, such as MAP [9], use a crossbar type of switch to assign the processing elements to the control units. All of the control units are not always used. When PASM is executing an SIMD task only one of the MCs which is executing the task is used to make requests for pages.

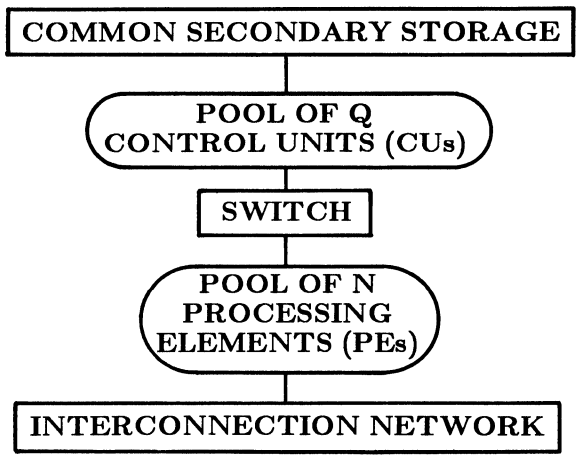


Figure IX.1: A general model of a multiple-SIMD machine.

The optimal service rate analysis may be applied to the general case by letting the used control units correspond to the MCs which are making the requests and the unused control units correspond to the MCs which are executing tasks but not making page requests. Thus the analysis applies to the general case where the SIMD machines have a power of two processing elements. The power of two constraint may also be eased to allow any size SIMD machine.

X. Conclusion

In this paper a queueing network has been analyzed to determine the "optimal" page request service rate for the Control Storage of the PASM multimicrocomputer system. It has been shown that the optimal service rate for the PASM Control Storage is much lower than might be expected. Two possible methods for varying the Control Storage service rate include varying the number or type of disks, or changing the method of storing pages on the disks. Simplifying assumptions were made about average execution time, task page fault rate, the distribution of the number of MCs which a task requires, etc. Based on experience any or all of these assumptions can be changed to reflect actual or expected system characteristics. Operational analysis has been used to determine the MC idle time which results from MCs having to wait for page requests to be serviced. Simulation results have been given which support the analytical result for the average number of independent instruction streams and the worst case MC idle time.

This study can also be applied to the use of PASM in the MIMD mode of operation or in a combination of

the MIMD and SIMD modes. When PASM is operating as a number of virtual MIMD machines of varying sizes, the MCs may be used to help coordinate the activities of the Parallel Computation Unit processors. The coordination activity of MIMD mode requires the MCs to execute significantly fewer instructions than in the control activity of SIMD mode. Hence, the page request rate is significantly lower for MIMD mode than for SIMD mode. Since it is expected that in MIMD mode each MC will have its own instruction stream, it can be treated as one SIMD partition, and incorporated into the run-time statistics.

In summary, a model has been developed for the PASM control system memory hierarchy. Using any combination of system feature assumptions and actual system characteristics (from experience), the model can be used to determine the "optimal" service rate for the Control Storage. Furthermore, using the model, values for the parameters which characterize the expected task environment and secondary storage service rate can be varied to determine the impact on MC utilization. The model can be adapted for use in any multiple-SIMD machine with common secondary storage for the multiple control units.

Acknowledgment

The authors would like to thank Prof. Peter J. Denning, Prof. Dorothy E. Denning, George B. Adams III, and Robert J. McMillen for their comments and suggestions.

References

[1] P. J. Denning, "Virtual memory," *Computing Surveys,* vol. 2, pp. 153-189, Sep. 1970.

[2] P. J. Denning, "Working sets past and present," *IEEE Trans. Soft. Engr.,* vol. SE-6, pp. 64-84, Jan. 1980.

[3] P. J. Denning and J. P. Buzen, "The operational analysis of queueing network models," *Computing Surveys,* vol. 10, pp. 225-262, Sep. 1978.

[4] P. J. Denning and K. C. Kahn, "An L=S criterion for optimal multiprogramming," *Int'l. Symp. Comp. Performance, Modeling, Measurement, and Evaluation,* ACM, Mar. 1976, pp. 219-229.

[5] M. J. Flynn, "Very high-speed computer systems," *Proc. IEEE,* vol. 54, pp. 1901-1909, Dec. 1966.

[6] L. Kleinrock, *Queueing Systems, Vol. 1: Theory,* John Wiley and Sons, Inc., New York, 1975.

[7] J. T. Kuehn, H. J. Siegel, and P. D. Hallenbeck, "Design and simulation of an MC68000-based multimicroprocessor system," *1982 Int'l. Conf. Parallel Processing,* Aug. 1982, to appear.

[8] J. Leroudier and D. Potier, "Principles of optimality for multiprogramming," *Int'l. Symp. Comp. Performance, Modeling, Measurement, and Evaluation,* ACM, Mar. 1976, pp. 211-218.

[9] G. J. Nutt, "Microprocessor implementation of a parallel processor," *4th Symp. Comp. Architecture,* Mar. 1977, pp. 147-152.

[10] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller, Jr., H. E. Smalley, Jr., and S. D. Smith, "PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Trans. Comp.,* vol. C-20, pp. 934-947, Dec. 1981.

[11] D. L. Tuomenoksa, *Design and Analysis of an Operating System for a Reconfigurable Multimicrocomputer System,* Ph.D. Dissertation, Purdue University School of Electrical Engineering, in preparation.

[12] D. L. Tuomenoksa and H. J. Siegel, "Analysis of multiple-queue task scheduling algorithms for multiple-SIMD machines," *3rd Int'l. Conf. Distributed Computing Systems,* Oct. 1982, to appear.