ISSN 0190-3918
also listed under
IEEE Catalog Number 81CH1634-5
Library of Congress Number 79-640377

INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING

PROCEEDINGS

OF THE

1981 INTERNATIONAL CONFERENCE

ON

PARALLEL PROCESSING

August 25–28, 1981

Ming T. Liu and Jerome Rothstein
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

1981 INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING
Ming T. Liu and Jerome Rothstein
ISSN 0190-3918
also listed under
IEEE Catalog Number 81CH1634-5
Library of Congress Number 79-640377
IEEE COMPUTER SOCIETY PRESS

PROCEEDINGS

OF THE

1981 INTERNATIONAL CONFERENCE

ON

PARALLEL PROCESSING

Ming T. Liu and Jerome Rothstein
Ohio State University
Editors

Papers presented on
August 25-28, 1981

Co-Sponsored by

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

and the

ISSN 0190-3918
also listed under
IEEE Catalog Number 81CH1634-5
Library of Congress Number 79-640377

IEEE Computer Society

In Cooperation with the

Association for Computing Machinery

IEEE Catalog No. 81CH1634-5
ISSN 0190-3918

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.I 548a 001.6′4 79–640377
MARC–S

Library of Congress 79

IEEE Catalog Number 81CH1634-5
Library of Congress Number 79-640377
Computer Society Number 354
ISSN 0190-3918

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

Tenth anniversaries are traditional occasions for reflecting on the past, evaluating trends in the current situation, and speculating on the shape of things to come. The last ten years have seen the growth and maturation of the International Conference on Parallel Processing from a two-day invitational meeting of RADC contractors (1972) at which seventeen papers on diverse aspects and applications of the Rome Air Development Center Associative Processor (RADCAP) were presented, to a truly international meeting covering all phases of parallel and distributed processing. There were 136 papers submitted to this conference, an all-time record, compared to 117 in 1980, 93 in 1979 and 1978, and over 80 in 1977 and 1976. The number of papers from abroad has also grown, being 34, 31, and 23 in 1981, 1980, and 1979 respectively, from 16, 10, and 10 countries. As in previous years, the high quality of the papers submitted made final selection extremely difficult. We wish to thank the 214 referees, including 87 non-authors, for their indispensable aid in selecting the 66 finalists for the 14 sessions of contributed papers. Each manuscript submitted was sent to three referees; only their prompt and conscientious help, despite their other obligations, made it possible to have the proceedings available on time.

An innovation this year, replacing the traditional keynote speech of the ceremonial session the evening before regular sessions begin, is the panel of five invited speakers on the history of parallel processing. Professor Tse-yun Feng organized it, for which we are most grateful. He is acutely aware that much important history is inevitably hidden from those participating in it, and therefore requests help from attendees and others, in compiling as complete a record as possible. Now is the time to do it, before memories fade and pioneers pass on.

Another innovation is the tutorial on parallel processing the day before the conference. Such tutorials have become increasingly popular adjuncts of many meetings, and this one was set up in response to suggestions made by attendees of previous conferences. We hope it establishes a new tradition of excellence.

A special issue of the IEEE Transactions on Computers on Parallel and Distributed Processing is planned for December 1982. Professors Ming T. Liu and Jerome Rothstein are the guest editors. Papers presented at this conference or modifications thereof will be considered for inclusion as will others submitted by respondents to this and other calls for papers appearing elsewhere. The closing date for submission of manuscripts is January 1, 1982. One hundred pages have been allocated to the special issue; we hope it will be of permanent reference value.

The growth of interest in parallel and distributed processing in the last decade has been explosive, and will doubtlessly continue unabated. This conference could easily have grown very large, with parallel sessions and many more papers. However, the attendees have voted, year after year, against departing from the traditions of no parallel sessions, emphasis on attendance by active workers in the field, and of holding the conference far from the competing attractions of a metropolitan or resort milieu. The opportunities for prolonged, intense, personal interactions with established and upcoming researchers were felt to outweigh disappointments like being put on a waiting list and not being able to attend because of the rarity of cancellations. All this can change in the future, but only if the attendees wish it to.

We would like to thank Dean Donald D. Glower, College of Engineering, The Ohio State University for his constant encouragement, and Professor Tse-yun Feng for his sage advice and counsel about the endless details of managing this enterprise. The assistance of Professor Chuan-lin Wu is also appreciated. Last, but not least, we appreciate the devoted assistance of Jy-jine Lin in computerizing so much of the routine involved.

Ming T. Liu and Jerome Rothstein
Technical Program Co-Chairmen

TABLE OF CONTENTS

Page

SESSION 1: HISTORY OF PARALLEL PROCESSING

SESSION 2: LANGUAGES AND COMPILERS

SESSION 3: DISTRIBUTED SYSTEMS AND NETWORKS

SESSION 4: NUMERICAL ALGORITHMS

SESSION 5: ASSOCIATIVE PROCESSORS AND PROCESSING

SESSION 6: MULTIPROCESSOR ARCHITECTURES

SESSION 7: RECENT RESULTS I

SESSION 8: NON-NUMERICAL ALGORITHMS

SESSION 9: SPECIAL-PURPOSE PROCESSORS

SESSION 10: INTERCONNECTION NETWORKS

SESSION 11: VLSI ARCHITECTURES

SESSION 12: ARRAY PROCESSORS AND PROCESSING

SESSION 13: Recent Results II

SESSION 14: PERFORMANCE EVALUATION

SESSION 15: SCHEDULING

AUTHOR INDEX

LIST OF REFEREES

M.A. Abidi
W.B. Ackerman
S. Afshar
D. Agrawal
A. Andrews
Arvind
J.L. Baer
E.E. Balkovick
B.W. Ballard
U. Banerjee
J.A. Bannister
T.P. Barnwell
K.E. Batcher
H.K. Berg
T.S. Berk
S.Y. Berkovich
B Berra
A.T. Berztiss
B. Bhargava
L. Bic
A. Bilgory
J.G. Bonar
F.A. Briggs
M E. Brown
R.E. Bryant
F.J. Burkowski
S.E. Butner
P. Chan
K.M. Chandy
T.L. Chang
I.N. Chen
K.W. Chen
F. Chin
J.C. Chou
W. Chou
A. Chow
Y.C. Chow
T.W. Christopher
W.W. Chu
H. Chang
E.M. Clarke, Jr.
D. Cohen
S. E. Conry
F.C. Crow
K. Culik
D. Degroot
E. Dekel
N. Deo
N. Dimopoulos
P.J. Drongowski
M. Dubois
D.D. Dunlop
R.L. Earle
C.S. Ellis

M Evens
G.M. Fachs
K.M. Falavarjani
J. Fawcett
D. Fisher
B.E. Flinchbaugh
C C. Foster
C.R. Foulk
M.A. Franklin
P. Frederickson
M. Freeman
H.C. Fu
D. Fussell
L.W Fung
D.D. Gajski
D. Gannon
O.N. Garcia
H. Gerhauser
M. Gerla
B.K. Gilbert
M.J. Gonzalez, Jr.
R. Gordon
A. Gottlieb
P. Greene
J.P. Hayes
C.J.M. Hodges
L.A. Hollaar
P. Hsia
T.C. Hu
J.C. Huang
K. Hwang
K.B. Irani
R.C. Jaeger
R. Jain
B. Jayaraman
S.F. Jennings
A.K. Jones
H.F. Jordan
R.N. Kapur
S.P. Kartashev
J.L. Kennedy
D.S. Kerr
D. Klappholz
J.C. Knight
H. Kobayashi
H.S. Koch
M. Krieger
R. Kuhn
J. Kuo
A.J. La Salle
D.H. Lawrie
C.C. Lee
H. Lee
K.Y. Lee

D. Leinbaugh
S.P. Levitan
R. Lian
L. Lilien
S.L. Lillevic
J.J. Lin
M W. Linder
G.J. Lipovski
J.W.S. Liu
K.Y. Liu
T.S. Liu
V. Lo
T.N. Long
B.D. Lubachevsky
N. Magid
S. Makam
M. Malek
B. Malm
P.N. Marinos
R.C.O. Martins
R.J. McMillen
R.E. Merwin
J.H. Mirza
S. Mittal
C. Mohan
R.K. Montoye
W.W. Myre
A. Mukhopadhyay
K.J. Mundell
W. Murphy
V.P. Nelson
L.M. Ni
C.N. Nikolaou
A.A. Nilsson
E.D. Nugent
M.J. O'Donnell
W.F. Ogden
Y. Oh
A.E. Oldehoeft
E. Oliver
T. Ozsu
D.A. Padua
E.W. Page
J.H. Patel
D. Paulish
D.J. Pease
C.E. Perkins
D.K. Pradhan
F.P. Preparata
R.W. Priester
C.S. Raghavendra
C.V. Ramamoorthy
J. Ramanathan
C.C. Reames

S.M. Reddy
A.P. Reeves
H.K. Reghbati
L. Rudolph
S.K. Sahni
A. Sameh
D.H. Schaefer
S.A. Schuman
K.G. Shin
S.G. Shiva
H.J. Siegel
L.J. Siegel
D.P. Siewioriek
A. Silbershatz
M.L. Skinner
B.W. Smith
C.H. Smith
D.R. Smith
L. Snyder
S. S. Soo
S. Sowrirajan
J. Spragins
V.P. Srini
J.A. Stankovic
N.C. Strole
S. Su
C. Sunshine
J.M. Surprise
P.H. Swain
E. Swartzlander
A.Y. Teng
G.S. Tjaden
H.C. Torng
W.N. Toy
R.H. Travassos
S.K. Tripathi
D.P. Tsay
L. Uhr
L.D. Umbaugh
A.V. Veidenbaum
P.S. Wang
D.F. Wann
R.G. Wedig
Y. Wei
B.W. Weide
H.O. Welch
L.D. Wittie
M. Wolfe
C.L. Wu
S.S. Yau
P.C. Yew
M. Yuschik

PARALLELISM IN COMPUTING

Sidney Fernbach
Control Data Corporation
Livermore, California 94550

Abstract. The parallelism in computers is reviewed from the early systems, such as the Univac I to the present day systems. Some degree of parallelism has always existed, sometimes for reliability, at other times for improved performance. With the highly reliable components currently in production, the main reason for today's parallelism is to obtain as high a performance as possible for the dollar.

There has always been a degree of parallelism in digital computers as we know them today. At first it was for reliability purposes, later to achieve greater performance as well as for reliability.

The first commercially available computer was the Univac I designed initially for Census Bureau work. It was designed as a decimal machine, having 6 bits to represent alphanumeric characters. Because of the fact that it used mercury delay lines, the machine was highly serial, sending bits down the delay lines one by one. On the other hand, there was more duplication of curcuits in Univac I than in most machines built since. Checking was provided by automatic comparison of results coming out of duplicate arithmetic circuits. This of course was done for reliability purposes, there being 5600 vacuum tubes in the system. Incidentally this structure was also true of the BINAC which was conceived earlier than the Univac.

Other computers of the same vintage, (late 40's and early 50's) used either relays, drums or delay lines and were in the most part serial in nature. When the electrostatic tube came into use, soon thereafter, most machines were binary in nature; fetching, storing and operating on words in a parallel mode. The earliest of these seem to have been the Bureau of Standards SEAC and MIT Whirlwind. Others soon followed -- mostly the IAS family of computers as well as some of international flavor such as those built in Manchester, England. The commercial vendors quickly came out with their version; IBM with the 701 and ERA with the 1103. These were 36 bit binary computers. For the most part these machines were highly serial.

It was recognized even in the early 50's that performance could be gained through more parallel operations, but few designers or manufacturers thought it important enough to go all out for performance. Early machines were pushed strongly by the Department of Defense for use in cryptographic work. Later the AEC, needing much higher performance than that made available with the 701/704 or 1103/1103A started to stir the pot with specially built systems incorporating parallel design. One of the first of these was the LARC. A version of this was specified by the Lawrence Livermore National Laboratory in early 1955. It called for a number of processors sharing a common memory. The initial specs were far more demanding then those that ended up in the machine that was finally built. They required both binary and decimal arithmetic units, for example. The final version was an all decimal machine allowing for two CPU's and an I/O processor to function concurrently. Unfortunately there never was enough money in the budget to acquire a 2 CPU system, although hardware allowances were made for the addition of a second unit. The two LARC's eventually built and delivered (in 1960) had but one CPU and the one I/O processor. The memory of the system could have up to 39 independently addressable parts each of 2500 words for a total of 97,500 words. The delivered systems had only 30,000 words. Input-output was taken care of by the issuance of summary commands to the processor unit. The CPU's alerted the I/O Processor to their presence and also checked for completion. Memory overlapping allowed for one instruction to be executed while the operand address was being transferred to/from memory and the operand address of another instruction was being indexed. The memory bus was time slotted so that systems had access to one or more of the 8 time slots of 0.5 sec. each.

The main back-up memory in this system consisted of up to 24 magnetic drums which allowed for 3 read and 2 write operations to take place concurrently.

The IBM 7030 or STRETCH was designed and built at the same time as LARC. It also incorporates a great deal of parallelism. The most interesting is the look-ahead feature. While one instruction is being executed several more may be fetched and interpreted. Unfortunately branching, if it occurs, forces the look-ahead to undo what it had already done.

Another interesting machine designed in the 50's was the Gamma 60, designed by Compagnie des Machines Bull in Paris. This system consisted of a variable number of independent and different processors, sharing common two-way distribution busses. The processors did not have to be identical; as a matter of fact there were four different types. The Central Control Unit had 2 major subunits, the Transfer Distributor (TD) and the Program Distributor (PD). Priority decisions were made in this unit, data transfer requests being handled by the TD and instruction requests by the PD.

Another interesting machine of the same vintage was the RW-400 or Polymorphic Data System. This was built by Ramo-Wooldrich Computers. It used a large cross bar switch to interconnect computer Modules/Buffer Modules with peripheral device modules. One of the computer modules acts as master and the

0190-3918/81/0000/0001$00.75 © 1981 IEEE

others as slaves. Any data from peripherals may be requested, stored in a buffer module until needed then moved directly to the requested computer module.

The National Bureau of Standards also built a multiprocessor called PILOT. It had 3 processors, each different from the others. One processor was the arithmetic unit, another the housekeeping unit and the third the I/O processor.

The LARC and STRETCH served the scientific world well for a number of years, despite the fact that each had its problems and was delivered late (1960-1961). By that time the transistor generation was upon us and numerous highly capable machines were on the market. None of them matched the LARC and STRETCH in performance, but their levels were gradually being reached. Some of the features in these two systems crept into others.

During this same period of time there were non-general purpose commercial machines that were also being built with parallel features, but I am not going to discuss these here. For example, FAA had a unique requirement for utmost reliability and hence multiple systems usually were built for this agency. Also the seismic industry had great need for high performance "vector" type of operations performed on Array Processors attached to standard equipment.

The first big jump in performance by way of concurrency after LARC/STRETCH was found in the CDC 6600. This machine had 10 functional units as well as 10 peripheral processors. Each peripheral processor had its own memory for programs and for buffer space. Each can interrupt the central processor and monitor the central program address. Each PP takes one minor cycle (100 ns) or 1/10 of the major cycle as its slot to perform one of its steps.

The functional units consist of 2 multiple, 2 add, 1 divide, 1 shift, 1 branch, 1 Boolean, and 2 increment units which can be operating concurrently, each being initiated at the start of a minor cycle.

The 7600 was a follow on the 6600 with higher speed components. In organization it was very similar. The chief difference was in the memory organization, a high speed memory of 64K words was backed up by a large 512K word slower memory. Again parallelism came to the rescue. Eight word "swords" could be read out of large core with one instruction. There was also a high speed swap that enabled communication between the two memories to permit operations at high speeds. The 7600 was about 5 times the 6600 in performance. IBM had less parallelism in its equivalent level machines named 360/91(95) and 370/195.

Even as these machines were being designed and built, there were other efforts to provide even greater parallelism. Dan Slotnick, from whom you will hear the historical background in more detail and with more accuracy, had designed a system he called SOLOMON. This was accomplished while he was at Westinghouse, although the ideas had been percolating in his mind while he was still at IBM.

This system was to have 1000 processors, each with its own memory operating in unison at commands of a central instruction issuing unit. They worked in lock-step fashion, such that, when an add instruction came along, each did its add using operands in its own memory, concurrently with the others. Thus a factor of 1000 could be achieved in performance over a single processor (if all could be in operation simultaneously). There was a lockout feature, so that if 1000 pairs of operands were not available, some processing elements would remain idle. To handle certain types of mathematical problems more effectively, each processor was enabled to communicate with its 4 nearest neighbors.

The Lawrence Livermore Laboratory initiated attempts to have DOE (then AEC) order a system from Westinghouse. Unfortunately, when the top management at Westinghouse learned how much money IBM and Univac were supposed to have lost on STRETCH and LARC, respectively, the corporation got cold feet and backed out. LLL, being still interested in the concept of parallel processors tried to find other manufacturers to build a system. IBM showed some interest and had one of its excellent architects, Jim Pomerene design a SOLOMON-like machine. It incorporated all the latest technology IBM had come up with for the 360/90 system. Instead of 1000 processors, only 32 were proposed, but these were each very powerful units in their own right. This PNDC, as it was called also never saw the light of day. Other computer researchers and designers at IBM decided that pipelined structures were better than a parallel network of processors. They convinced the IBM management to give up PNDC. For a time this seemed like the end of the road. DOE (AEC), NSF, and ARPA representatives met to discuss the situation and to decide whether it might be possible to join forces in having a SOLOMON-like machine built. Before either AEC or NSF could collect its resources, ARPA, with Ivan Sutherland in the lead was off and running. John Foster, who was head of D.D.R. and E. invited a group of "experts" in to decide on whether or not to fund such a computer. The decision was "go". The resulting machine was ILLIAC IV, originally intended for Dan Slotnick's lab at the University of Illinois, but installed upon completion at NASA/Ames in California instead. The initial intent in this system was to have 256 processors, each with 2000 words of memory. Because of rising costs, the number of P.E.'s was reduced to 64.

One interesting feature of the ILLIAC IV which does not usually get much attention is the high performance disk associated with it. Early in the actual operation of ILLIAC IV, it was found that the 2000 word memory was too small, so the disk-file subsystem actually was made the main memory. There were dual files, each with $5X10^8$

bits of storage capacity and each being to sustain a data flow rate of 500 megabits/sec. Since the data path to the array was 1 billion bits wide, it was possible with proper synchronization to obtain a very high band width interchange with the processor memories. Used this way, the Illiac IV, for certain problems demonstrated performance not yet matched by more modern computers.

When Illiac IV was contracted for and got under way, it was considered an experimental machine. The intent was to learn to use such a system for solving large scale problems which kept growing in size. Since there was the chance for failure and since other concepts like "pipelining" were being proposed, LLL with the consent of AEC decided to try the alternate route of the pipelined machine. Again this was to be experimental. A contract was negotiated between LLL and Control Data Corporation which resulted in the STAR-100 computer system. Simultaneously, Texas Instruments which was involved in the early work on Illiac IV became interested in building a "pipelined" machine. With internal customers initially, T.I., went ahead with the project that resulted in the Advanced Scientific (or Seismic) Computer (ASC). Both of these machines had multiple pipe capabilities; the STAR relied on external processors to handle I/O, the ASC had its own peripheral processor. Performance on these machines for highly vectorized problems was very good. Scalar capabilities were very poor.

Overall impressions left with the computing community concerning the vector computing systems of the late 60's - early 70's were bad. Only one Illiac IV, 4 STAR - 100's, and 7 - T.I. - ASC's were delivered. It wasn't until the late 70's that faith was restored in high performance machines. Seymour Cray, now of Cray Research, Inc., was able to build a high performance scalar system thoroughly integrated with a vector system into a beautiful package. Now, scalar problems could run faster than on any other system, and if any degree of vectorization was possible, the additional parallelism improved performance substantially. Each, the scalar and vector processor had functional parallelism as well. Chaining of vector operations was also possible.

The realization of the need for scalar processing did not go unnoted by CDC. A new machine was designed to replace STAR-100. This was done in two steps. The first, resulting in Cyber 203 added a scalar unit to the two-pipe vector system and at the same time replaced the original core memory with a speeded-up semi-conductor memory. The second step resulting in the Cyber 205, replaced the vector unit with a faster LSI unit, now with up to 4 identical pipes.

Not to be outdone by the others, Burroughs Corporation, who had built the Illiac IV now decided to build a much superior version, named the Burroughs Scientific Processor (BSP). This machine was designed to have 16 processors. A new algorithm was employed in this system to permit access to memory with no conflict. This time the memory was accessible to all processors. Two levels of memory were employed; 524 K words of parallel processor memory and 4 M words of file memory. There were in addition to the central processor, an I/O processor and maintenance processor also. It was even possible to have two BSP systems tied together with a system manager. Actually the system manager was the front-end standard B7800 computer system. In this processing system there was no real scalar processor; one had to rely on the frontend. In this description I have used the past tense, because as of this time the BSP has been abandoned as a product.

Burroughs is not quite out of the large scale parallel processor design completely, as yet. There is an on-going effort to design a 1 Gigaflop machine for NASA/Ames to carry out Navier-Stokes calculations. This machine as described in earlier papers has 512 processors working concurrently. This may change, of course in this final year of preliminary design. Burroughs is competing with Control Data for this NASA contract; the Control Data design is more along the lines of the Cyber 200 series of machines. Burroughs has had much more experience in parallel systems. Besides the above mentioned systems, the Corporation built a PEPE prototype. This was a machine originally designed by the Bell Telephone Laboratory for use in Ballistic Missile Defense systems. Whether there will be a follow-on to PEPE is hard to say at this time. This was also a multiprocessor.

Other parallel processors have been designed and built primarily for special purposes such as image processing. One is currently being constructed by Goodyear for NASA/Goddard. This is a follow-on to a bit oriented machine called the STARAN. ICL in England also built a similar machine with 4096 processors. This one is called DAP - one is in operation at St. Mary's College in London.

Because of the great strides made in microprocessor development, performancewise as well as costwise, there are any number of attempts to assemble numerous microprocessors in a multiprocessor system. As with most computer concepts, multiprocessing is rather an old one. Some early versions have already been mentioned. Dual processors are commonplace, most manufacturers having tried their hands at these at some time.

The most ambitious attempts have been made by Carnegie-Mellon University, first with its C.MMP having 16 minicomputers tied together and later with its CM*, with 50 processors tied together in a number of modules. The hardware configurations are relatively easy to provide. The software provided the rub. Good system software and efficient algorithms for applications are much harder to devise.

Other recent attempts to provide high performance systems are those of CDC, Denelcor, and the Lawrence Livermore Laboratory. CDC was built and delivered and Advanced Flexible Processor (AFP) made of the same LSI components used in Cyber 205. This system consists of 4 modules each with variable functional units structured in a ring type architecture. The initial system was to be used for a special application and did not need floating point. The Denelcor system, also at this point in time designated in a 4 processor configuration is being constructed for the Aberdeen Proving Ground to be used in Ballistic Calculations.

The Livermore system called S-1 is being sponsored by the U.S. Navy to be used for signal processing. In this case 16 memories are being tied to 16 processors by a cross-bar switch.

Whether the parallelism being put into a multiprocessor is capable of being effectively utilized has yet to be demonstrated. Certainly the availability of a large number of processors at low cost implies that many can remain idle if the overall performance can be increased. It seems plausible that the future designs should incorporate "pipelined" processing in multiprocessing elements.

As for the future, we seem not to be making as much headway as we should. Kung and his systolic approach, Dennis and Company and their Data-Flow concepts seem to have much merit. It is too early to say that we will see such systems before the 90's -- but it seems unlikely. The more ingenious young people in their experiments with microprocessors no doubt will dream up better ways of designing parallelism into computers. Our main hope, however, is that the problem designers and software experts will help make it possible to take advantage of all these concepts in the not too distant future.

HISTORY OF PARALLEL PROCESSING AT GOODYEAR AEROSPACE

W. C. Meilander
Digital Systems Marketing
Goodyear Aerospace Corporation
Akron, Ohio 44315

Abstract

Associative memories have been talked about in scientific circles for a long time. This paper describes some of the first efforts to bring that talk into the realm of reality. At Goodyear Aerospace, we continue to develop techniques for fabricating and using associative memories. The techniques used in associative memories are presented, and the disadvantages of the methods used at any time are discussed. The associative memory (AM) logically leads to the associative processor (AP). Most of the advantages of AM's are retained in AP's, and many new capabilities are added. In fact, the AP is becoming a very powerful tool in handling the highly dynamic data bases of air surveillance and command and control systems.

The associative processing effort is augmented by the endeavors associated with the microcomputer array processor (MAP) and the massively parallel processor (MPP). The MAP and MPP broaden the capabilities of parallel processing into the fields of electronic warfare and image processing.

Background

Vannevar Bush made a strong case for associative processors in 1945[1]: "There is a growing mountain of research. But there is increased evidence that we are being bogged down today as specialization extends. The investigator is staggered by the findings and conclusions of thousands of other workers - many of which he cannot find time to grasp, much less to remember - as they appear. Yet specialization becomes increasingly necessary for progress, and the effort to bridge between disciplines is correspondingly superficial."

Dr. Bush continues: "But there are signs of change as new and powerful instrumentalities come into use." He then discusses many of the discoveries made in the past few centuries that have led to the increased activity of the 20th century. He cites the importance of communication in the scientific world with: "Mendel's concept of the laws of genetics was lost to the world for a generation because his publication did not reach the few who were capable of grasping and extending it; and this sort of thing is undoubtedly being repeated all about us, as truly significant attainments become lost in the mass of the inconsequential."

Dr. Bush considers the application of machines to "logical processes" with "formal logic used to be a keen instrument in the hands of the teacher in his trying of students' souls." He then describes approaches for selecting pertinent information from the mass of data available. His discussion of "memex instead of index" states: "Our ineptitude in getting at the record is largely caused by the artificiality of systems of indexing. When data of any sort are placed in storage, they are filed alphabetically or numerically, and information is found (when it is) by tracing it down from subclass to subclass. It can be in only one place, unless duplicates are used; one has to have rules as to which path will locate it, and the rules are cumbersome. Having found one item, moreover, one has to emerge from the system and re-enter on a new path.

"The human mind does not work that way. It operates by association. With one item in its grasp, it snaps instantly to the next that is suggested by the association of thoughts, in accordance with some intricate web of trails carried by the cells of the brain. It has other characteristics, of course; trails that are not frequently followed are prone to fade, items are not fully permanent, memory is transitory. Yet the speed of action, the intricacy of trails, the detail of mental pictures, is awe inspiring beyond all else in nature.

"Man cannot hope to fully duplicate this mental process artificially, but he certainly ought to be able to learn from it."

Early Effort

Many have agreed with Dr. Bush. Activity has continued since 1945 to develop the concepts espoused by him. These efforts, at best, only scratch the surface of the thoughts in Dr. Bush's paper. Yet, they appear to offer relief from some of the laborious indexing tasks that are bogging down our present endeavors to retrieve relevant data from an ever-increasing data base. An associative memory offers a system that allows retrieval of related data from a memory based on the data content in the memory. This can be understood when it's realized that an associative memory can directly implement a relational data base. In this context, the machine appears to emulate capabilities of the human mind.

Slade and McMahon described a cryotron catalog memory system in 1956[2]. This paper is generally accepted as the earliest record of a hardware approach to the problem of searching memory by content instead of address. The Western Reserve University (WRU) search selector, discussed below, may be an earlier effort.

In 1958, Goodyear Aerospace - while working with the concepts of associative memories - held a number of discussions with Dr. Jim Perry and Dr. Allen Kent. Perry and Kent were working with the techniques of information retrieval at the School

0190-3918/81/0000/0005$00.75 © 1981 IEEE

of Library Sciences at Western Reserve University. Their work covered one of the earliest associative processors fabricated. The concepts of their approach were presented in 1955. The machine developed was called the WRU search selector. It was designed to search a document data base.

The search selector$^{(3)}$ (Figure 1) was designed and built by Perry in 1956. It was a relay machine and used a Flexowriter tape reader to input the data base to be searched.

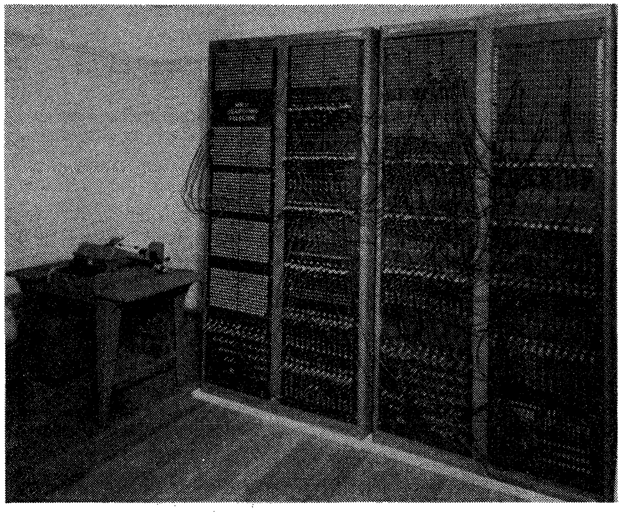


Fig. 1 - Western Reserve's Search Selector

The data base was formed from information abstracted from documents by knowledgeable reviewers. Keywords of the abstract were encoded (often by the same reviewers) via a dictionary into four character groups. The encoded information along with the document accession number was stored on punched paper tape.

Queries were encoded using the same dictionary. The queries were stored in the search selector. Ten independent queries could be entered at one time. The system provided for queries using logical AND, OR, and EXCLUSIVE-OR operators and combinations of these operators. The search selector program was entered through the patch cord system shown in Figure 1. After the query was programmed, the punched paper tape data file was passed through the system. Whenever a query was satisfied, the document accession number was read from the tape and typed along with the number of the query. The machine was used for several years in searching a file of documents for members of the American Society of Metals. A General Electric 225 computer replaced the WRU search selector about 1960.

Why Associative Memories?

The concept of associative memories derived from many different requirements. In the WRU machine, it was used for evaluating a coded request against a file of coded documents. In many other cases, the requirement, similarly, stemmed from the desire to search unordered data. At Goodyear, efforts were underway to find a method for locating items in memory on the basis of memory contents. This early activity was prompted by a desire to examine the present position of a large number of simulated targets being updated through a digital differential analyzer. The goal was to locate and display each target at the proper time as a simulated antenna scanned the space. The store would be searched in the azimuth field for the current azimuth and the associated target range read for display. The search needed to be completed in a few microseconds. We wanted a faster approach than software could provide.

Software Approaches

Software associative searches were performed in sequential machines when the amount of data stored was small. Breakthroughs in list processing were achieved when such techniques as hash coding, chained lists, and inverted files were implemented. These techniques eliminated the need for laborious searches of unorganized data (unless the field you were searching was not a key field). They also generated the complex file structures in use today, with their attendant complex update problems. The user does not realize the extent of the management software, since these complex file management structures are often a part of today's operating system.

Hardware Approaches

The desire to break away from the limitations of the sequential processor prompted much effort in the early 1960's. At that time, hardware techniques were advancing, and a variety of associative devices were suggested. Prominent among these were cryotrons, tunnel diodes, magnetic cores, magnetic films, and multiaperture magnetic devices.

In 1959, Goodyear Aerospace began using multiaperture magnetic devices in associative memories. Several problems existed in the application of magnetic devices to associative memories:

1. A non-destructive method for evaluating the storage state must exist. (When magnetic cores are used for storage, a chosen word was destructively read and rewritten. If one were to interrogate an entire core memory, as is necessary in an associative memory, all data must be read and rewritten simultaneously, which would be impractical.)
2. A low signal-to-noise level exists when a magnetic device is non-destructively interrogated (pulses of short duration must be amplified and distinguished from noise).
3. High energy is required to change the device state. This is true of all ferrite storage systems.
4. Switching times of the storage elements are relatively slow compared with other devices such as the cryotron.

To evaluate multiaperture magnetic devices for associative memories, transfluxors were purchased from RCA. Limitations of the transfluxor led to the development of a multiaperture logic element (MALE) (see Figure 2) and a model content addressable memory using the MALE.

The MALE(4) provided for storage of data in a word direction. A simultaneous exact match search of all stored words in memory could be realized. The MALE could be interrogated non-destructively and provided an EXCLUSIVE-OR operation. Initially, a response store was set for each word. The interrogation was made and reset the searched word that did not match the query word. The words that remained matched the query. Interrogation time of the MALE was about five microseconds. Limit searches in the MALE proceeded on a bit serial basis (five microseconds per bit). Greater than or less than search used the EXCLUSIVE-OR logic, at the stored bit level, to test the memory state for either greater than or less than the input argument.

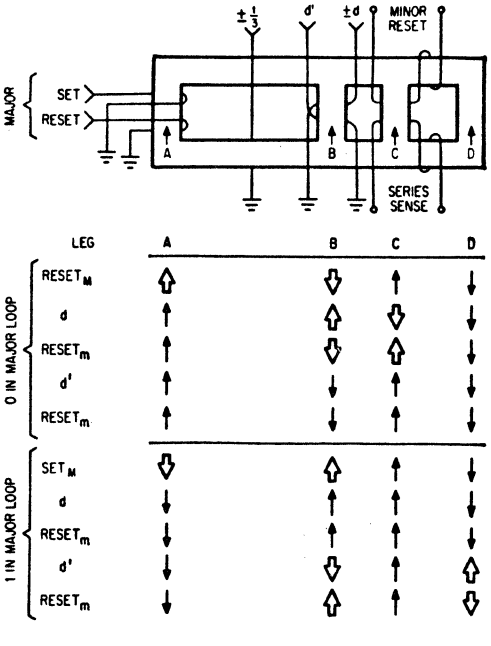


Fig. 2 - MALE Flux-State Diagram

Search Memory

The MALE was used to implement an associative memory for evaluation with the U. S. Navy's USQ-20. The search memory(5) (Figure 3) had 256 words with 30 bits per word. A block diagram of the search memory is shown in Figure 4. The machine instructions included write, erase, exact match search, greater than search, less than search and a number of optional instructions, no response required, response required, mask, no mask, count responders, etc. The memory was delivered in 1963.

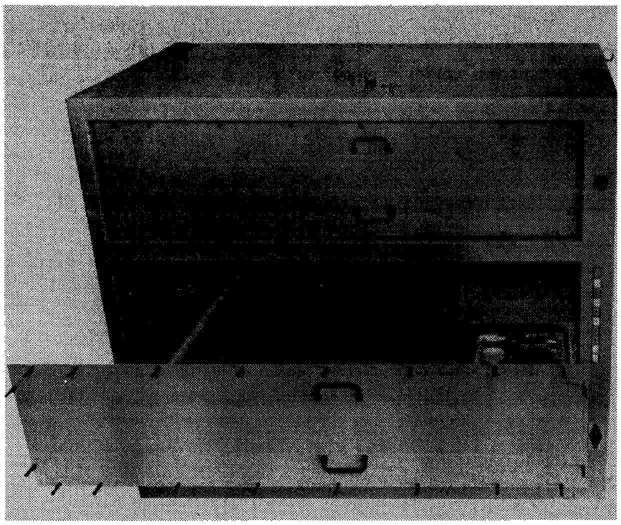


Fig. 3 - Search Memory

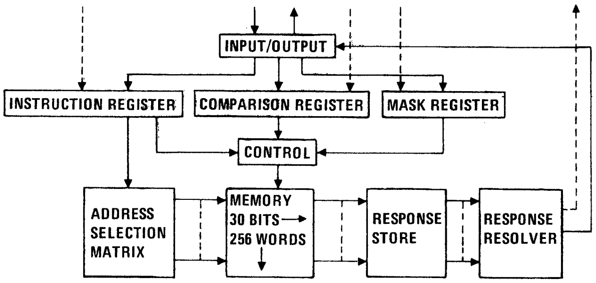


Fig. 4 - Search Memory Block Diagram

The MALE elements used in the NTDS search memory were difficult to fabricate; thus, a search was conducted for more readily available elements to implement the EXCLUSIVE-OR function. As a result, it was found that a conventional toroidal core could be interrogated without destroying its state.

BILOC

A toroid can be non-destructively interrogated using cross field switching techniques (6,7). However, the high cross field current, low signal level, and critical wire alignment mitigate against good performance. Apicella and Franks(8) discovered that applying a transverse bias field to the core reduces problems.

The static bias field results in:

1. A reduction of core switching time to about one-third of the unbiased switching time.
2. An order of magnitude increase in the cross field non-destructive output voltage.
3. The ability to achieve a logical EXCLUSIVE-OR function in the core.

Thus, a storage/logic element is produced that (1) can store a state, (2) be non-destructively interrogated, and (3) can provide a match or no-match comparison between the stored state and the interrogation. That is, the Boolean expression $AB + \bar{A}\bar{B}$ produces zero output, and the expression $\bar{A}B + A\bar{B}$ produces a one output. A is the query state, and B is the stored state.

This element, a biased logic core, was named BILOC. BILOC required very fast rise time pulses since the output voltage existed only during the pulse rise (or fall) time. Pulse rise times of the order of 20 nanoseconds and currents of about one ampere were used. The transverse bias field was of the order of 100 oersteds.

RADC Associative Memory

BILOC was used in implementing and delivering an associative memory in 1966 to the Rome Air Development Center.[a] The RADC associative memory had 2048 words of storage. Each word was 48 bits long. The associative memory was coupled via DMA to a CDC 1604B host computer. In operation, data to be searched was moved from the 1604B to the associative memory. The queries were then moved from the 1604B to the associative memory along with a response request. Results were transferred from the associative memory to the host 1604B.

A comprehensive set of instructions provided for conventional read/write of the memory and a set of logical interrogations, which included:

1. Input interrogand, equal, not equal, greater than, greater than or equal, less than, less than or equal, next higher value, and next lower value.
2. Find the maximum or minimum value.
3. Resolve instructions such as read first/next responder address or data, count responders, jump on no response (or its inverse).
4. Write next available location or write at given address.
5. The capability of concatenating searches to implement complex searches.

The RADC associative memory brought out several facts. Among these were:

1. The desirability of dropping the parallel search capability since only exact match searches could use this feature.
2. The desirability of processing selected entries in memory. Since transferring them to the host required time, the associative memory could operate at only 35 percent of its capability because of the necessity for input/output.
3. The desirability of a wide band I/O path.
4. The desirability for an internal program store to minimize I/O with the host machine.

Associative Processing

These facts led to a goal at GAC; namely, to achieve full parallel processing within the associative memory. That is, make it a true associative processor. The associative processor would accept unprocessed data at its input and produce processed results at the output, thus greatly reducing the input/output requirements and making greater use of the machine's capability.

However, the extremely high energy demands for the simultaneous write of 2048 cores needed to realize associative processing necessitated a search for a storage medium that was more easily alterable. The search led to plated wire. Plated wire offered the features of relatively low interrogate and write currents and was easily fabricated in our laboratories. Goodyear Aerospace conducted plated wire R&D from 1965 until 1969.

Plated Wire Associative Processor

In 1969, Goodyear Aerospace examined an air interceptor processing task[9] and demonstrated a plated wire associative processor[10,11,12,13]. The machine used a bit slice-oriented organization (Figure 5). The bit slices could be interrogated at a 100-nanosecond rate. Input was either to a bit slice or to any 16-bit word location in the array. Output from a selected word in the array was bit serial. In addition to conventional read and write operations, the array performed a large set of search, logic, and arithmetic operations at high speeds.

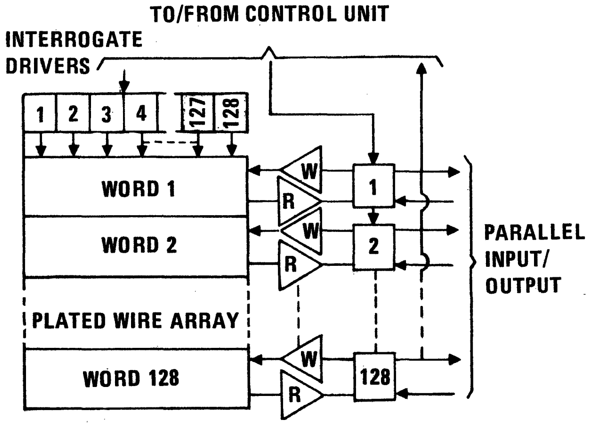


Fig. 5 - Bit Slice-Oriented Organization

Search operations are exact match with comparand, mismatch with comparand, greater than comparand, less than comparand, between limiting comparands, search flag, maximum value, and minimum value.

[a] Contract AF30(602)-3549.

Logic operations are set response toggles, reset response toggles, complement toggles, shift response toggles, write flag from response, and write common to selected words.

Arithmetic operations are add common argument, subtract common argument, add memory fields, subtract memory fields, multiply memory fields, divide memory fields, multiply by common argument, and divide by common argument.

The Knoxville Experiment. The plated wire associative processor, under contract to Univac, was programmed and installed at the Knoxville, Tennessee, air traffic control terminal. Several firsts were realized for this FAA installation. They were: automatic track initiation and update on beacon and primary radar reports, automatic turn detection, Mode C altitude tracking, air-to-air conflict prediction, conflict resolution, and automated voice advisory warning against other aircraft and terrain.

The plated wire associative processor had several drawbacks. Among these were the lack of production wire for the memory, the small signal output from the wire, and the requirement for bit serial readout of data from the array.

Early Integrated Circuit Efforts

About 1970, LSI content addressable memories (CAM) began to appear from companies that had integrated circuit capability. These CAM's offered advantages over plated wire such as low switching current, high speed, and parallel readout. But there were problems.

A study by Shore and Polkinghorn[14] considered a word-parallel, bit-parallel LSI associative processor. This cellular organization required about 130 transistors at each bit of storage and would provide parallel limit search and arithmetic operations (this is in contrast with the serial operations Goodyear Aerospace had been using). Later, Shore concluded in a paper[15] entitled "Second Thoughts on Parallel Processing" that parallel computers would be extremely expensive and could never compete with the conventional processor. His results seem correct when based on the cellular organization he had earlier studied. The general conclusions reached in Shore's "Second Thoughts" paper apparently assumed that all parallel processing hardware would require logic at the bit level.

If processing hardware in a parallel machine is implemented within storage at the bit level, then system cost increases nearly linearly with the amount of storage as Shore's paper indicates. Further difficulties ensue because of the necessity to access each stored bit in both the word and bit direction. Figure 6 shows an organization for a content addressable memory. The cost of implementation was quite high. For example, wiring a CAM using a typical CAM chip of 64 bits required 3b+2w+8c connections, where b is the number of bits, w is the number of words, and c is the number of chips needed. Then, making an array of 256 words by 256 bits would require 2304 leads. Goodyear Aerospace concluded that this would be an unsatisfactory approach. A search for a method to use conventional memory devices in a bit or word mode was realized in Batcher's invention of the MDA memory and flip network[16,17]. These inventions had their genesis in Batcher's work on sorting networks[18].

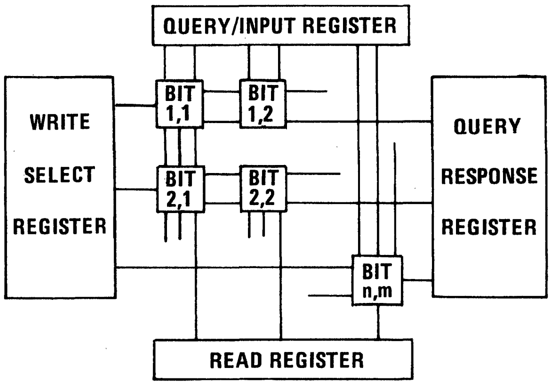


Fig. 6 - Content Addressable Memory Organization

Integrated Circuit Associative Processor

The MDA memory and flip network allowed conventional memory chips to be written in a word mode and read in a bit slice mode or vice versa. The inventions yielded an associative memory capability with only slightly greater amounts of hardware than a conventional memory. An associative processor was easily realized with a simple bit serial processing element (PE). One PE configuration is shown in Figure 7. The PE logic functions are given in Table 1.

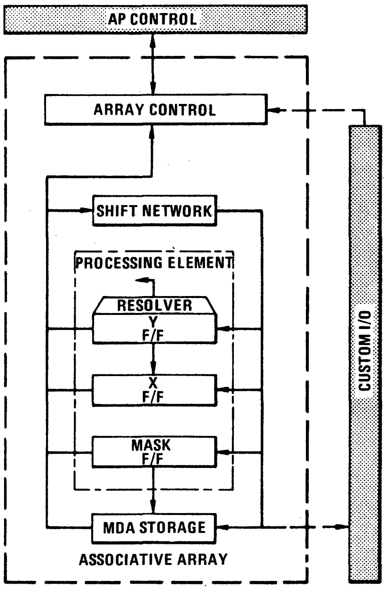


Fig. 7 - Associative Array Block Diagram

TABLE 1 - LOGIC FUNCTIONS

<table>
<tr><td></td><td colspan="4">COL</td><td>24</td><td>25</td><td>26</td><td>24</td><td>25</td><td>26</td><td>24</td><td>25</td><td>26</td><td>24</td><td>25</td><td>26</td><td>24</td><td>25</td><td>26</td></tr>
<tr><td>Common Register Bit = 1</td><td>16</td><td>17</td><td>18</td><td>19</td><td>K3X</td><td>K3Y</td><td>K4</td><td>K3X</td><td>K3Y</td><td>K4</td><td>K3X</td><td>K3Y</td><td>K4</td><td>K3X</td><td>K3Y</td><td>K4</td><td>K3X</td><td>K3Y</td><td>K4</td></tr>
<tr><td>Common Register Bit = 0</td><td>20</td><td>21</td><td>22</td><td>23</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td rowspan="2">Logic Function</td><td rowspan="2">NJ</td><td rowspan="2">NK</td><td rowspan="2">K1</td><td rowspan="2">K2</td><td colspan="3">New State of</td><td colspan="3">New State of</td><td colspan="3">New State of</td><td colspan="3">New State of</td><td colspan="3">New State of</td></tr>
<tr><td>X</td><td colspan="2">Y</td><td>X</td><td colspan="2">Y</td><td>X</td><td colspan="2">Y</td><td colspan="2">X</td><td>Y</td><td colspan="2">X</td><td>Y</td></tr>
<tr><td>Exclusive OR Complement F</td><td>0</td><td>0</td><td>0</td><td>0</td><td>X</td><td colspan="2">$Y\oplus\bar{F}$</td><td>$X\oplus\bar{F}$</td><td colspan="2">Y</td><td>$X\oplus\bar{F}$</td><td colspan="2">$Y\oplus\bar{F}$</td><td colspan="2">$X\oplus Y\bar{F}$</td><td>Y</td><td colspan="2">$X\oplus Y\bar{F}$</td><td>$Y\oplus\bar{F}$</td></tr>
<tr><td>Inclusive OR Complement F</td><td>0</td><td>0</td><td>0</td><td>1</td><td>X</td><td colspan="2">$Yv\bar{F}$</td><td>$Xv\bar{F}$</td><td colspan="2">Y</td><td>$Xv\bar{F}$</td><td colspan="2">$Yv\bar{F}$</td><td colspan="2">$XvY\bar{F}$</td><td>Y</td><td colspan="2">$XvY\bar{F}$</td><td>$Yv\bar{F}$</td></tr>
<tr><td>Logical AND</td><td>0</td><td>0</td><td>1</td><td>0</td><td>X</td><td colspan="2">YF</td><td>XF</td><td colspan="2">Y</td><td>XF</td><td colspan="2">YF</td><td colspan="2">$\bar{Y}XvXF$</td><td>Y</td><td colspan="2">$\bar{Y}XvXF$</td><td>YF</td></tr>
<tr><td>NO-Op</td><td>0</td><td>0</td><td>1</td><td>1</td><td>X</td><td colspan="2">Y</td><td>X</td><td colspan="2">Y</td><td>X</td><td colspan="2">Y</td><td colspan="2">X</td><td>Y</td><td colspan="2">X</td><td>Y</td></tr>
<tr><td>Load Complement F</td><td>0</td><td>1</td><td>0</td><td>0</td><td>X</td><td colspan="2">$\bar{F}$</td><td>$\bar{F}$</td><td colspan="2">Y</td><td>$\bar{F}$</td><td colspan="2">$\bar{F}$</td><td colspan="2">$\bar{Y}XvY\bar{F}$</td><td>Y</td><td colspan="2">$\bar{Y}XvY\bar{F}$</td><td>$\bar{F}$</td></tr>
<tr><td>NOT-Inclusive OR</td><td>0</td><td>1</td><td>0</td><td>1</td><td>X</td><td colspan="2">$\bar{Y}\bar{F}$</td><td>$\bar{X}\bar{F}$</td><td colspan="2">Y</td><td>$\bar{X}\bar{F}$</td><td colspan="2">$\bar{Y}\bar{F}$</td><td colspan="2">$\bar{Y}XvY\bar{X}\bar{F}$</td><td>Y</td><td colspan="2">$\bar{Y}XvY\bar{X}\bar{F}$</td><td>$\bar{Y}\bar{F}$</td></tr>
<tr><td>AND Complement F</td><td>0</td><td>1</td><td>1</td><td>0</td><td>X</td><td colspan="2">$Y\bar{F}$</td><td>$X\bar{F}$</td><td colspan="2">Y</td><td>$X\bar{F}$</td><td colspan="2">$Y\bar{F}$</td><td colspan="2">$\bar{Y}XvX\bar{F}$</td><td>Y</td><td colspan="2">$\bar{Y}XvX\bar{F}$</td><td>$Y\bar{F}$</td></tr>
<tr><td>Clear to Zero</td><td>0</td><td>1</td><td>1</td><td>1</td><td>X</td><td colspan="2">0</td><td>0</td><td colspan="2">Y</td><td>0</td><td colspan="2">0</td><td colspan="2">$\bar{Y}X$</td><td>Y</td><td colspan="2">$\bar{Y}X$</td><td>0</td></tr>
<tr><td>Input (F)</td><td>1</td><td>0</td><td>0</td><td>0</td><td>X</td><td colspan="2">F</td><td>F</td><td colspan="2">Y</td><td>F</td><td colspan="2">F</td><td colspan="2">$\bar{Y}XvYF$</td><td>Y</td><td colspan="2">$\bar{Y}XvYF$</td><td>F</td></tr>
<tr><td>Inclusive OR</td><td>1</td><td>0</td><td>0</td><td>1</td><td>X</td><td colspan="2">YvF</td><td>XvF</td><td colspan="2">Y</td><td>XvF</td><td colspan="2">YvF</td><td colspan="2">XvYF</td><td>Y</td><td colspan="2">XvYF</td><td>YvF</td></tr>
<tr><td>NOT AND Complement F</td><td>1</td><td>0</td><td>1</td><td>0</td><td>X</td><td colspan="2">$\bar{Y}vF$</td><td>$\bar{X}vF$</td><td colspan="2">Y</td><td>$\bar{X}vF$</td><td colspan="2">$\bar{Y}vF$</td><td colspan="2">$\bar{Y}XvYFvY\bar{X}$</td><td>Y</td><td colspan="2">$\bar{Y}XvYFvY\bar{X}$</td><td>$\bar{Y}vF$</td></tr>
<tr><td>SET to One</td><td>1</td><td>0</td><td>1</td><td>1</td><td>X</td><td colspan="2">1</td><td>1</td><td colspan="2">Y</td><td>1</td><td colspan="2">1</td><td colspan="2">XvY</td><td>Y</td><td colspan="2">XvY</td><td>1</td></tr>
<tr><td>Exclusive OR</td><td>1</td><td>1</td><td>0</td><td>0</td><td>X</td><td colspan="2">$Y\oplus F$</td><td>$X\oplus F$</td><td colspan="2">Y</td><td>$X\oplus F$</td><td colspan="2">$Y\oplus F$</td><td colspan="2">$X\oplus YF$</td><td>Y</td><td colspan="2">$X\oplus YF$</td><td>$Y\oplus F$</td></tr>
<tr><td>NOT Inclusive OR Complement F</td><td>1</td><td>1</td><td>0</td><td>1</td><td>X</td><td colspan="2">$\bar{Y}F$</td><td>$\bar{X}F$</td><td colspan="2">Y</td><td>$\bar{X}F$</td><td colspan="2">$\bar{Y}F$</td><td colspan="2">$\bar{Y}XvY\bar{X}F$</td><td>Y</td><td colspan="2">$\bar{Y}XvY\bar{X}F$</td><td>$\bar{Y}F$</td></tr>
<tr><td>NOT AND</td><td>1</td><td>1</td><td>1</td><td>0</td><td>X</td><td colspan="2">$\bar{Y}v\bar{F}$</td><td>$\bar{X}v\bar{F}$</td><td colspan="2">Y</td><td>$\bar{X}v\bar{F}$</td><td colspan="2">$\bar{Y}v\bar{F}$</td><td colspan="2">$\bar{Y}XvY\bar{X}vY\bar{F}$</td><td>Y</td><td colspan="2">$\bar{Y}XvY\bar{X}vY\bar{F}$</td><td>$\bar{Y}v\bar{F}$</td></tr>
<tr><td>NEGATE</td><td>1</td><td>1</td><td>1</td><td>1</td><td>X</td><td colspan="2">$\bar{Y}$</td><td>$\bar{X}$</td><td colspan="2">Y</td><td>$\bar{X}$</td><td colspan="2">$\bar{Y}$</td><td colspan="2">$Y\oplus X$</td><td>Y</td><td colspan="2">$Y\oplus X$</td><td>$\bar{Y}$</td></tr>
</table>

⊕ Exclusive OR	F :Bit from input network (Source determined by bits 29-31 of Associative Instruction Format)
v Inclusive OR	X :Old State of X - Response Store Register
- Complementation	Y :Old State of Y - Response Store Register

Design of a solid-state associative processor began in 1971. The first STARAN system[19] was completed in April 1972 and was demonstrated at the International Air Exposition "Transpo 72" at Dulles International Airport. A STARAN B system is shown in Figure 8.

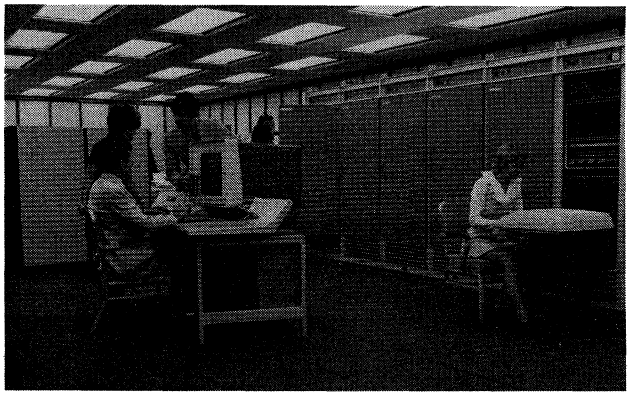


Fig. 8 - STARAN B System

The demonstration showed the capability of an associative processor to handle the air traffic control (ATC) processing requirement. Figure 9 shows the program flow in the STARAN S-500 programmed to operate with up to 500 aircraft tracks. In this system, digitized radar reports were received via data link from an ARSR radar site. This was supplemented by the generation of 250 simulated tracks based on 250 four-leg flight plans entered into the machine. The ATC operations performed are listed below.

1. Radar input processing
2. Primary 2D tracking
3. Secondary 2D tracking
4. Altitude tracking (Mode C)
5. Flight plan update
6. Target simulation
7. Maneuver detection
8. Conflict prediction
9. Conflict resolution
10. Automatic voice advisory
11. Keyboard processing
12. Full digital display processing

The system was set up and demonstrated with live radar in six locations in the United States and Canada. The demonstration could be speeded up, in simulation, by a factor of 30 times. This yielded effective performance as if:

- 7500 flight plans were updated per 10-second scan
- The new flight plan position generated 7500 radar reports, which were used to update 7500 tracks
- 30 displays were being driven by the system.

In 1972, another R&D effort - a "real" relational data base - was implemented in STARAN. The system used a parallel head disc and retrieved and entered data based on the content of the stored data[20].

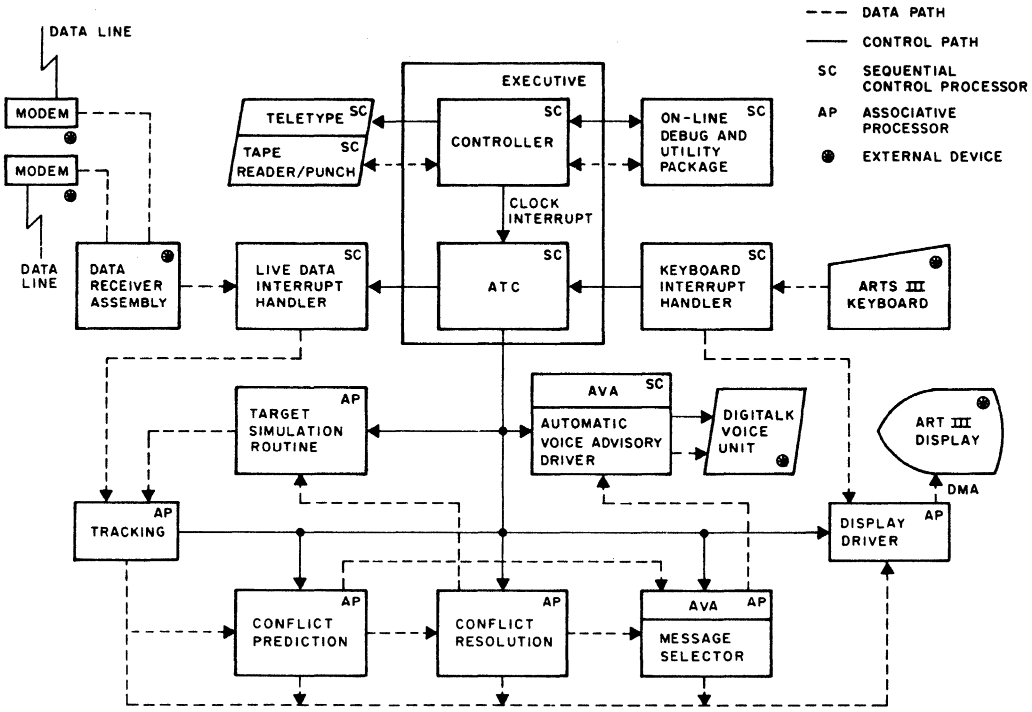


Fig. 9 - STARAN S-500 Program Flow

STARAN B installations were made at Rome Air Development Center, Defense Mapping Agency, Engineering Topographic Laboratories, Johnson Space Flight Center, and Goodyear Aerospace. Many applications were studied or were programmed. Some of them are for the space environment; catalog maintenance; detection and surveillance; weapons support; object identification; and sensor systems status[22,23]. A number of suggestions about STARAN B were incorporated in the STARAN E: (1) AWACS passive[24] and active[25] tracking, (2) data base management[26], and (3) image processing[27,28,29].

STARAN E

STARAN B was followed by STARAN E in 1975. STARAN E provided improvement over the earlier STARAN B in several areas (Table 2).

TABLE 2 - STARAN B/STARAN E COMPARISON

Item	STARAN B	STARAN E
Array page size	256x256	256x256
Max storage/array	1 page	64 pages
	0.008 Mbytes	0.5 Mbytes
Parallel I/O data rates	80 Mbytes/sec	80 Mbytes/sec
Cycle steal	No	Yes
Host interface	Slow	Fast
Proc-to-memory bandwidth	80 Mbytes/sec	215 Mbytes/sec
Processing rate (ops/sec)		
16-bit add	$11.5\text{x}10^6$	$15.4\text{x}10^6$
16-bit search	$48.0\text{x}10^6$	$60.6\text{x}10^6$

Microcomputer Array Processor (MAP)

Goodyear Aerospace's activities in electronic warfare led to a number of studies of parallel processing for the EW requirement. These efforts led to the MAP digital processing system designed for electronic warfare applications. This system is comprised of two major subsystems: a preprocessor that is a digital tracking device and a multiprocessor that is a programmable computer subsystem. The preprocessor compares each digitally encoded radar pulse intercepted by the receiver system against a file of emitters being tracked by the preprocessor. Limit searches of frequency, pulse width, and angle of arrival as well as PRI tracking are used in the association process between intercept and emitter. The current feasibility model of the preprocessor is a microprogrammable device that can process in excess of 300,000 intercepts per second from several hundred emitters in real time. Expansion to several million intercepts per second from a thousand emitters is possible.

The multiprocessor subsystem finds emitters among the intercepts that fail association in the preprocessor. This subsystem (Figure 10) consists of a number of independent processors that concurrently work on the emitter establishment problem. Each processor is a 32-bit programmable computer with its own dedicated memory and a capability to execute approximately four million instructions a second. In addition to the dedicated memory, each processor can communicate with numerous banks of global memory. The various global memory modules and their communication structure serve to tie the individual processors together in a symmetrical multiprocessor computer architecture. The multiprocessor system is modular and can contain as few as two and as many as eight processors coupled with from 1 to 16 banks of global memory. A 32-million instructions per second execution rate is achieved. Expansions beyond these limits are possible if every processor does not have to access every global memory module. A four-processor system (with three banks of global memory) was installed at Wright Patterson AFB in 1979 for use by the Air Force Avionics Laboratory. This system can execute approximately 16 million instructions per second and support a memory access rate of 20 million words per second.

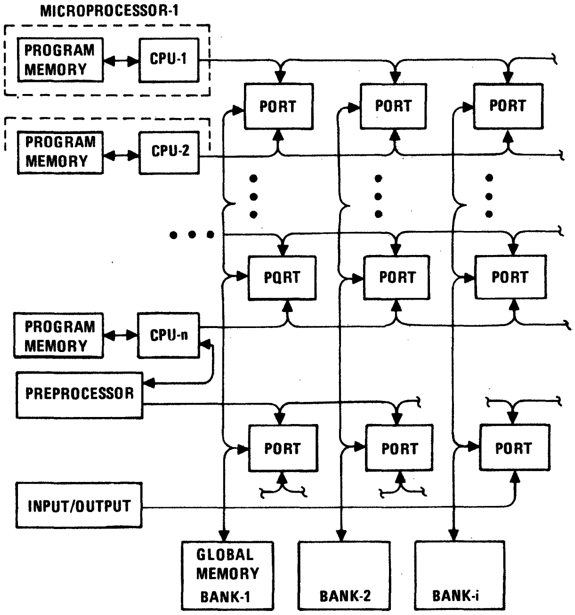


Fig. 10 - MAP Architecture

Airborne Surveillance

The capabilities of associative processing led to a study of its potential in the airborne surveillance environment. The study showed that the associative processor could augment a conventional airborne processor. Many of the inherently parallel functions such as report correlation, tracking, and display processing could be performed in the associative processor. Processing throughput could be increased by more than an order of magnitude.

A second study demonstrated the expected benefits. This was accomplished by interfacing a STARAN E to the host computer and by programming the machine to carry out many surveillance functions. A parallel effort was conducted to develop a processing element chip. The chip development was necessary to realize the associative processor capability within the very limited space (less than 0.5 cu ft) and power (less than 320 watts). The chip, using CMOS/SOS technology, was successfully fabricated by Rockwell International and demonstrated 11 months after the development was started.

Goodyear Aerospace is currently under contract from Grumman Aerospace to design and build a number of prototype ASPRO units.

ASPRO Organization

A block diagram of ASPRO[21] is shown in Figure 11. ASPRO is divided into five functional subsystems:

1. Control memory contains both program and buffer memory and is also connected to a host computer to allow for data, control word, and status transfer.
2. Program execution control is responsible for maintaining the correct program flow by executing program branches and returns as required and establishes correct timing and interlocking of the operation to be performed as defined by the instructions.
3. Data path contains the working registers and an arithmetic unit. All data to and from the control memory and/or the array is passed through this portion.
4. Array control identifies the array operation to be performed and supplies correctly synchronized control signals to the array.

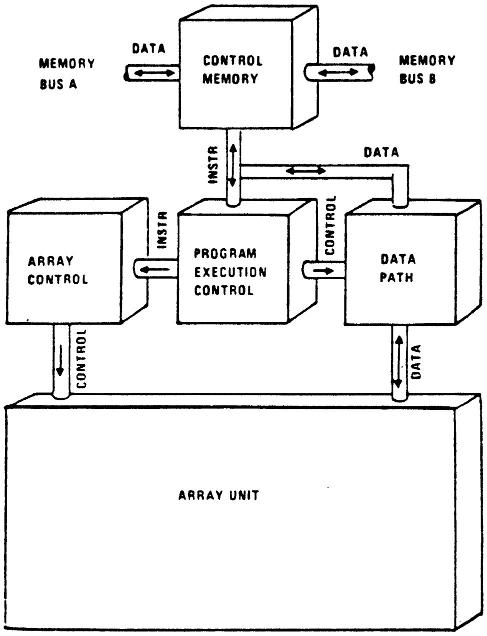


Fig. 11 - Block Diagram of ASPRO

5. Array unit is made up of 17 array modules. Sixteen modules of 128 words each make up the 2048-word array. The spare module may be switched in should one of the basic modules be found in error. Each module includes a 128-word by 4096-bit array of solid-state multidimensional access (MDA) storage and 128 processing elements (PE's).

The array consists of four basic components: array memory, flip or permutation network, processing elements, and resolver. Access can be made in either the bit or word direction, depending on the mode bit of the instruction.

Massively Parallel Processor (MPP)

In December 1979, NASA Goddard awarded a contract to Goodyear Aerospace to construct a massively parallel processor (MPP) to be delivered in the fourth quarter of 1982. The MPP was developed for image processing satellite data. The expected workload is between 10^9 and 10^{10} operations per second.

The major components of MPP are shown in Figure 12. The array unit (ARU) processes arrays of data at high speed and is controlled by the array control unit (ACU), which also performs scalar arithmetic. The program and data management unit (PDMU) controls the overall flow of data and programs through the system and handles certain ancillary tasks such as program development and diagnostics. Three staging memories buffer and reorder data between the ARU, PDMU, and external (host) computer.

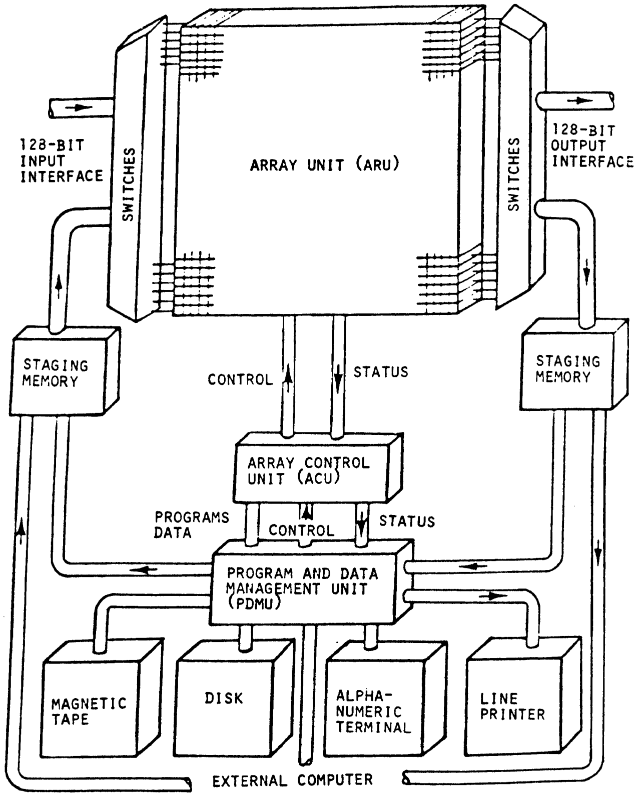


Fig. 12 - MPP Block Diagram

Array Unit

Logically, the array unit (ARU) contains 16,384 processing elements (PE's) organized as a 128 by 128 square. Physically, the ARU has an extra 128 by 4 rectangle of PE's that is used to reconfigure the ARU when a PE fault is detected. The PE's are bit-serial processors for efficiently processing operands of any length. The ARU has a very high processing speed (Table 3). The bandwidth between PE's and memory is 1.6×10^{11} bits per second.

A study showed the desirability of making edge-connectivity a programmable function. The top bottom and right-left edges can either be connected or left open. A spiral mode connects the 16,384 PE's together in one long linear array.

I/O for the array is up to 160 Mbytes per second and can be transferred through the ARU I/O ports. Processing is interrupted for 100 nanoseconds for each bit plane of 16,384 bits transferred - less than one percent of the time. The 96 boards of the ARU are packaged in one cabinet. Forced-air cooling is used.

Array Control Unit

Like the control units of other parallel processors, the array control unit (ACU) performs

TABLE 3 - 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	430
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	216
Multiplication of Array by Scalar	
8-bit integers (16-bit product)	2340
12-bit integers (24-bit product)	1260
32-bit floating-point numbers	373

*Million operations per second

scalar arithmetic and controls the PE's. It has three sections that operate in parallel: PE control, I/O control, and main control. PE controls performs all array arithmetic of the application program. I/O control manages the flow of data in and out of the ARU. Main control performs all scalar arithmetic of the application program. This arrangement allows array arithmetic, scalar arithmetic, and input/output to be overlapped for minimum execution time.

Program and Data Management Unit

The program and data management unit (PDMU) controls the overall flow of programs and data in the system (Figure 12). Control is from an alphanumeric terminal. The PDMU is a minicomputer (DEC PDP-11) with custom interfaces to the ACU memories and registers and to the staging memories. The operating system is DEC's RSX-11M real-time multiprogramming system.

The PDMU also executes the MPP program-development software package. The package includes a PE control assembler to develop array processing routines for PE control, a main assembler to develop application programs executing in main control, a linker to form load modules for the ACU, and a control and debug module that loads programs into the ACU, controls their execution, and facilitates debugging. This package is written in Fortran for easy movement to the host computer.

Staging Memories

The staging memories reside between the wide I/O ports of the ARU and the PDMU. They also have a port to an external (host) computer. Besides acting as buffers for ARU data being input and output, the memories reorder arrays of data.

Arrays of data are transferred through the ARU ports in bit-sequential order. That is, the most (or least) significant bit of 16,384 elements followed by the next bit of 16,384 elements, followed by the next bit of 16,384 elements, etc. Reordering is required to fit the normal order of satellite imagery in the PDMU or the host. Thus the staging memories are given a reordering capability.

The large multidimensional access staging memory uses 1280 dynamic RAM circuits for data storage and 384 RAM's for error-correcting-code (ECC) storage (a 6-bit ECC is added to each 20-bit word). Initially, the boards will be populated with 16K bit RAM's for a capacity of 2.5 Mbytes. The memory can be programmed to input and output imagery in a wide variety of formats.

The Future

Current efforts in ASPRO, MPP, and MAP will yield improved processing systems in those areas where parallelism can be effectively applied. The breadth of application seems quite wide. A number of users have effectively converted "clearly sequential processes" into parallel algorithms for parallel solution.

We see smaller, more powerful parallel processors occupying less space and using less power being developed in the near future. We see the parallel processing technology as a most cost effective tool for real-time command and control, and other data base management tasks.

We haven't satisfied all the thoughts posed by Dr. Bush, but a first step is readily implemented in today's parallel processors. That step is the virtual elimination of the elaborate indexing structure required in today's processing systems. We've reduced "our ineptitude in getting at the record . . . largely caused by the artificiality of systems of indexing."

References

(1) Bush, Vannevar: Science - The Endless Frontier: A Report to the President, Government Printing Office, 1945, Office of Scientific Research and Development.

(2) A. E. Slade and H. O. McMahon, "A Cryotron Catalog Memory System," Proc. EJCC, Volume 10, pp 115-120, December 1956.

(3) Allen Kent, "A Machine That Does Research," Harpers Magazine, April 1959, Vol 218, No. 1307, pp 67-71.

(4) G. T. Tuttle, "How to Quiz a Whole Memory At Once," Electronics, Vol 36, pp 43-46, Nov. 15, 1963.

(5) Russell G. Gall, "A Hardware-Integrated GPC/Search Memory, Proc. FJCC, 1964.

(6) Tellman, R. M., IRE Transactions on Electronic Computers, Vol EC9, page 323 (1960).

(7) Winter, H. M., 1964 Proceedings of the Intermagnetics Conference, page 8-2-1.

(8) Apicella, A. and Franks, J., "BILOC-A High Speed NDRO One Core Per Bit Associative Element," Proceedings of Intermagnetics Conference, 1965.

(9) Costanzo, A., Garrett, J., "Application of an Associative Processor to an Interceptor Radar System, Proceedings of NAECON, 1969.

(10) W. C. Meilander, M. Bialer, and J. Garrett, "A Mission Oriented Associative Processor Using Plated Wire." Chapter 7 of Parallel Processor Systems, Technologies and Applications, SPARTAN Books, 1970; p 153.

(11) Fulmer, L. C. and Meilander, W. C., "A Modular Plated Wire Associative Processor," IEEE Computer Group Annual Conference, 1970.

(12) Rudolph, J. A., Fulmer, L. C., Meilander, W. C., "With Associative Memory, Speed Limit is No Barrier," Electronics, June 22, 1970.

(13) Rudolph, J. A., Fulmer, L. C., Meilander, W. C., "The Coming of Age of the Associative Processor," Electronics, Feb. 15, 1971.

(14) Shore, J. E., and Polkinghorn, F. A., "A Fast, Flexible, Highly Parallel Associative Processor," Report 6961. Naval Research Lab., Washington, D. C., Nov. 28, 1969.

(15) Shore, J. E., "Second Thoughts on Parallel Processing," Proc. 1972, IEEE, Intercon, pp 358-359.

(16) Pat. No. 3,800,289 dated Mar. 26, 1974, Multi-Dimensional Access Solid State Memory, Kenneth E. Batcher.

(17) Pat. No. 3,812,467, May 21, 1974, Permutation Network, Kenneth E. Batcher.

(18) Batcher, K. E., "Sorting Networks and Their Applications," Proceedings, The Spring Joint Computer Conference, 1968.

(19) Batcher, K. E., "Flexible Parallel Processing and STARAN," Wescon, Sept. 1972.

(20) Moulder, Richard, "An Implementation of a Data Management System on an Associative Processor," Proceedings, National Computer Conference, 1973.

(21) Smit, J., "Architecture Descriptions for the Massively Parallel Processor (MPP) and the Airborne Associative Processor (ASPRO) Very High Speed Computing Symposium."

(22) P. A. Gamelin, STARAN as a Space Computation Center (SCC) Processing Alternative. MITRE Report MTR 2836, June 1974.

(23) D. L. Baldauf, "Experiences with an Operational Associative Processor". MITRE Report MTR 2879, June 1974.

(24) Prentice, Brian W., "Implementation of the AWACS Passive Tracking Algorithms on a Goodyear STARAN", International Parallel Processing Conference, 1974, pp 250-270.

(25) Summers, Lt. Michael W., An Assocative Processor Application Study, RADC report RADC-TR-75-318, January 1976.

(26) Creswell, C. T.; Peters, Carol; Young, Brian R., "SACWARDANS ASSOCIATIVE PROCESSOR STUDY, PRC Information Sciences Co., RADC Report RADC-TR-74-341, January 1975.

(27) Rohrbacher, Donald, and Potter, J. L., Image Processing with the STARAN Parallel Computer, August 1977, pp 54-59.

(28) Gambino, Lawrence A., and Boulis, Roger L., "STARAN Complex" Defense Mapping Agency, USARMY Engineer Topographic Laboratories". Proc. 1975 Sagamore Computer Conference on Parallel Processing.

(29) Sherwin Ruben, John Lyon, Rudolf Faiss and Mathew Quinn Application of Parallel Processings of the International Conference on Parallel Processing, 1976.

CENTRALLY-CONTROLLED PARALLEL PROCESSORS*

D.L. Slotnick
Computer Science Department
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

Abstract

The circumstances surrounding the conception and development of centrally-controlled array processors are described. The period of time involved is from 1953 to 1975. It brackets the Westinghouse SOLOMON systems, their precursors and the University of Illinois ILLIAC IV. Some reflections on past and possible future interplay between university and government laboratories on the one hand and industry on the other are made at the conclusion.

The First Stirrings

In June 1952, with a new bride and new Masters degree in Mathematics, I took a job as a programmer with the Electronic Computer Project at The Institute for Advanced Study in Princeton. I had no idea what a programmer was expected to do but a school friend, Adolph Nussbaum, who was already working there and had arranged my apparently successful interview with Herman (H.H.) Goldstine, assured me that it was interesting and honest work and I did need a job. I left Princeton in February 1954 to return to school for a Ph.D. and though my stay at the Institute had been for only twenty months it played a significant role in my development. First, it was my initial contact with what became and remains my profession and it was the place where my vision broadened from the myopia of a young, partially cooked mathematician to encompass my still-enduring interests in physical science and technology. In particular, I had the good fortune to learn the rudiments of logic and hardware design from members of one of the most capable engineering staffs ever assembled. I remember, particularly, Leon Harmon, Hugh (Hewitt D.) Crane and Julian Bigelow as tutors. It was also at Princeton that I first thought of building a parallel processor. The idea was stimulated by the physical appearance of the magnetic drum that was being built to augment the 1,024 word primary memory of the IAS machine. The disposition of heads and amplifiers over the drum's 80 tracks (2 banks of 1,024,40-bit words) suggested to me the notion of, first, inverting the bit/word relationship so that each track stored the successive bits of a single word (in fact, of several words) and, second, associating a ten tube serial adder with each track so that in a single drum revolution an operation could be executed on the contents of the entire drum. The idea was to do, in parallel, an iterative step in a mesh calculation. I remember, probably under the influence of the 1,024 word Williams tube memory, desiring to build a 1,024 track drum to represent in this fashion a 32 by 32 mesh. I even had the temerity to make it the subject of the only conversation I had with von Neumann that didn't concern itself with one of the mundane programming chores that I occasionally did for him. He considered the idea for perhaps half a minute when he said that he thought it would require too many tubes. It was not a devestating personal setback because it had been the work of only a few evenings and some casual conversation with Harmon and Crane. I essentially considered final judgement to have been rendered and didn't seriously take up the idea again for roughly five years. During those five years I completed a Ph.D. degree (in applied math at what is now called The Courant Institute in New York) spent a year on a post-doc at Princeton (the University this time) and succumbing to the lure of action and regular meals took a job at IBM in Poughkeepsie.

I don't recall the immediate stimulus to taking up the idea again but know that it was early in 1958 as a member of the Computer Organization Department in the Research Labs. This Department, under W.J. Lawless and M. Clayton Andrews, was an environment that invited, in fact demanded, far ranging frontier exploration. The technological scene had by this time totally turned over; from receiving tubes and CRT stores to transistors and cores. My serial adder now became a small circuit board and the PE memory a core plane. I did some designs working on my own and began to think more generally about how one would need to modify algorithms to utilize parallelism. These thoughts and a few corridor conversations with my friend and colleague John Cocke resulted in my writing up [Cocke 1958]. This report describes, in some detail, including a derivation of the $O(\log_2 n)$ speed up where n, the degree of the polynomial, is assumed to be less than the number of processors. It also suggests parallel algorithmic approaches to the solution of ordinary and partial differential equations. My efforts in parallel computation never amounted to much at IBM and it was partly, but by no means exclusively, for this reason that I left IBM in June 1960 for what was to be a strange but important interlude lasting only 8 months at the newly formed Baltimore Aerospace Division of an old airplane manufacturer, Aeronca Manufacturing Corporation. What lured me to Aeronca was the promise that I could pursue my own ideas on parallel computers, the well-chronicled IBM frustration syndrome and a large raise in pay.

It would be wrong to say that IBM played no role in the development of my ideas but the role was largely indirect. It was at IBM, while working for Rex Rice on the design of a small

(a) *To appear in Annals of the History of Computing.

general-purpose computer, that I really learned the trade. My debt to Rex is great, though when I talked to him about parallel computers, although he listened, it was not always overly patiently. He even witnessed some of my drawings but Rex, then as always, to the great good fortune of the computer field, had his own fish to fry- and I got paid to help him fry his fish, not conversely. Rex will appear again, in an important way, much later in this story. Of the IBMers, Lawless alone showed some interest in my ideas and was the first, in 1959, to alert me to the work of Konrad Zuse, which to my only mild embarassment I confess I have still never looked at, who described a drum-based equation solver (perhaps similar to the machine I had thought about in Princeton) which is described in [Zuse 1958]. Regretably, I am here engaging in the dubious scholarly practice of quoting a non-primary source, for the only work I have even seen that refers to this Zuse paper (it gives a 1 line description) is [Kuck 1978] and my attempts to get the reference, it is in German, from the 4 libraries in the U.S. which allegedly possess it or from a colleague who is "looking for his copy" have thus far been unsuccessful.

Now for Aeronca and the almost-real world. Two people stand out from this interlude. One is Dr. Gordon J.F. MacDonald, who as a visiting scientist at the Goddard Institute for Space Studies, then in Silver Spring, Md., gave me both my first real encouragement and my first research contract; both were important. Moreover, Gordon understood critically and in precise detail what I was talking about. His support was thus particularly meaningful. With the help of this contract I did the first, fairly complete overall design and detailed an enriched Processing Element (PE) (I started calling them "elements" because large numbers of "elements" seemed somehow easier to take than large numbers of "units") to gate level. As a consequence of this contract, unforeseen to me, the government acquired a permanent license which permitted me to continue the work, under government sponsorship, after I left Aeronca.

The second of the two people is Chuck (Charles N.) Valenti. Chuck was the salesman who was given the responsibility of trying to help this innocent Ph.D. find an agency that would recognize some sort of stake in my research. Chuck had, particularly for a salesman, a rare combination of attributes. He was a true believer in the capitalist system, a patriot, very smart and essentially honest. He believed, and convinced me, that if you couldn't sell it then it probably wasn't worth all that much. He also knew the DOD like the back of his hand (it was Chuck who first revealed to me, for example, that every fourth door in the Pentagon was the entrance to a men's room). Trailing Chuck around and giving my pitch while he opened doors, watched, criticised and schemed gave me my first glimpse of how things worked in the complement of IBM, which I had previously considered barely non-empty.

The reason I left Aeronca still strikes me as amusing. The Technical Director, to whom I reported, had singularly eclectic interests comprehending the importing of spaghetti making machines and marble, a housing development corporation, ESP and a process developed by a retired colonel in Pennsylvania for ridding crops of all manner of blights by simply placing, for example, a leaf from a plant in the troubled field on top of a very special box of the venerable colonel's design. I was assigned the task of investigating this phenomenon for possible commercial exploitation. I examined the circuit schematic and found that, among its many unusual features, it seemed to function without a power supply and no detectable closed circuit. I reported that I thought the whole business preposterous nonsense and took the opportunity to also express my dim view of ESP as a means of secure battlefield communication. Although I wasn't fired on the spot the strain in the situation grew worse rapidly and I soon left, having arranged for a job with the nearby Air Arm Division of Westinghouse at Friendship Airport, Maryland (all the names of nearly everything have long since changed). It was at Westinghouse that things finally began to take off and I will discuss my four years there in some detail.

SOLOMON

I was hired by the Engineering Manager, Harry B. Smith, a first-rate radar man and an excellent manager. We agreed that I would be based in an existing computer development group that had some good people in it and some substantial accomplishments to its credit- primarily in the area of airborne analog computers. I was, from the beginning, however, given the freedom and wherewithal to pursue my own ideas. I followed up some of the contacts Valenti and I had initiated and quickly secured support from the Rome Air Development Center (RADC) and the United States Army Signal Corps Laboratory at Fort Monmouth, New Jersey. The principals I dealt with were Al (Alan A.) Barnum, Morris (A.) Knapp, and Bill (William) Moore at Rome and Dave (David) Haratz, Milt (M.A.) Lipton and Dr. Ed (Edward) Reilley at Fort Monmouth. Within 3 or 4 months from the time I joined Westinghouse I had started a small group with 2 young engineers who were the first to work with me on developing the SOLOMON design. They were Carl (W.C.) Borck and Bob (Robert C.) McReynolds. Carl, Bob and I then spent a most productive and gratifying year working out the design and some programming details which we reported first in a Workshop on Computer Organization [Slotnick 1963] held at Westinghouse, under RADC and Westinghouse Sponsorship in October 1962. Carl, Bob and I later presented a more detailed design article [Slotnick 1962] at the 1962 Fall Joint Computer Conference which to our surprise and pleasure won the first AFIPS Prize and which, together with a companion report on applications [Ball 1962], became the standard citations for SOLOMON. A word about the name SOLOMON before discussing the design; it was suggested to me because of both the (wise as) King Solomon connotation and his 1000 wives (servants, in a ruder and simpler era). SOLOMON was designed to have 1000 (OCTAL) Processing Elements (PE's). I had no acronym in mind. Much later the tortured Simultaneous Operation Linked Ordinal MOdular Network was devised by a creative salesman, Jerry McKindles with, I must shamefacedly confess, my help. The final design of SOLOMON (later, as we shall see, to be called SOLOMON I) was reported in [Gregory 1963]. By then I had a group of 12 engineers, under John (J.G.) Gregory, who so creatively and energetically supervised the later Westinghouse design and development work. Some of the others who figured prominently were Bill (W.W.) Beydler, Art (A.B.) Carroll, Marv (M.G.) Graham, Ed (E.R.) Higgins, Jim (J.R.) Hudson, Bill (W.H.) Leonard, George Shapiro, Dave (D.K.) Sloper and Bob (R.M.) Trepp.

In discussing the design I will utilize figures from both [Slotnick 1962] and [Gregory 1963]. The main ideas remained the same throughout the SOLOMON and, in fact, the ILLIAC IV program. These were of a PE array controlled from a central source, as shown in Figure 1. The program store was associated with the central control while operand storage was in the array as shown in the Processing Element block diagram of Figure 2. Each PE possessed a memory composed of two core frames. The two-address system employed, used a frame for each operand and wrote the result over one of them. Operands could also be broadcast from the central memory or come from (and results go to) the 4 nearest neighbors of any PE. This was essentially the drum design of the Institute days with the drum tracks replaced by the core frames. The connectivity was the same but I added the ability to wrap around the extreme rows and/or columns under program control. Also new were the Geometric Control Registers, shown in Figure 3, which permitted the selection of rows, columns or row/column intersections by number. A special buffer, the L-buffer did matching between the conventionally organized word-oriented processor of the central control unit and the serial-by-bit PE's. That is, words were handled in parallel (serial by word, or block of words) between L-buffer and central control but serial by bit (all the kth bits of a column of PE words) between L-Buffer and array. The core frames were assembled into stacks, as shown in Figure 4, which shared address decoding and drivers. Thus, the same address was selected for all the frames in a stack. In ILLIAC IV the technology was by then, as we'll see, able to permit independent PE memory addresses.

What Carl Borck named Mode Logic was to remain the main "local" (applicable to a PE, as opposed to "global"-applicable to the whole array via central control- terms I borrowed from mathematics because they fit so well) logical capability in all subsequent array computers. Each PE had a 2 bit register in which a local data-dependent "mode value" could be set. Each instruction had a corresponding 4 bit field specifying any subset of the 4 possible mode states which were "allowed" to participate in that instruction. That is, only PE's in one of the mode states specified in the instruction could change states during that instruction cycle. PE's in an "off" mode state could still provide operands to a neighbor if called upon to do so, and thus could influence the state of an "on" PE but an off PE could not itself change state. This very simple, almost irreducible residue of local control usually functions adequately as the principle array conditional branch. Word length was variable under the control of a setable register. This was easy in a bit serial system and necessary during this era of expensive memory.

The PE's were grouped into subarrays of 32x8. A system could contain up to 8 subarrays (2,048 PE's). We did many design studies with independent subarray controllers as we did with numerous redundant (for reliability) configurations of PE rows, columns and subarrays. The optimistic 4 quadrant ILLIAC IV design resulted from these efforts.

We never built a full-scale SOLOMON but did build significant experimental models; a 3 by 3 model with complete PE's, a 10 by 10 model with a somewhat different PE and numerous special breadboards to measure electrical characteristics in order to optimize signal distribution within the array and between the array, the central control and I/O units.

During 1962 and 1963 the PE evolved, with continued integrated circuit evolution, from the simple bit-serial element to a 24 bit element with 24 bit registers and byte-oriented PE arithmetic hardware. We built experimental PE's which added 24 bit numbers in 3.4 µsecs and multiplied them in less than 20 µsecs (using 10 mh circuits). Even programmed floating point times became reasonably respectable. The basic PE system module became smaller (256 to 32 PE's) as the PE itself became larger (1 to 24 bits). In ILLIAC IV this was to become modules of 64, 64 bit floating point PE's. For reasons soon to be clear the details of these later SOLOMONS (by now called by us at Westinghouse, SOLOMON II) were never published. They can be found, however, in [Westinghouse 1964a] and [Westinghouse 1964b].

Problem analysis and programming tried breathtakingly to keep pace over this thirty months or so of evolution of designs and models and simulations but except for some particular problems in partial differential equations, matrix inversion and a significant group in signal processing, never really managed it. We convinced ourselves early that the problem space for which this computer organization excelled was large enough to justify development of the full-scale machine, and, indeed, it nearly did. In programming we designed some new language constructs to overlay on a standard higher-level language, such as DO TOGETHER, but these likewise were to remain incompletely implemented.

At the beginning of 1964 RADC was pushing hard and with apparent success for a DOD program to build a full-scale SOLOMON but in March the situation turned totally sour. Our principal DOD Washington sponsor drowned in a tragic accident and the program's chief opponent, an old-line affiliate of an industrial competitor who was soon to leave DOD under something of a cloud, acted quickly to kill our chances. By this time I had assembled a group of 100 or so mouths to feed, about half of whom were working on SOLOMON (I had taken on additional responsibilities) and had to shift gears quickly to avert a total wipe-out.

Thus began the visits to the Atomic Energy Commission's Lawrence Radiation Labs at Livermore, California and Dr. Sid (Sidney) Fernbach. Sid labored mightily to stave off a disaster at Westinghouse. He wanted to see the technique tried on a decent scale but failing to squeeze the development cash out of Washington, he managed only a contract offer to lease a system were Westinghouse to develop it on its own funds and I couldn't argue Westinghouse into accepting it. The 6 months between the shut-off of most of our DOD funds and the AEC's unacceptable contract offer remain a frenzied blur with Sid's lasting encouragement and friendship as its only redeeming feature.

When the Group Vice-President at Westinghouse turned the contract down I, of course, instantly submitted my resignation which he turned aside in a speech so full of understanding and concern that I managed not to kill him. The Westinghouse Baltimore executives acting with their typical very considerable skill had me dismantle my group and then offered me the job of manager of Advanced Development, a group of 1,000 engineers plus a big support operation, which included the offer of a home for most of the people in my SOLOMON group. I accepted it as a means to resettle my group. I also tried for several months after the debacle to start my own company and raise the venture capital to build a machine for Sid but I couldn't quite pull it off although I had pledges for about three quarters of the money when I took another turn.

I had gotten to know John (R.) Pasta, head of the Computer Lab at the University of Illinois, when he invited me to join an informal group that met periodically to discuss directions in the computer field. He knew what was happening to me and encouraged me in many ways to reflect on the abiding nature of intellectual achievement as opposed to the transient goals I was becoming obsessed with. The academic life looked good, indeed, and after checking the other academic possibilities I decided to join John at Illinois because of their outstanding reputation in having not only used but built machines and because of my friendship and respect for John. In May of 1965 I moved my family to Urbana.

ILLIAC IV

The SOLOMON work had essentially come to an end at Westinghouse a year before I left for Illinois and that year was spent frantically and unsuccessfully looking for ways to get it going again while simultaneously taking on a tough new job assignment. When I arrived in Illinois I was anxious to do something else, in fact, anything else. Ivan Sutherland, whom I had visited at ARPA, where he headed the Information Processing Techniques Group, called and asked to visit me about a month or so after I came to Illinois. Without really expecting anything to come out of it other than a pleasant visit with a very bright and genial colleague, I of course happily set it up. As it turned out, what was on his mind was to see if I would be interested in developing a big parallel computer at Illinois. I thought about it, talked with John Pasta who was more than agreeable and, not without some "here we go again" trepidation, I told Ivan yes.

Ivan wanted to start the project with a small study phase but I absolutely refused. I wanted a two million dollar payment at the outset and a contract for a total of ten million. I did this to make sure that the ARPA committment was real and had passed the highest levels of review. Ivan agreed. I wrote the proposal and a few weeks later we had our contract. John and I decided with much regret that the days when a university could design and fabricate a big machine, by itself, were over and we decided that while we would do the architectural design and most of the software and applications work, we would rely on industry for detail design and fabrication.

For nearly a year I gathered a nuclear staff and worked to develop design specifications for a study phase procurement to be followed by the fabrication phase. I outlined the major approaches in [Slotnick 1966] and we incorporated them in a bid set. In August 1966, after many months of intensive contacts with industry, three 8-month contracts were awarded to RCA, Burroughs and UNIVAC. In April 1967 Burroughs was selected to go on to do the final design and development. The selection of Burroughs, while not quite by default, was hardly hotly-contested. Burroughs had teamed with Texas Instruments, who were to develop the integrated circuits for the PE even though they were, at the same time, building their own high-performance pipelined system. They (TI) had, in fact, tried to interest us in abondoning the parallel approach in favor of a pipeline. Control Data also had a whack at this. But it seemed pointless, from any point of view and, in fact impossible from mine, to think of developing three pipeline processors and no parallel processors.

The technical details of ILLIAC IV are quite well known; the standard citation is [Barnes 1968]. I will thus concentrate here on those surrounding circumstances and issues which most influenced the program.

My original intention was to build a system consisting of four subarrays (quadrants) of 64 PE's each; a PE now being a 12,000 gate floating point (48 bit mantissa-16 bit exponent) processor. The 4 quadrants could each operate independently; the system thus acting as a 4 unit multiprocessor. The system could also operate as 2-2quadrant units or as a single 4 quadrant (256 PE) unit. Each PE, moreover, could function as a single 64 bit floating point element as 2 - 32 bit floating point elements (24 bit mantissa- 8 bit exponent) or as 8 - 8 bit character-oriented fixed point elements; this last operating mode being directed toward signal and image processing applications. As we will see, by 1969 cost overruns made it necessary to reduce the size of the system to the single quadrant, described in [Slotnick 1971], that was built. (It is a quaint observation that the space still exist in the ILLIAC IV back panels to plug in the connectors to the missing three quadrants) The PE, however, underwent no significant change in logical organization.

There were two main contributions to the early overruns and consequent retrenchment. The first was the inability of Texas Instruments to produce the 64 pin ECL packages around which we had designed the PE. A great deal of inconclusive mutual finger pointing went on between TI and their suppliers but the upshot was the loss of somewhat more than a year of time and, all related and consequent expenses considered, perhaps 4 million dollars. The second severe setback was the inability of Burroughs to produce the magnetic thin film PE memories. In time and dollars this amounted to the loss of roughly an additional 2 million dollars and a year further delay.

We retreated from the 64 pin packages to standard 16 pin dual in line packages. In so doing, however, everything got bigger and more expensive. A lot of the logic design was salvageable as a consequence of making the 16 pin packages logically derivative of the abondoned 64 pin packages but board layout, back-panel wiring and all system-level hardware had, of course, to restart from scratch. The memory situation was even messier. Burroughs had a large investment in their thin films and didn't want to give up on them even after my own and independent review had concluded that they still represented an intolerable development risk. Semiconductor manufacturers were just beginning to gear up for memory chip manufacture; the manufacturing means were clearly at hand, or at least so it seemed to me, but the chips were not. I made the painful decision to drop films and go with semiconductors. After making the rounds of all the possible qualified suppliers I recommended the selection of Fairchild Semiconductors, whose memory systems operation was then headed by Rex Rice, under whom in a former life I had, as I have already discussed, learned the tricks of the trade. This selection met with considerable opposition from Burroughs and others. Such was the furor in fact, that ARPA, convened a panel of independent experts who carefully reviewed the situation and sustained my decision. It turned out that when Fairchild delivered their memory it was still the only high-speed semiconductor memory being delivered and that ILLIAC IV was the first large system to employ a semiconductor primary store.

By this time Ivan Sutherland had sought other if not greener pastures and the combination of Larry (Lawrence G.) Roberts and Bob (Robert) Taylor had replaced him in the ARPA computer operation. To Larry, without doubt, goes the distinction (however dubious) of having shed the second greatest amount of blood for ILLIAC IV. Larry had several set of interests in the machine. First there were the direct applications that had been identified including numerical weather prediction, sonar, radar and seismic signal processing and the, by now, usual list of computations that array computers do efficiently. Larry also shared my sick attachment to really big pieces of hardware. But in ILLIAC IV Larry also had the interest of the father of computer networks. Particularly, as the program's costs escalated from my initial 10 million dollar guess (the sometime alleged relationship to the Sonny Liston-Cascius Clay (now Muhammed Ali) fight gate receipts is neither totally true nor totally apocryphal) to the more than 30 million it ended up costing for one fourth of the initial system. To justify these escalated costs the ultimate availability of the machine to an ever larger community of users became mandatory and ILLIAC IV and the ARPANET became inseparably linked.

In 1969 the strains of running a million dollar per month project within an academic department were operating destructively. The relations between John Pasta and me and between ILLIAC IV and non-ILLIAC IV faculty generally had degraded beyond the point of repair and the project was made a free-standing Center in the Graduate College of the university. It was a terrible mess, due mostly to circumstances but in no small part to me, and I regret to this day the human hardships that resulted and the deep human relationships that were destroyed.

By the beginning of 1970 the hardware program had been marginally restabilized; the PE had been redone with the new lower level T.I. integrated circuits, PE memory chips were being delivered by Fairchild, boards were being produced at a decent rate, cabinets and cables existed and our internal departmental conflicts were coming to a head on a campus and in a country that was becoming unglued by the Vietnam War. In May of 1970 both the U.S. action in Cambodia and the Kent State shootings took place and my generally conservative position with regard to the war became untenable to me. I informed the university administration and ARPA that I thought it wrong for the ILLIAC IV to be installed and operated on the campus and that if it was I would play no part in it. The reaction by campus administrators was consistent with all my previous observations: They ranged from the proposal that the future location of the machine be decided by a binding student election, to delivering the machine to a profit company to be set up near campus and protected both by high walls and armed guards. In the presence of continuing demonstrations, frequently violent, by as many as 6 or 7 thousand students which were sharply focused at the project, these suggestions by my "superiors" only supported my conviction.

Due mainly to the efforts of Dr. Hans Mark, then Director of Ames, ARPA decided to ship the machine to the NASA Ames Research Center in Moffett Field, California, announcing their decision on 29 January 1971. In April 1972 Burroughs delivered the system to Ames. It was plagued by a variety of serious hardware problems. Some of the early circuit batches failed both at high rates and in modes troublesome to detect. Moreover, huge numbers of back-panel connections and of terminating resistors were equally bad. Although some successful runs were made as early as 1973, the machine wasn't running reliably until 1975. The story of ILLIAC at Ames is, however, not mine to tell.

The story of the ILLIAC IV hardware is, however, only a part of the ILLIAC IV story and perhaps not the most important part. From 1965 on, as a result of the ILLIAC IV program, first the Urbana campus and subsequently many other university, government and industrial laboratories have undertaken the analysis of the relations between computer architecture, algorithms, both numerical and non-numerical, and their expression in programming languages. The problems posed so long ago in [Cocke 1958] have begun to receive the attention that I believe they merit. Moreover, as I will presently discuss, parallel computation (or pipelined computation with which it shares many of its benefits and burdens) will not be the sole beneficiary of this attention. My own opinion is that the greatest advances in the efficient use of new architectures will accrue from research in numerical algorithms and that the benefits yielded by new languages will continue to primarily benefit the sanity and efficiency of the programmer and be of only secondary concern to the programee.

This orientation certainly influenced my priorities in running the ILLIAC IV program. While I think the ILLIAC IV language development work of Dave (David J.) Kuck and Duncan (H.) Lawrie has had lasting value and has influenced many other researchers, my main concern remains with the part of the problem solution process that, though cognizant of the machine logical structure, goes on before the programming starts. The research of Ahmed Sameh, whose distinguished career also started with ILLIAC IV, exemplifies this position. We have discussed these matters at some length in [Slotnick 1978].

It is time, in concluding this section, for the pleasurable business of acknowledging some of the ILLIAC IV principals. It is also a risky business because there are doubtless some I'll forget and others I'd just like to forget. I've already mentioned Kuck, who launched the language and application programming efforts and the one man gang Duncan Lawrie who produced GLYPNIR, which for ten years was the only working ILLIAC IV higher-level language, and did it by himself as a lowly graduate assistant. I've also mentioned Sameh who essentially initiated, and remains a principal contributor to, the field of parallel algorithms for calculations in linear algebra. Bob (Robert S.) Northcote made contributions in software as did a group of some of the brightest kids (many of whom now as middle-aged men remain among my friends and collaborators). I've ever known, including Pete (Peter A.) Alsberg, Gary (G.) Grossman, Dave (David M.) Grothe, Nelson Machado, Jimmy (James M.) Madden and Jim (James E.) Stevens.

On the hardware side there was the constant support of the many-sided Art Carroll and Dave (David E.) McIntyre. Masao Kato's energy and skill was an inspiration to all of us. For several of the early years Dick (Richard M.) Brown helped hold things together and there was always, at the center of the storm, Frieda Anderson. At Burroughs (E.) Gary Clarke, with whom I share 20 years of richly varied memories, started things off and remained the Godfather, the only one I could always talk to was Vern (Vernon Z.) Smith who also knew the best jokes. Dick (Richard A.) Stokes and George (H.) Barnes did creative technical work. At T.I. Harvey (H.C.) Cragon and Joe Watson led my good list and (J.) Fred Bucy was on it. I also had another list.

Reflections

I have a bit more to say about parallel computers. First, one of the most interesting directions remains insufficiently explored. In [Slotnick 1970] I outlined the simple notion of associating with each track of a rotating store some minimal logic capacity, much like the SOLOMON I PE. The resulting system I called a logic per track disk in evident generalization of the head per track disk. Such a system could, like my drum of old, search or process the complete contents of a rotating store every revolution and thus function as a calculating, and/or associative store. Figure 5 shows the general idea. The PE logic which looks at the tracks (I used a pair, in strict analogy to the SOLOMON frames and 2-address scheme) is just a simple serial bit stream (pair) processor with mode and routing logic. Nowadays one would want to include a few characters of RAM in the PE instead of the revolvers that were then appropriate. One could also think of replacing the disk tracks altogether by CCD's or bubbles. The optimal parameters of a modern serial memory hierarchy for such a system would depend strongly on the overall system size and application scope. My student Stu (Stewart A.) Schuster and his Toronto colleagues continued this line of investigation, which they described in a sequence of publications starting with [Ozkarahan 1975], but only some small scale models have thus far been built.

SOLOMON-like machines have been built by ICL [Parkinson 1976] and others. Even now, the MPP [Fung 1977] currently being built is a larger (128 by 128) machine of this same general class. It remains to be seen whether parallel machines with floating point PE's are a dead end. I don't, however, consider the question to be of the first importance for reasons I will presently get to.

I believe it has value to reflect on the aetiology of some of the major problems that occurred during the ILLIAC IV development. First and foremost, trying to provide technical leadership as well as administrative direction to a program of this magnitude from a base in a traditional academic unit or from anywhere on any university campus made all the sense of trying to build a battleship in a bathtub. We had neither the facilities nor personnel to manage from a distance and even if we did, our temperaments, as intellectually driven monomaniacs, demanded being in the middle of every significant decision and altogether too many insignificant ones. It was only at great personal cost that, several years into the program, I disciplined myself to the slightly lesser of the evils: more global management.

We also took on too much. Some of the battles were unnecessary. I obsessively wanted every bit of speed we could get from any source. My hindsight is clear, I should have used more comfortable technology; our role there was not indispensable. By sacrificing a factor of roughly 3 in circuit speed it's possible we could have built a more reliable multi-quadrant system in less time, for no more money and with a comparable overall performance level. This same concern for the last drop of performance hurt us as well in the secondary (parallel disk) and tertiary (laser) stores. But this is all looking backward which violates my nature. I would rather conclude by looking ahead, at new directions suggested by current technology, with the benefit of these experiences.

Let me say first that it will probably be quite a while before even every Cub Scout Troop much less every household has its own design automation system with direct links to its companion, computer-controlled electron beam lithography VLSI fabrication system. In the interim such systems will remain the possessions of a relative handful of manufacturers. These manufacturers will use them to manufacture only those systems for which they believe there is a sizable market. This is counterpoised to the first twenty years of the computer era, when the relatively simply attainable state-of-the-art development capabilities were shared by industry with both university and government labs. The result is that technology which intrinsically has the capability to launch an unparalleled era of system experimentation shows little sign of fulfilling this potentiality. While no single university can afford the many millions of dollars required to create and operate a facility that would be capable of turning out strange and occasionally wonderful prototypes, it would not be too much for an appropriately backed consortium. The technology is nearly at hand to permit serious experimentation with dozens of new and promising computing structures. It is with this in mind that I earlier said it didn't really matter whether centrally-controlled arrays with floating point PE's are or aren't a dead end. There can, in the near future, be many special-purpose systems developed to solve this or that particular class of large-scale computational problems; on a bad afternoon I can think of a dozen myself. The question is only whether the capability will become a reality. We must ask ourselves would we have charcoal-broiled steaks today had Prometheus given fire only to the Chrysler Corporation.

I don't want to finish without an explicit statement of my view of the field's evolutionary potential. First, to think of supplanting the primary role of the conventionally organized (Babbage-von Neumann) computer is nonsense. It is, literally, an epoch-making concept. What can, however, take place is the evolution of large systems (and I, of course, have reference only to large systems) to comprehend entire families of special- purpose "peripheral devices" in a way not different in principle than the way they now comprehend their library of programs.

References

Ball, J.R., Bollinger, R.C., Jeeves, T.A., McReynolds, R.C., and Shaffer, D.H., 1962, "On the Use of the SOLOMON Parallel-Processing Computer", Proc. 1962 Fall Joint Computer Conference.

Barnes, G.H., Brown, R.M., Kató, M., Kuck, D.J., Slotnick, D.L., and Stokes, R.A. 1968, "The ILLIAC IV Computer", IEEE Trans. on Electronic Computers, 17, August 1968 also in Bell, G. and Newell, A., "Computer Structures: Readings and Examples", McGraw-Hill, 1971.

Cocke, John and Slotnick, D.L., 1958, "The Use of Parallelism in Numerical Calculations", IBM Research Memorandum RC-55, July 21, 1958.

Fung, Lai-Wo, 1977, "A Massively Parallel Processor", Proc. of the Urbana Symposium on High Speed Computer and Algorithm Organization, April 1977.

Gregory, John, and McReynolds, R.C., 1963, "The SOLOMON Computer", IEEE Transactions on Electronic Computers, Vol. EC-12, No. 5, Dec. 1963.

Kuck, D.J., 1978, "The Structure of Computers and Computations", Volume 1, p. 263, Wiley, 1978.

Ozkarahan, E.A., Schuster, S.A., and Smith, K.C., 1975, "RAP-An Associative Processor for Data Base Management", Proc. National Computer Conference, 1975.

Parkinson, Dennis, 1976, "Computers by the Thousand", New Scientist, 17, June 1976.

Slotnick, D.L., 1966, "ILLIAC IV Design Questions-Preliminary List" File No. 693, Dept. of Computer Science, Univ. of Illinois, April, 1966.

Slotnick, D.L., 1970, "Logic per Track Devices" in Alt, F.L. and Feiberger, W.F. (Editors), "Advances in Computers", Vol. 10, Academic, 1970.

Slotnick, D.L., 1971, "The Fastest Computer", Scientific American Magazine, Vol. 224, No. 2, Feb. 1971.

Slotnick, D.L., Borck, C.W. and McReynolds, R.C., 1962, "The SOLOMON Computer", Proc. 1962 Fall

Joint Computer Conference, also in Swartzlander, E.A. Jr. (Editor) "Computer Design Development-Principal Papers", Hayden, 1976.

Slotnick, D.L., Borck, C.W. and McReynolds, R.C., 1963, "The SOLOMON Computer-A Preliminary Report", Proc. 1962 Workshop on Computer Organization, A.A. Barnum and M.A. Knapp, editors, Spartan, 1963.

Slotnick, D.L., and Sameh, Ahmed, 1978, "Numerical Calculation and Computer Design", Computers and Mathematics with Applications, Vol. 4, No. 3, 1978.

Westinghouse Defense and Space Center, 1964a, Computer and Data Systems Technology Group, Final Contract Report AF 30(602) 3417, "Multiple Processing Techniques", Submitted to RADC, Griffiss AFB, N.Y., 10 April 1964.

Westinghouse Defense and Space Center, 1964b, Aerospace Division, "Proposal for Parallel Network Processor", Negotiation No. J0417, for Univ. of Cal. Lawrence Radiation Lab, 14 August 1964.

Zuse, K. "Die Feldrechenmaschine", 1958, Mathematik, Technik, Wirtschaft-Mittelungen, Vol. 4, pp. 213-220, 1958.

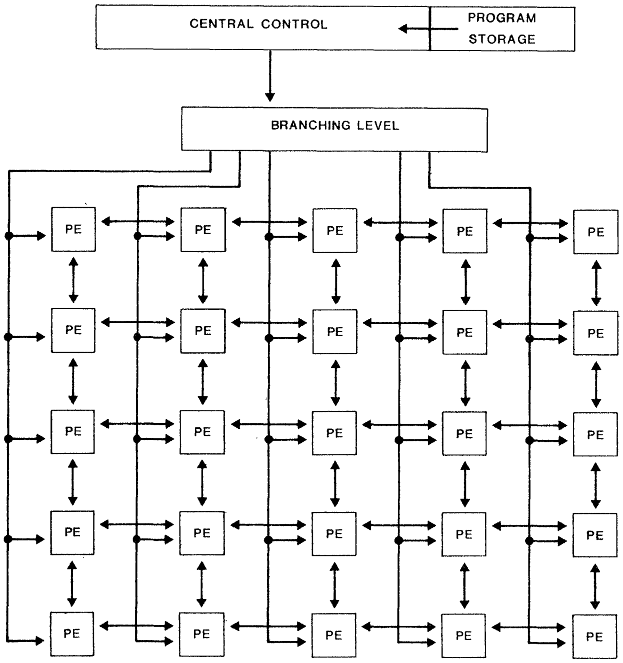


Figure 1. PE Array Under Central Control

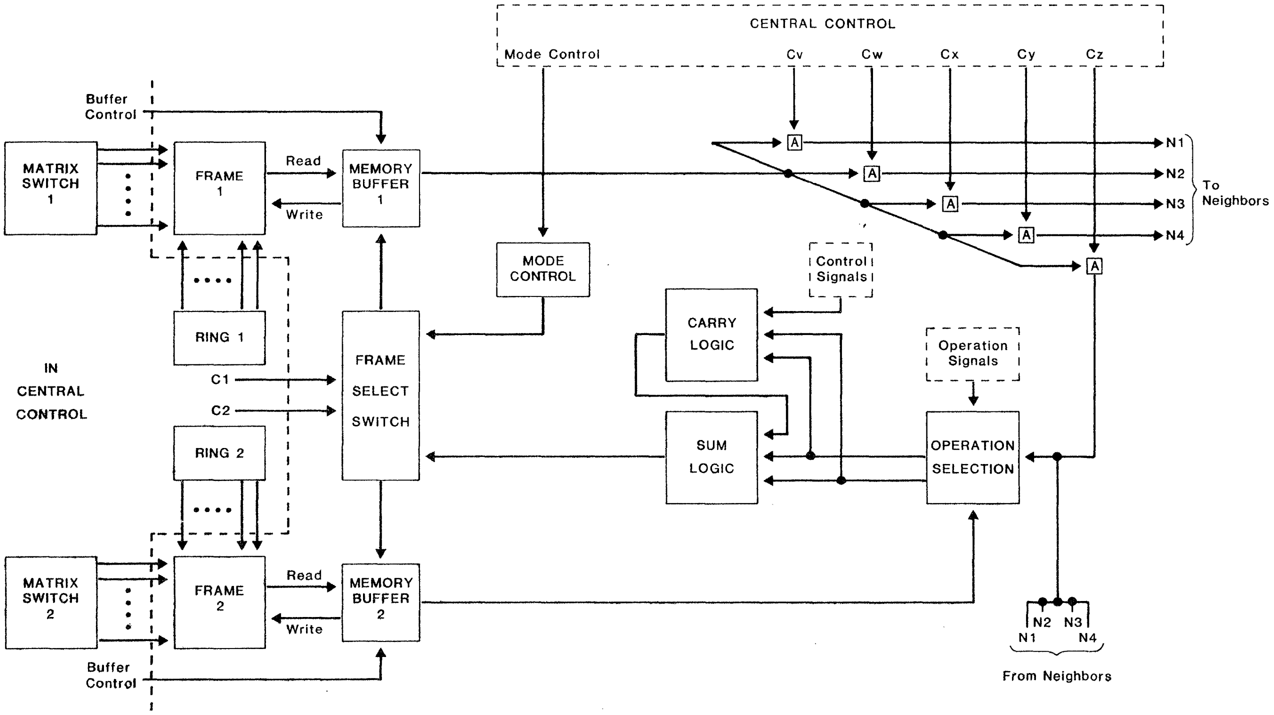


Figure 2. Processing Element (PE) Block Diagram

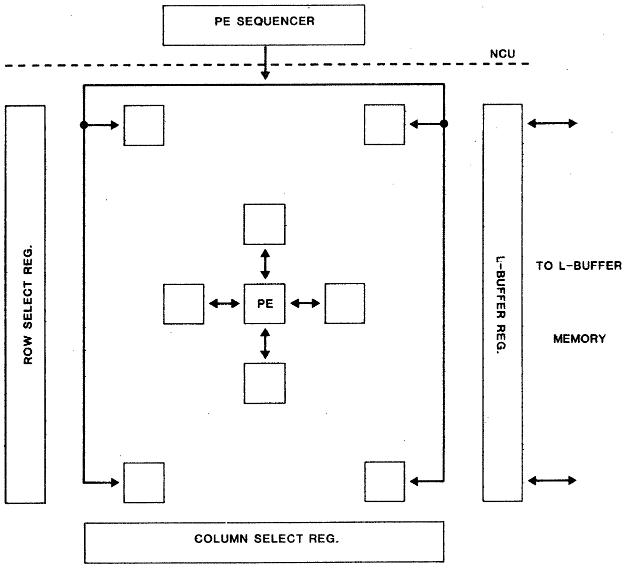


Figure 3. PE Network Organization

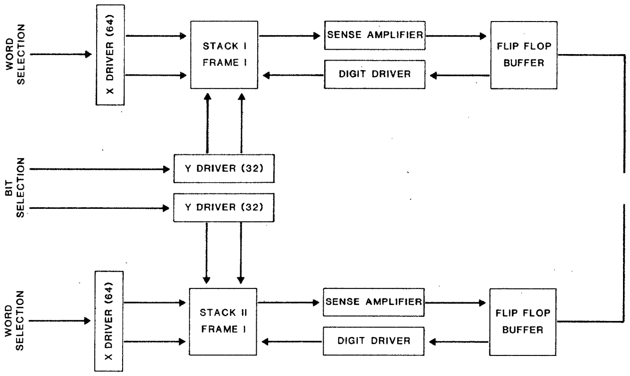


Figure 4. PE Memory Organization

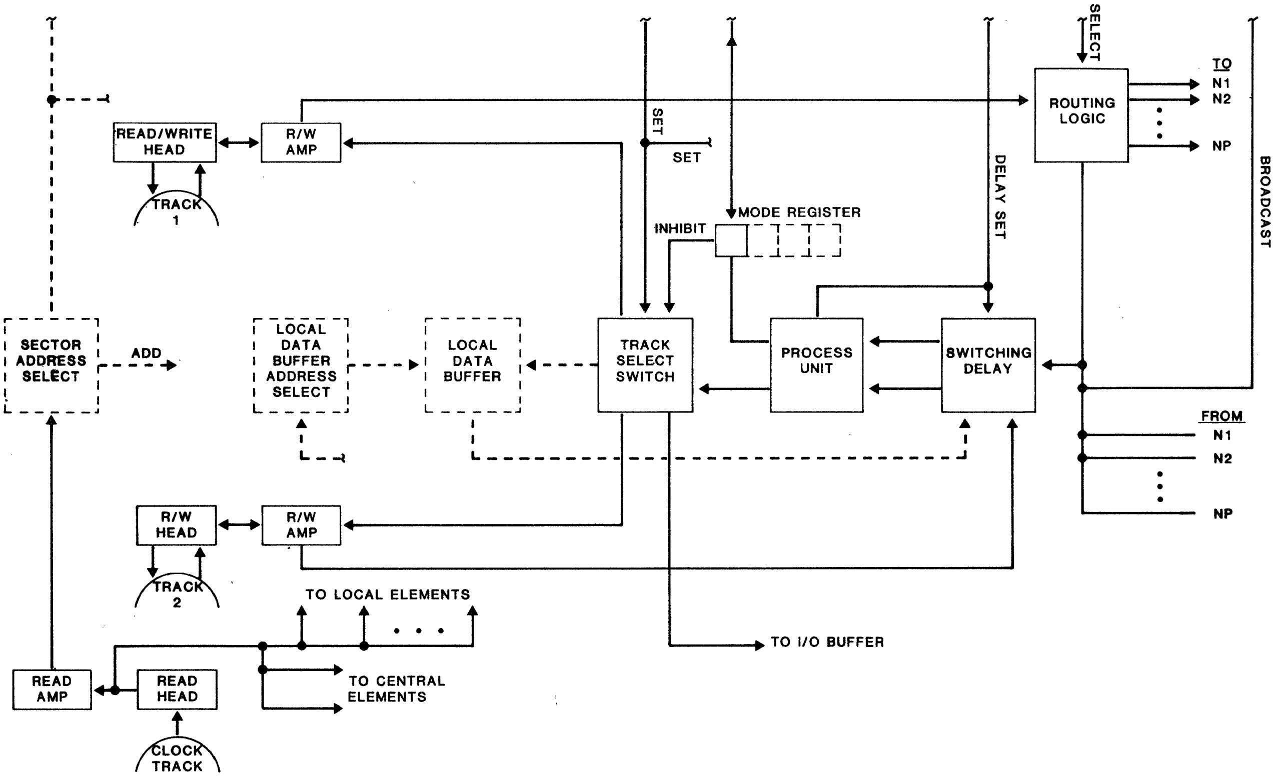


Figure 5. Logic Per Track System

THE HISTORY OF PARALLEL PROCESSING AT BURROUGHS

Richard Stokes and Robert Cantarella
Federal and Special Systems Group
Burroughs Corporation
Paoli, PA 19301

Introduction

Parallel processing in the context of the Burroughs experience has been synonomous with the development of the "supercomputer". While it is accurate to claim that, throughout the Burroughs standard product line, the application of parallel processing design is in ample evidence, the main stream of the work on supercomputers is centered in the Federal and Special Systems Group, Paoli, Pa. For almost two decades, the challenge of the parallel machine has been actively pursued without interruption. In that time a series of major systems have been developed, starting with ILLIAC IV, then PEPE, followed by BSP; and this paper describes the historical events in the development of these systems. A new parallel design currently under study for NASA called the Flow Model Processor (FMP) is not discussed here.

These machines as a group represent some of the most ambitious undertakings in the industry (Table 1). With the exception of the FMP, all have been completed in a fully working sense, and all substantially met their original design objectives.

As a group they are certainly a tribute to the designers whose skills harnessed enormous quantities of logic and memory circuits in concerted processing functions. Their contribution to computer science has been made, but perhaps not fully realized. The design rationale of these machines as a machine class (SIMD) provides the only demonstrable performance response for that class of large scientific applications that have vectorizable programs.

This 19-year history is intended as a synopsis of the plans, events and results of three major engineering experiences at the Burroughs Great Valley Laboratories. Unfortunately history, like art, is seen through the mind of the beholder and where serious omissions or errors occur they are certainly not intentional. The lessons learned and the experience derived from these endeavors are continuing to serve our engineering staff in the development of the FMP.

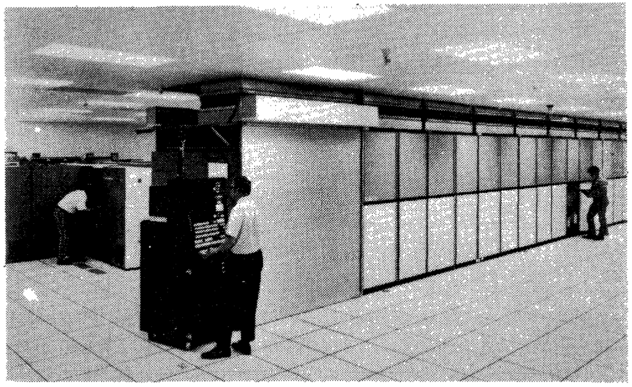
Table 1. Comparison of Parallel Processor Capabilities

	PEPE	ILLIAC IV	BSP
Data Word Size	32 bits	64 bits	48 bits
Instruction Word Size	32 bits	32 bits	24-48 bits
Backing Store	In host	Paged to PE	N-Mos RAM
Memory Cycle	100 ns	250 ns	160 ns
Number of Processing Elements	Up to 288	64	16
Processing Element	32-bit floating point accumulator oriented	66-bit floating point accumulator oriented	48-bit floating point memory oriented.
Microprogrammed	Yes	Yes	Yes
Processing Element Connections	Linear array	4 nearest neighbors	Cross Bar
Parallel Operation Within Arithmetic Unit	Yes	Yes	Yes
Associative Addressing	Yes	Pseudo	No
High Order Language	PFOR	GLYPNIR	FORTRAN
Processing Speed			
Add	300 ns[1]	500 ns[1]	160
Multiply	1.9 us[1]	700 ns[1,2]	320

1. Time for one PE; all PEs may operate in parallel
2. Two operations may complete in this time
3. May be computed as N^2 times 0.85 s, where each operand is assumed to consist of N bits.

ILLIAC IV

The ILLIAC IV computer was a product of the mid-sixties, its original goals reflecting the prevailing optimism in the country and particularly in the young computer industry. It was the era of the "main frame houses" that continued to demonstrate Groche's Law with regular ease.



Illiac IV Installed at NASA Ames Research Center, Mountain View, California

The seeds of the ILLIAC IV program evolved from a project called Solomon developed at the Westinghouse Corporation in Baltimore, Maryland. The circumstance that marked the official beginning of the ILLIAC IV program was the move by Dr. Daniel Slotnick, a Solomon principal, from Westinghouse to the University of Illinois and the subsequent designation of that institution as the prime contractor by the Advanced Research Projects Agency of the Department of Defense.

The program plan was to have the University develop the system software and subcontract the hardware development on the basis of a competitive proposal. Study definition contracts awarded to Burroughs, Control Data Corporation and RCA resulted in three proposals in which Burroughs was awarded the hardware development contract in 1967.

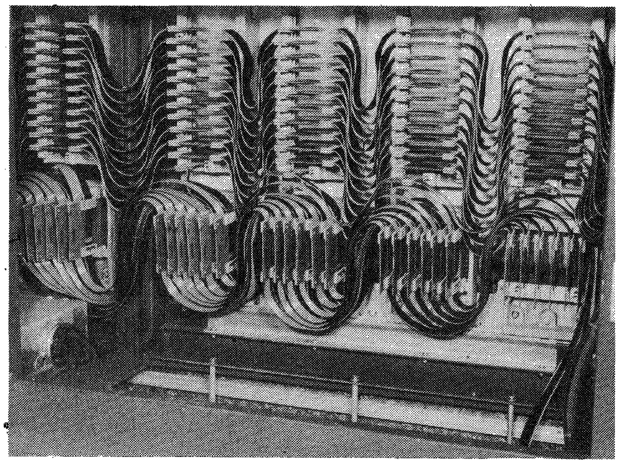
The central objective of the system was 10^9 operations per second. This, of course, placed considerable emphasis on hardware component speeds and parallel architectural design [1]. The proposed system contained 4 independent quadrants of 64 Processing Elements (PE) each, for a total of 256 PE's. Each PE contained an arithmetic element and a data memory and was interconnected to other PE's which were a distance of ±8 and ±1 in designated value. Thus in a 8 × 8 array, a nearest neighbor connection pattern was realized.

Each quadrant was driven by a Control Unit decoding a single instruction stream and broadcasting the microstep for array instruction execution. The Control Unit has a program memory and a separate station for executing CV instructions concurrently with array instruction. ILLIAC IV was a classical SIMD design.

The Hardware

The key components of the system design were: plainer thin film memories and multichip ECL logic circuit packages. Later events were to show that both choices were not realizable in the final system.

Thin film memories had been in development in Burroughs and elsewhere for several years prior to the start of ILLIAC IV. Thin film was considered the performance successor technology to magnetic cores and Burroughs was actively engaged in the process of moving this technology from the laboratory into production. Two factors conspired to preclude this expectation before production was realized: the tenacity of magnetic cores and the pace of semiconductor memories. When this situation became apparent, thin films were discontinued, as a product and, in turn, for ILLIAC IV.



ILLIAC IV Backplane

Upon the demise of thin film memory at Burroughs, a contract was awarded to Fairchild Semiconductor for the PE memory system using a 64-bit bipolar component. This contract was one of the more successful projects of ILLIAC IV, calling for the design and production of 70 memory units, each with a capacity of 4K words. Considering the tight schedule and the new technology, many things that might have gone wrong did not: the memories were delivered on schedule and to specification.

The total capacity of 250K words, limited by cabinet volume, was a performance disadvantage for the growing application programs that were run on the system.

As part of the Burroughs proposal, Texas Instrument Corporation, acting as a subcontractor to Burroughs, agreed to provide the Processing Elements (PE) of the system, fully assembled and tested. A PE was a 64-bit floating point arithmetic [2]. The design was based upon a multichip package in which four (up chips) were mounted on a common substrate and interconnected by wire bonding. The circuit packages, 24-pin ceramic, were to be connected on a multilayer printed circuit board, one per PE.

The published reason for the termination of the multichip development by the contractor was low production yield. The design process contained the fundamental weakness of the multichip approach by postponing testing to a complexity level not justified by the value added and not repairable.

The fall-back position was the use of the more conventional 14-pin DIP packaged ECL on smaller, 2-signal-layer, printed circuit boards, connected by a wired backplane. The logic circuits used were the TI2500 circuit family, implying that the fault of the initial design was the package scheme.

The foregoing component problems were the major ones and contributed to schedule delays and cost increases for redesign. In time, the program scope had to be reduced from four to one quadrant (256 PE's to 64 PE's) where the 10^9 operations per second would not be possible.

The Software

The system software development was the responsibility of the University of Illinois, which undertook the development of a new Algol-like compiler called TRANQUIL [3]. In addition, an assembly language development called GLYPNIR [4] commenced at about the same time.

TRANQUIL was, of course, a major undertaking dealing with a parallel structure unlike any previous experience in compiler design. It contained language extensions to allow the users to identify parallel (vector) constructs and to manage the conditional states of the PE array. A preliminary version of TRANQUIL was completed and compared against the available GLYPNIR for object code performance.

The results were disappointing but not necessarily unreasonable for the early stage of the compiler. TRANQUIL, however, was discontinued and GLYPNIR became the principal language for programming ILLIAC IV. Later, after the system was installed at NASA Ames, another language emerged called CFDL (Computational Fluid Dynamic Language). CFDL was based on Fortran and supported the principal applications for that agency.

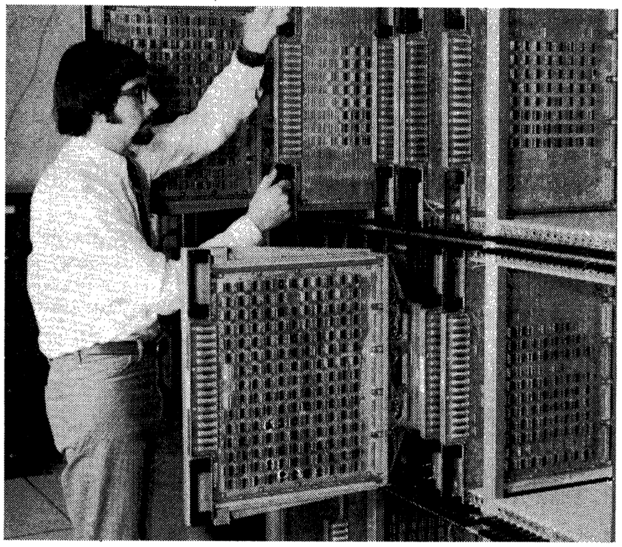
The Completion

The ILLIAC IV system was shipped to NASA Ames in April 1972 and was accepted by the customer that December. The selection of the NASA site in lieu of the original one at the University of Illinois was due in part to the campus unrest of that era and the possible target the system presented. The system has been operational now for almost a decade and is considered an effective and productive resource in the mission of that agency.

To the people who designed and built the ILLIAC IV, it was certainly a triumph of skill and determination. The size and complexity of the system (250 thousand, dual, in-line components) is a challenge by today's standard. ILLIAC IV also made its contribution to the science:

a) It demonstrated that a SIMD architecture could be used effectively on some important applications.
b) It showed that a system of that size and complexity could be used productively and reliably.
c) It made the user community "vector conscious" and motivated the work toward vectorizing compilers and the inclusion of vector operations in later product designs.

A major drawback to a wider use of ILLIAC IV was the evolution in user environment. Modern compilers and operating systems removed the user from the hardware details of programming. The programming pioneering days were coming to a close.



ILLIAC IV Control Unit

PEPE (Parallel Element Processing Ensemble)

The history of PEPE development discloses a number of different corporations that contributed in varying measure to the final delivered product. PEPE as an architectural concept began in the mid-sixties at Bell Laboratories, New Jersey, under the auspices of the Army Ballistic Missile Defense Agency (ABMDA). An early prototype was assembled there at the time AT&T decided to divest itself of military development contracts. As a result, the System Development Corporation took charge of PEPE and, in turn, engaged Honeywell in support of the hardware design.

In March, 1973 Burroughs was awarded a contract by SDC to build a revised and enhanced version of PEPE for ABMDA, Huntsville, Alabama. The system Burroughs was contracted to build was specified in detail, focusing primarily on the problem of radar data processing for missile defense systems.

The execution of the contract by Burroughs is considered an industry paragon and Burroughs was singled out for an outstanding performance award by the U.S. Army for this achievement. The completed PEPE system was shipped from Burroughs Great Valley Laboratories, Paoli, Pa. to Huntsville in May 1970 and accepted by the customer by November of that year. The only significant change from the original contract was the reduction of the number of processing elements from 36 to 11 due to a program funding reduction.

The Design

The PEPE design is considered special purpose because it is driven by the single application of radar target correlation and tracking. This application naturally lends itself to parallel processing since the processing functions are identical for multiple target returns and predictions. The PEPE is really three distinct linear arrays, each of which performs the parallel functions of correlation, tracking, and radar control, respectively. A Processing Element is a single orthogonal slice of these hardware elements, including a common memory and incorporating each of the three functions.

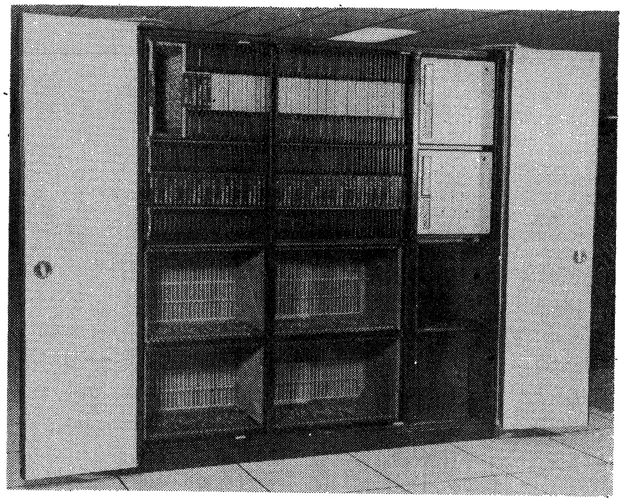
Another important aspect of the PEPE application is that there is no requirement for inter-PE communication. This permits the PE's to associate in a loosely coupled "ensemble," with a significant reliability advantage as a result. Multiple failures in PE would degrade but not fail the system. The system was packaged with 36 PEs in a cabinet and a maximum of 288 PEs was permitted.

The logic component family used in PEPE was the Motorola 10K ECL Family. MECL 10K was a mix of MSI and SSI completely packaged in ceramic DIPs. The memory was a 1K bipolar RAM produced by Fairchild Inc. The novel design of the printed circuit boards featured a combination of printed wiring and wrapped post wiring that avoided the problems of multilayer boards. This design, called the composite board, was used successfully on the BSP.

The Epilog

The PEPE system was interfaced with a CDC 7600 host system in the Huntsville complex and used to develop application programs. Later the system was shipped to McDonnell-Douglas, Huntington Beach, California for its intensive benchmark testing. These activities are classified and the results cannot be published here. It can be reported, however, that the hardware performed exceedingly well and the system was returned to Huntsville.

The PEPE contribution might have been more formidable if the world political climate had warranted it so it may be assumed that it fulfilled a vital need. From an engineering viewpoint, it was simply a job well done.

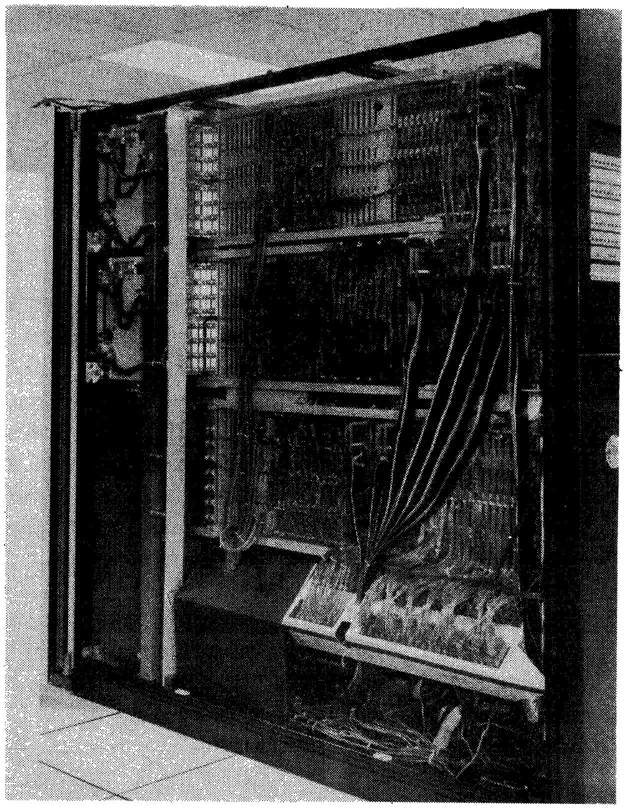


PEPE Cabinet, Front View

BSP (Burroughs Scientific Processor)

The Burroughs Scientific Processor (BSP) was the result of an effort to develop a standard product supercomputer that would serve the scientific user community with massive computational requirements. This application requires machines with special architectures that can perform at levels beyond those achievable by circuit speed alone.

Fortunately, the programs often exhibit an internal structure in which the same operator can be applied to arrays or vectors of data. This had led to the development of several SIMD supercomputers of either an arithmetic pipelined or parallel processor design (e.g. ASC, STAR, and ILLIAC IV [1]). Both techniques had resulted in vector computers



PEPE Backplane

whose effective computational rates on suitable applications were one to two orders of magnitude greater than that of serial processors constructed of equivalent speed circuitry.

The generality of these machines was limited by restraints on the application programs. Due to pipeline start-up time, very long vectors of data were often required. A small scalar content could seriously degrade performance levels. Finally, they were difficult to program, often requiring assembly language coding and memory residency analysis in order that the speed of the machine be fully realized.

For these and other reasons, the only machines that had achieved commercial success by the early 1970's were the CDC 6600 and 7600 series which achieved their performance levels primarily by the use of very high speed circuitry and multiple function arithmetic processors.

Given the recently completed ILLIAC IV program and ongoing PEPE program, Burroughs had developed expertise in parallel processing which could be applied to developing a commercial supercomputer. This, coupled with the Corporation's desire to field a FORTRAN processor to complement the product line and provide a test bed for a new generation of high speed current-mode logic (BCML), provided the impetus for the development.

Although the BSP was not commercially successful, prototype and production models of the BSP were built, made operational, and in fact, met most of their design goals. The state of the computing art was advanced in several areas.

Design Goals

The beginnings of the program can be traced to a feasibility study on repackaging ILLIAC IV which was conducted in 1972. A survey of the user community clearly showed that a more refined, easier to use machine was required. This led to the development of the set of design goals listed below.

Standard Product. The BSP was to be a standard product. This implied that it was to conform to the corporate standards for manufacturability, testibility, reliability, maintainability, high level language programmability, ease of use and cost. It would be developed and manufactured by a standard M&E (Manufacturing and Engineering) plant. Corporate standard hardware technology was to be employed, providing a volume basis for material costs and manufacturing tooling.

Attached Processor. The BSP was to be attached to a large scale commercial computer system such as the B 7700. This provided the capability to extend the FORTRAN performance of these machines and provided the user with access to the sophisticated system software developed for commercial large systems.

Technology Driver. The Corporation was currently engaged in the development of a high speed current mode logic family and its associated liquid cooled packaging technology, intended for use in Burroughs commercial plants. The BSP was to be a driver for this program. Thus it would provide schedule pressure on the components plants in advance of commercial requirements and be a test bed to shake down the technology.

Programmability. The BSP was to be efficiently programmable exclusively in a high order language. In practice, this meant that FORTRAN was the obvious choice. Any extensions were to be application oriented and machine independent. A vectorizer was to be provided as a means of efficiently executing existing codes.

Ease of Use. The machine was to be easy to use. This was motivated by users' desire to minimize the cost of developing and maintaining application codes.

Performance. The BSP was to be capable of sustaining 20 to 40 MOPS on typical application codes in weather forecasting, nuclear reactor design, structural analysis, and other similar fields. This was to be measured on such standard benchmarks as the Livermore Loops.

In order to achieve these goals, several key technical problems had to be solved.

Scalar Problem. Some means had to be found to minimize the impact of scalar processing. This had been a bottleneck in then-current designs.

Pipeline Start-up and Short Vector Performance. A method had to be found for ameliorating the effect of pipe-start-up time so that high performance could be achieved on relatively short vectors.

Memory Conflicts and Residency. A memory structure had to be devised that would minimize the effect of memory conflicts which occurred when elements of operand vectors resided in the same memory bank. This structure could not require the user programmer to exhaustively study the application and specify special residency requirements.

Automatic Bit Vector Control. Bit vector control for data dependent branching and sparse vector operations had to be built into the machine and made easy to use.

Generalized Parallel Processing. The parallel processor had to be generalized so that it could be effectively employed in more applications. Research in parallel processing had resulted in many parallel algorithms for speeding up operations previously thought to be serial (e.g. linear recurrences [8]).

Balanced I/O Structure. High performance secondary store was required and had to be accessible without excessive operating system overhead.

Self-checking and Fault Tolerance. Extensive self-checking and fault tolerant mechanisms were to be built into the machine so that high reliability and trustworthiness could be achieved. This was to be done without seriously degrading the performance of the system.

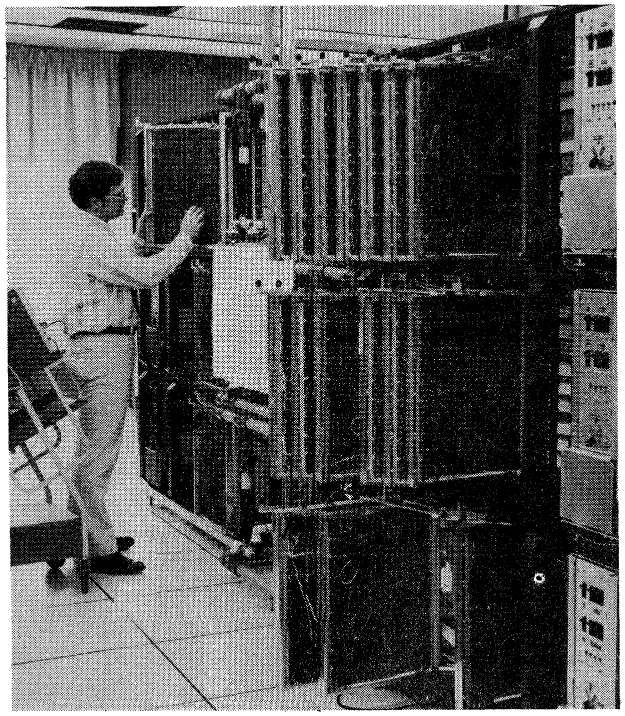
Architectural Design

The solution of these problems was undertaken during the preparation of the PDA (Product Development Authorization — an internal proposal). This effort was completed in June, 1974.

The first issue to be decided was whether a pipelined or parallel processing approach would be taken. The latter was chosen because of the ease of implementing many of the sophisticated algorithms which had been discovered and the expertise which had developed during the ILLIAC IV program. Finally, the iterative nature of parallel processors made them more suitable for VLSI implementation in the future.

Once this had been decided, the memory conflict problem was then attacked. Although many skewing techniques were known for minimizing conflicts, none had the generality and uniformity that was desired. The result of this effort was a scheme [9] which offered conflict-free access to any linear vector whose skip distance was not a multiple of the prime number of memory banks. Even more importantly, the memory mapping was application independent.

The use of microprogramming was explored as a method of simplifying the programming of the machine and as a means of directly executing many common FORTRAN constructs such as nested DO loops with embedded assignment statements. This resulted in the development of the template concept, which allowed the overlapping of vector operations within the temporal pipeline of the parallel processor and solved the pipeline start-up problem. (Parallel processors do exhibit another



BSP Cabinet

start-up phenomenon in that full speed is not achieved until the vectors are at least as long as the width of the array.)

The scalar problem was attacked with an eye to minimizing the number of scalar operations and overlapping their execution with that of the parallel processor rather than relying solely upon raw circuit speed. Scalar operations were reduced by the application of parallel algorithms, automating memory indexing and parallel processor control operations in hardware, and offloading I/O operations to a smart controller.

The remaining problems were solved in an exhilerating rush of discovery that culminated in a design which is remarkably similar to the final design documented in C. Jensen's paper [6]. The one major difference is that there were 67 slower dynamic memory banks which fetched vectors of length 64. The 16 arithmetic processors then executed the operation in 4 steps. Thus, the machine reached full speed at vectors of length 64. This allowed the use of low cost main memory.

BSP Demonstrating Class 6 Qualification

Detailed Design

In the detailed design phase of the program (June, 1974 to August, 1976) the implementation of the concepts developed during the proposal was pursued. It had not been clear that the alignment network and automatic indexing hardware could be built out of a reasonable number of IC's or that there would not be a combinatorial explosion of microcode. These problems were overcome and the design had successfully incorporated the features of the architecture.

The applications group had found that length of vectors in many codes were shorter than 64. It would be desirable to improve the short vector performance of the machine. The advent of low cost high speed static NMOS memories such as the 2147 made it possible to do this. The number of memory modules was reduced to 17 and the memory cycle time speeded up by a factor of 4. This allowed the parallel processor to come up to speed at vector lengths of 16 while providing the additional benefit of simplifying the design.

This had the result of throwing the design into imbalance. The scalar processor had to prepare descriptors four times as fast as before. The scalar unit had to be speeded up in order to fully take advantage of the faster parallel processor.

The Turning Point

A related sequence of events occurring in 1977 had a large effect on the program. It had been observed that the scalar unit was, itself, functionally complete and could be offered as a lower cost Attached FORTRAN Processor (AFP). This product appeared to be relatively free and was adopted. However, it resulted in two releases, two sets of software, the development of a DISK version of the I/O system, and an interface to the B 6800. This represented a significant additional workload on the project.

The BCML development was very late and did not meet the original performance goals. A proposal to implement the first machine in the proven hardware of the PEPE system was rejected because the objective of driving the technology was deemed essential.

It was becoming clear that the performance of the scalar unit would not support application programs that did not contain a sufficiently high content of vector operations. The design of the scalar unit was straightforward, to minimize the overall development risks to the program. The performance on the Livermore Logics benchmarks (a scalar-vector mix) reinforced our strategy, but a broader product approach would require a performance enhancement of the unit. At this point, with limited time and resources, it was felt the problem could be addressed in a subsequent product upgrade after the production start of the present design.

Making It Work

The machine was debugged during 1977 to 1980. There were many problems to overcome. Initially, late deliveries of circuits delayed the program. When sufficient quantities were available, the hardware was built and put into system test.

The hardware technology was completely new, from the circuits to all three levels of packaging. In addition, the emerging CCD technology was to be employed for a second level store. Given the number of new items, it perhaps is not surprising that some design problems surfaced.

The first design of the sockets exhibited loose contacts, the proms speeds drifted, and there was a damaging latent fault in the zinc pillow blocks. These blocks were screwed in to hold the PWB assembly together and were under high pressure. They exhibited a cold flow phenomenon which caused the screws to slowly pull out. The assemblies were literally pulling themselves apart. A third of the machine had to be reworked in the midst of debugging. The CCD devices exhibited a high soft failure rate and were difficult to manufacture.

These problems were overcome and the production hardware was fully qualified, very reliable, and exceptionally stable. There were practically no electrical intermittents reported. The CCD memory was replaced by a dynamic RAM system. While this process of shaking down the hardware technology fulfilled one of the main objectives of the program, it delayed getting the machine into the marketplace at a critical time when CRAY was making deliveries for almost 2 years.

The software set was new and fully featured. The maturization of this amount of software took a long time and prevented us from routinely running customer benchmarks. This was aggravated by the temporary loss of all 7700's for customer shipments, which resulted in no system manager to debug the deliverable software (the alternate, but different, 6800 software was used instead). Nonetheless, by 1979, limited benchmarks could be run to measure the performance characteristics of the system.

Performance Measurement and Marketing. In the codes that were tested, the design lived up to its promise as an excellent vector processor. The livermore loops ran at over 20 MOPS. In general, most comparisons showed that the machine was equivalent in performance to the CRAY I for many vectorizable codes. This was true even though the short vector performance of the parallel processor was only being partially realized and the hardware components were considerably slower.

Although the large main memory and fast secondary store was an advantage in large problems, users preferred the CRAY due to the guaranteed performance levels that could be achieved on existing non-vectorized and scalar codes.

Conclusion. The cancellation of the BCML and CCD programs, the attendant cost increases, the loss of an appropriate marketing window, and the lack of a dominant scalar speed led to the cancellation of the product. The design proved that it was possible to configure a parallel processor which was competitive in vector applications and considerably more general than those that had been designed in the past. This drive for generality is expected to continue into the next generation of MIMD architectures.

References

(1) Barnes, G. et al. "ILLIAC IV Arithmetic Element," **IEEE Transactions on Computers** (August 1968), Vol. C 17, No. 8, pp. 746-757.

(2) Davis, R. L. "ILLIAC IV Arithmetic Element," **IEEE Transactions on Computers** (September 1969), Vol. C-18, pp. 800-816.

(3) Abel, N. et al. "TRANQUIL - A Language for an Array Processing Computer," **Proceedings AFIPS Joint Computer Conference** , Vol. 34, pp. 57-73.

(4) Lawrie, D. "GLYPNIR - A Programming Language for ILLIAC IV," **Communications of ACM** (March 1975), Vol. 18, No. 3, pp. 157-164.

(5) Stokes, R. A. "Burroughs Scientific Processor," **Proceedings of the Symposium** on High Speed Computation , University of Illinois.

(6) Jensen, C. "Taking Another Approach to Supercomputing," **Datamation** (February 1978), pp. 159-172.

(7) Kuck, D. J. "A Survey of Parallel Machine Organization and Programming," **ACM Computing Surveys** (March 1977), Vol. 9, No. 1, p. 29.

(8) Chen, S. C. and Kuck, D. "Time and Parallel Processor Bounds for Linear Recurrence Systems," **IEEE Transactions on Computers** (July 1975), Vol. C 14, No. 7, pp. 701-717.

(9) Lawrie, D. "Access and Alignment of Data in an Array Processor," **IEE Transactions on Computers** (December 1975), Vol. C 24, No. 12, pp. 1145-1155.

CONTROL DATA 6600 AND STAR-100

James E. Thornton
Network Systems Corporation
Brooklyn Park, Minnesota 55428

Abstract -- This paper reviews some of the starting point assumptions and considerations for two design projects at Control Data Corporation; namely the CDC 6600 and CDC STAR-100. Each of these has had follow-on computer families, CYBER 70/170/700 for the former and CYBER 203/205 for the latter. Both projects were very ambitious and plowed new ground in the use of parallelism in large scale computer design.

The CDC 6600

The design of the CDC 6600 began in 1960 [1]. The first transistor computers had just been delivered to the field that year. Ideas having to do with parallel processing were presented at a short-course conference at UCLA in which STRETCH, LARC, ATLAS, ILLIAC-II and GAMMA 60 were examined. These machines were in development in the United States, England and France. Each attempted to exploit ways to reduce the idle activity within parts of the computer waiting for other parts to complete a sequential action. Since access to memory was often the biggest delay of this kind, many of the ideas had to do with relieving this burden. An example was the "instruction lookahead" of STRETCH.

In 1960 it had already become apparent (as it has grown more important over the years) that brute force circuit performance or parallel operation were the two main approaches to any advanced computer [2]. The 6600 project attempted a fast version of the building block circuit in use at the time only to fail early on in the project. This resulted in a restart with a more complex packaging and cooling scheme providing a significantly higher density of parts. Discrete transistors were used since integrated circuit families had not become available.

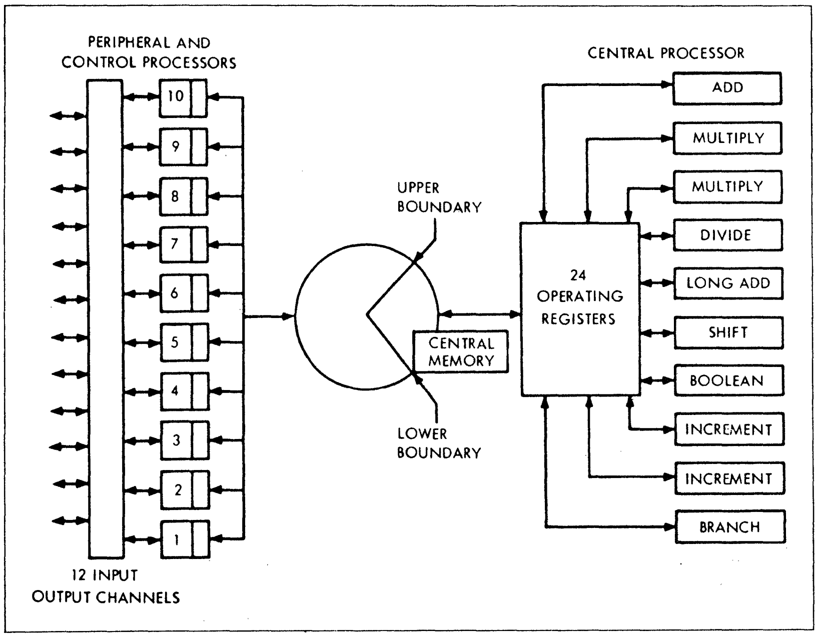


Figure 1. Block diagram of 6600

0190-3918/81/0000/0033$00.75 © 1981 IEEE

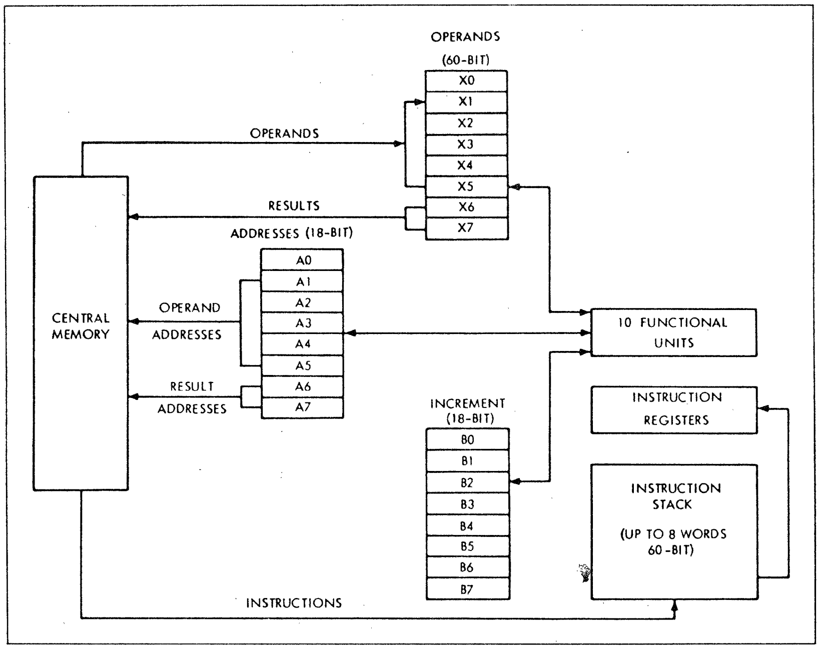


Figure 2. Central processor operating registers

The first area of parallel operation began as a separation of input/output operations from the CPU, (Figure 1). It was felt that independent small processors with direct access to central memory could be dynamically assigned to control peripheral devices and transfer data between each device and central memory. These peripheral processing units (PPUs) would contend with each other for the channels to the devices and for the access to central memory. In the latter, the PPUs would also contend with the CPU for access to central memory. With the exception of this latter factor, the PPUs could be designed separately from the CPU and represented a very convenient separation of design duties for the design team. The resulting design was an innovative "barrel" of registers together with a single arithmetic unit implementing ten small processors each having its own memory. Later implementations in follow-on CYBER products included physically independent PPUs as integrated circuit versions were introduced.

In the CPU additional areas of parallel operation were applied to instruction lookahead, multiple working registers and functional units. See Figure 2. Taking the registers first, the idea of inserting registers between the execution logic and central memory provided a means to optimize and overlap read and write references to central memory. At the beginning of the execution of a CPU program (or at a restart following interruption) a single simultaneous exchange is made of the contents of the 24 registers and a prepared location in central memory. This action provided very rapid exchange of jobs (or operating system routines) enhancing the ability to support multiprogramming.

During execution of the CPU program instructions are fetched from central memory into an instruction stack capable of containing 8 60-bit words, see Figure 3. Each new instruction word is immediately fed to the functional unitsfrom execution but is also retained and "pushed up" for possible re-use in certain loop routines or the like without the further requirement of fetching from central memory. In certain follow-on CYBER products of lower performance the instruction stack was not utilized nor were separate functional units. The instruction stack was a principal important ingredient in establishing a high degree of concurrent operation in the functional units.

A further important ingredient in supporting concurrent operation in the functional units was a control unit called the SCOREBOARD [3]. In essence, this unit kept track of reservations of the working registers allowing functional units to reserve each register it needed for either

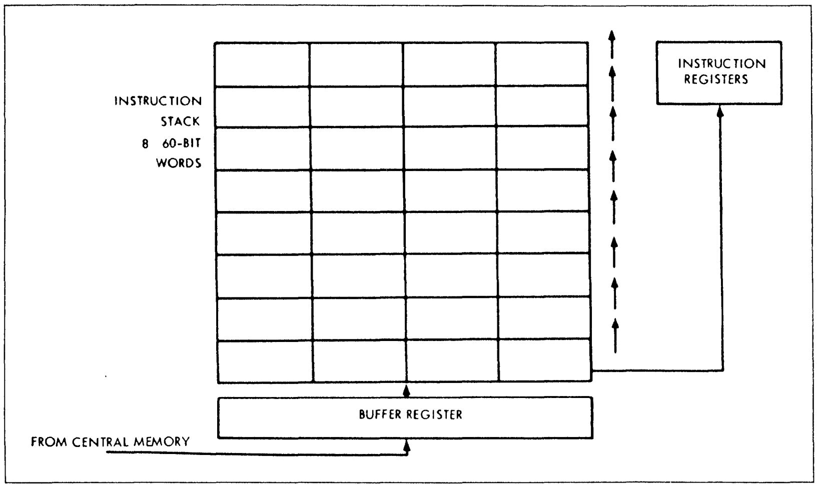


Figure 3. 6600 Instruction stack operation

reading out or writing into the register. Instructions could be "issued" to a functional unit in order, could be executed out of order, but would return results to registers in order. As a result, no functional unit would block the issuance of instructions unless a unit was busy or a register reservation could not be made.

Optimization of short program loops in the instruction stack could produce dramatic overlap of functions. Also simply the implicit use of this control technique and the existence of ten functional units provided a degree of natural concurrency.

From a design point of view each functional unit was specially designed to execute its narrow group of instructions interfacing only to the registers and minimal control signals. As a result several of the units achieved very high performance avoiding "impediments" of sharing logic with other functions. Later CYBER products incorporated further "pipeline" arithmetic design to such functional units.

Two principal criticisms were leveled at the CDC 6600. The first was that it was not a time sharing machine. This criticism has been hotly contested and arose from the lack of interrupt on the PPUs and the lack of virtual memory for memory management. A second criticism was the lack of variable length string arithmetic and instructions for character handling and decimal numbers. Software routines to accomplish these requirements proved slow in relation to the IBM 360 for example. For scientific and binary oriented computing though, this machine was superior for its day and for many years even to today.

The CDC STAR-100

Control Data began this project in response to a request for proposal (RFP) from Lawrence Livermore Laboratory (LLL). Preceding this were requests for information (RFI) and other interaction with the LLL people as to the possibility of CDC being the manufacturer of an ILLIAC IV type of machine. Management response to that suggestion was negative. Also our technical response was that we had a different idéa. The essence of this different idea was to build on our growing knowledge of "pipeline" architecture in response to the requirement.

The CDC 7600 (follow-on to the CDC 6600) utilized an improved functional arithmetic unit design which allowed each unit to be entered with new input operands well before previous operands were processed and results obtained. This added pipelining of the execution units along with the instruction stack and control brought additional concurrency to the machine. It was felt that this could be enhanced further by explicit pipeline instructions for the CDC STAR-100.

During 1965 and 1966 Control Data faced significant competitive pressure from IBM in particular and was attempting to expand the role of the CDC 6600 into commercial (non-scientific) markets. Principal competitive factors were the lack of variable length byte oriented instructions, decimal arithmetic and virtual memory. Interal strategies in CDC were pressing for new machines stressing these properties. Thus the STAR-100 project moved to respond. The variable length and 8-bit byte STRING oriented instructions were a somewhat natural fit with the idea of ARRAY instructions utilizing highly parallel pipeline execution. The name STAR-100 was formed from STRING and ARRAY with the objective of 100 million operations per second. STRING operations were assumed to be executed in a separate functional unit and thus were not considered an impediment to the high performance end of the machine.

Moving to the eight-bit world from the octal and six-bit environment was a major learning experience and further complicated by a shift from one's complement to two's complement representation of binary numbers. But the fundamental new area of design was the processing of vectors and arrays through pipeline arithmetic.

The CDC STAR-100 computer was structured around a 4-million to 8-million byte high-bandwidth magnetic core memory. Instructions specify operations on variable length streams of data allowing full use of the memory bandwidth and the arithmetic pipelines [4]. In streaming mode the system has the capability of producing 100 million 32-bit floating point results per second. See Figure 4. Memory has 32 interleaved banks, each bank containing 2048 512-bit words (for the 4-million byte capacity). The memory was an outgrowth of previous extended core storage (ECS) systems built for the CDC 6600. The long word length and relatively slow cycle of 1.28 microseconds with interleaved banks was suitable for streaming use. Working registers in local storage include 256 sixty-four bit general registers. Operands could enter the multiple pipeline arithmetic either from the general registers or from central memory; similarly, results could return to the general registers or central memory.

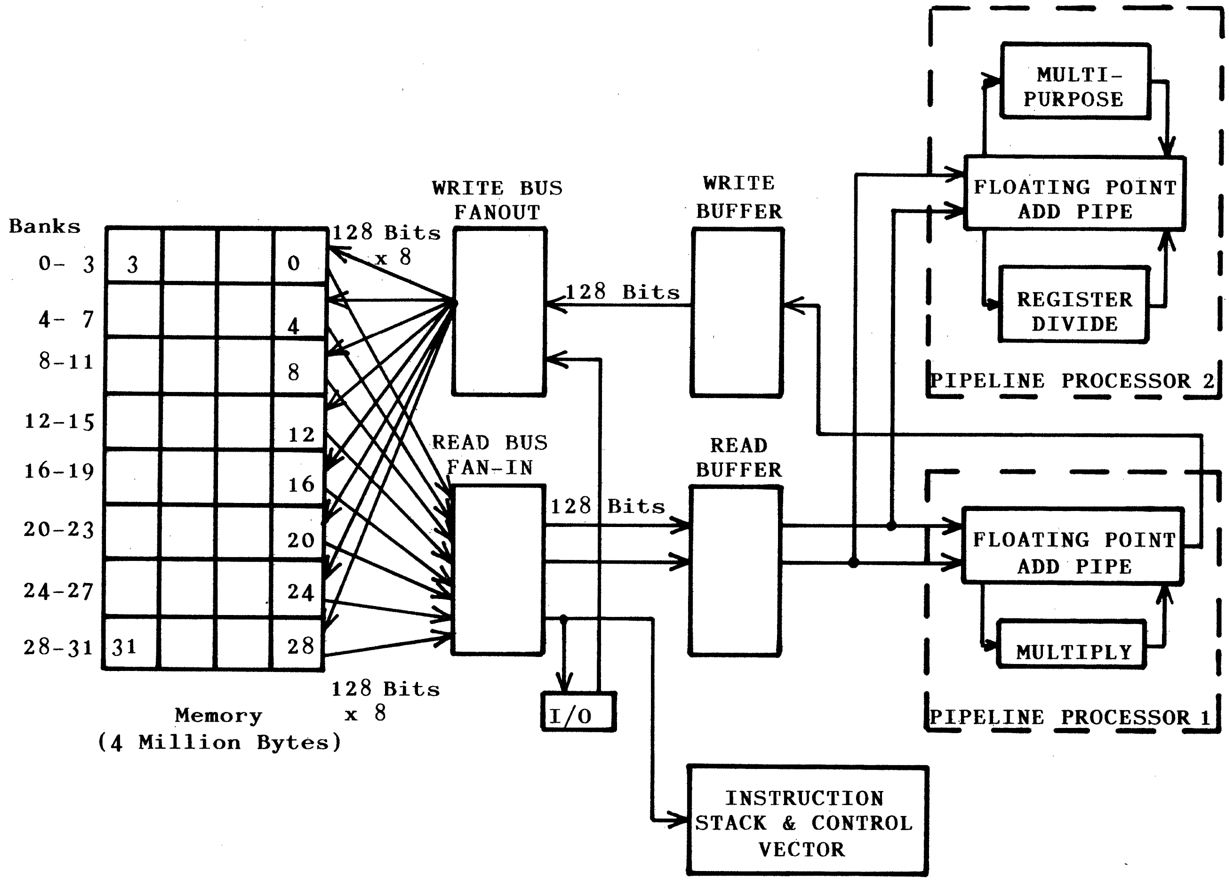


Figure 4. STAR-100 Memory-Pipeline data paths

Vector instructions perform operations on ordered scalars. Such instructions are 64 bits in length and contain the instruction code and three pairs of eight-bit designators. These designators provide the means to support a three address environment. In general two input streams and one result stream are defined. for the two input streams each pair of designators defines working registers (of the 256 general registers) which contain the base address, vector length and an offset to the base address. Length and offset are also held in defined working registers together with the base address of a control vector. The control vector is a bit string in which each unique bit is associated with the storing of each result element in the result stream. A bit in the control vector can prohibit the storing of a result element, thus providing for certain masking and boundary controls.

Thus a single explicit instruction can direct the execution of many floating point operations in a highly organized fashion. In practice, the preparation and synchronization of the three streams was very complicated and required a longer "start up" period than had been expected. As a result, the CDC STAR-100 was more efficient the longer the vectors were. Also with a slow central memory, scalar operations were not competitive although offset somewhat by the high speed general registers.

Both projects suffered delays with the CDC STAR-100 being the longer and several years in duration. Problems with 6600 occurred very early and cost only about a year. Problems with the STAR-100 occurred very late in the planned schedule and resulted in an extended delay. Both projects were exceedingly aggressive and far reaching. The properties of the CDC 6600 have enabled a long lasting product line. The properties of the CDC STAR-100 are only now being exploited in the CYBER-200 machines.

References

[1] Thornton, J.E., "The CDC 6600 Project," Annals of the History of Computing, October 1980, Volume 2, Number 4, pp. 338-348.

[2] Thornton, J.E., "Parallel Operation In the Control Data 6600," October 1964. AFIPS Proceedings FJCC, pt 2 Volume 26, pp. 33-40.

[3] Kuck, D.J., "The Structure of Computers and Computation," 1978, Volume 1, pp. 312, 328.

[4] Hintz, R.G. and Tate, D.P., "Control Data STAR-100 Processor Design," September 1972, COMPCON 72 Proceedings, pp.1-4.

PROGRAMMING DISTRIBUTED APPLICATIONS IN ADA: A FIRST APPROACH[1]

by

Stephen A. Schuman
Massachusetts Computer Associates, Inc.
Wakefield, Mass. 01880

and

Edmund M. Clarke, Jr.
Christos N. Nikolaou
Center for Research in Computing Technology
Harvard University

Abstract -- This paper addresses the problem of programming distributed systems within the framework of the Ada language, which provides primitives for interprocess communication based upon the model of Communicating Sequential Processes. We first discuss our basic assumptions concerning the underlying target configuration, the physical communication medium which is to support that application and pattern of the logical communication within the application proper. We then develop a first approach for constructing such applications using the separate compilation facilities of Ada. Finally, we consider two possible protocols for implementing the requisite distributed interprocess communication, referred to as the Remote Entry Call and the Remote Procedure Call, respectively.

1. Introduction

This paper addresses the problem of programming distributed applications within the framework of the Ada language [3,2,5]. Our ambitions here are confined to outlining a first approach in this area, whence a number of significant issues associated with the construction of such software are, of necessity, deferred. We begin in Section 2 by setting forth the basic assumptions which underly the overall approach described herein. Section 3 is concerned with establishing an appropriate compile-time framework, within which the programming of an application destined for a multi-processor target configuration can be carried out in much the same way as one intended for a uni-processor target. In the final section, we turn to the development of protocols to support the requisite "interprocessor procedure call" capability, so that the applications of interest can then be programmed without further regard to the distributed nature of the underlying target configuration. Two successive versions of such a protocol are defined. These are referred to as the Remote Entry Call and Remote Procedure Call, respectively.

2. Basic Assumptions

This section outlines our basic assumptions concerning the nature of the distributed application systems to be programmed in Ada. Abstractly, we wish to conceive of some given target configuration, onto which a certain application is ultimately to be mapped, as a network of communicating "Ada Virtual Machines" (AVMs). Every such configuration may therefore be characterized in first instance by an undirected graph, as depicted for example in Fig. 2-1:

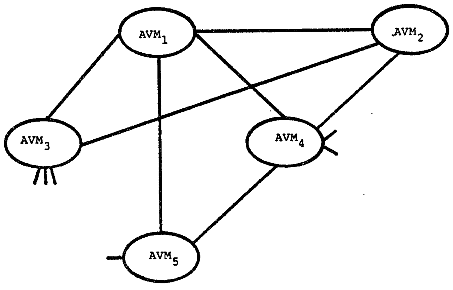


FIGURE 2-1: A network of communicating Ada Virtual Machines.

The individual nodes of a particular network correspond to fully independent (autonomous) processors, each of which is capable of executing a complete Ada program. Accordingly, as Ada Virtual Machine is to be viewed as an idealized *single-processor* environment that directly implements the run-time facilities required to support the semantics of the full Ada language. Thus the concept of an AVM embodies an abstract object machine for which Ada source programs might conventionally be compiled (but disregarding all dependencies upon a specific hardware architecture and/or host operating system); concretely, it may be thought of as providing its own address space, scheduler and real-time clock, together with a certain set of

[1] At Massachusetts Computer Associates, Inc., this research was supported in part by the U.S. Army CORADCOM, through the Scientific Services Program under Delivery Order No. 1704 from Battelle Columbus Laboratories.

At Harvard University, this research was supported in part by NSF Grant MCS-7908365 and by Contract N00039078-G-0020 with the Naval Electronics Systems Command.

0190-3918/81/0000/0038$00.75 © 1981 IEEE

external interrupts, low-level device interfaces, etc. We refer to this environment as a "virtual" (rather than "actual") machine so as to also eliminate considerations arising from the fact that several such machines might sometimes be multiprogrammed on the same physical processor (e.g., in the context of an underlying time-sharing system).

The connecting edges appearing in a given network represent possible paths of *bidirectional* communication between distinct processor nodes. (Non-connecting edges, like those shown in Fig. 2-1, are meant to suggest additional paths of communication, for instance with various devices attached to the individual virtual machines; however, interactions with purely local resources of this sort are of no direct interest here, and so will not be further discussed.) The connectivity of such a network is assumed to be sufficient for supporting the intended pattern of interprocessor communication, meaning that each edge corresponds to a path whereby both the requisite data and any appropriate control signals can be physically transmitted between the two connected nodes; moreover, the bandwidth of these connections is presumed to be adequate for the application at hand.

We shall assume that the target configuration for any specific application is always *statically* defined--i.e., that the number of virtual (and even actual) processors is established once and for all, and that the necessary paths of communication exist from the outset. The primary stipulation which we impose is that all interactions between separate nodes of the network thereby defined must be achieved by explicit communication across these more or less "thin wire" connections. In other words, we preclude interactions based upon the existence of shared memory or any form of centralized control. This implies that the application in question must be formulated from the beginning as a *distributed system*. The issue we wish to address is how one might go about programming such applications in Ada, so as to be able to effectively map those programs onto the given multiprocessor configuration.

Ada provides an adequate basis for programming systems of communicating sequential processes [1],and for supporting synchronous communication between these processes. Once some desired pattern of logical communication has been established (for example, that depicted in Fig. 2-2), there is no particular difficulty involved in formulating the specifications and subsequent definitions for the corresponding caller and server processes (or subsystems). Insofar as the resultant program is destined to be executed on a single processor configuration (as represented by the Ada Virtual Machine considered here), the job is effectively done once all of the separate compilation units comprised by that program have been successfully compiled (since an AVM is assumed to be capable of directly executing any complete Ada program, regardless of its textual decomposition).

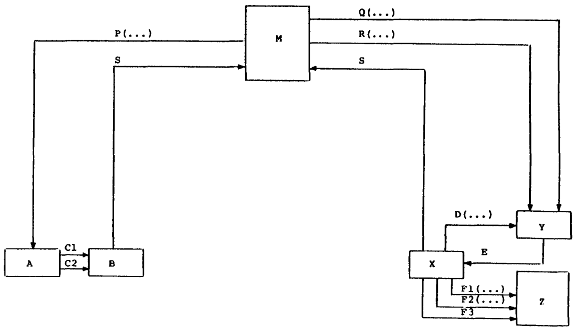


FIGURE 2-2: Example Application, in terms of Communicating Sequential Processes

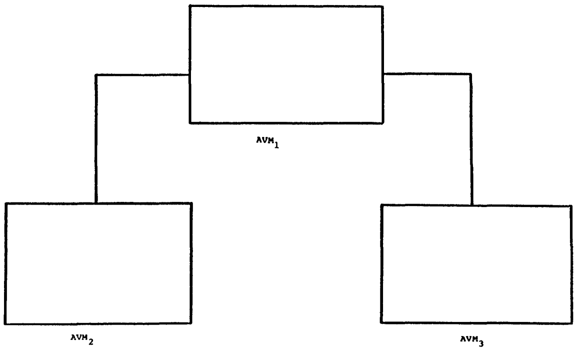


FIGURE 2-3: Example Target Configuration, in terms of Interconnected Ada Virtual Machines

However, when the target configuration is a network of interconnected AVMs (e.g., Fig. 2-3), then it is far less obvious how to proceed. The effect that we should like to achieve is to be able to essentially "superimpose" the intended pattern of communication upon the underlying network (as suggested by Fig. 2-4), thereby preserving the overall logical structure of the application. While the ability to do so presupposes that the application in question was formulated as a distributed system in the first place (i.e., based solely upon communicating sequential processes), it should then be possible to map that structure onto any appropriate target configuration, whether centralized or distributed. This is the premise of the approach outlined in the present paper.

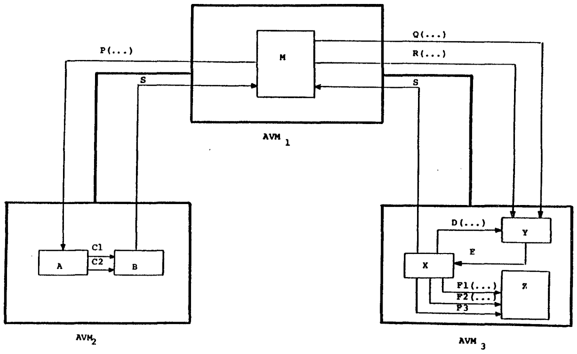


FIGURE 2-4: Superposition of Example Application upon the given Target Configuration

3. Overall Framework

In this section, we shall outline a basic approach to constructing a distributed application, such as that depicted in Fig. 2-4, by making extensive use of the separate compilation facilities in Ada (and also of the related capabilities for generic program units). The framework to be developed here must be regarded as simply a first approach to the problem whence many practical aspects associated with building distributed software will have to be glossed over (or neglected entirely) in the present context. (In particular, we shall be concerned solely with constructing a definition for the steady-state operation of a given application, even though it is well known that the issues involved in startup and shutdown of a distributed system are far more difficult to address.) This approach nonetheless provides a number of important insights into the nature of the problem itself.

The package declaration that follows shows, in skeleton form, an initial specification for the application as a whole:

```
package Config is

   type NODE is (NN$1, NN$2, ..., NN$n);
                             -- Node Names
   type NSET is array (NODE) of BOOLEAN;
                             -- Set of Nodes

   package Node$1 is ... end;
   ...

   package Node$h is

     type OPER is (OP$1, OP$2, ..., OP$k);
     -- Op Codes for Remote Services
     -- other type definitions ...

     Host: constant NODE := NN$h;
     Conn: constant NSET := (...=> True,
                            others => False);
     -- other constant declarations ...

     generic
       Site: in NODE;
     package Service is
       procedure P$1 (...);
       ...
       procedure P$k (...);
     end Service;

   end Node$h;

   ...
   package Node$n is ... end;

end Config;
```

In order to formulate such definitions, we have adopted the (purely lexical) convention of writing names with an embedded dollar sign, so as to be able to refer to unique identifiers as if they were elements of a set distinguished by means of subscripts. For instance, the declaration of the enumeration type NODE is meant to suggest a range of values NN_1, NN_2, ..., NN_n, whereas in practice the individual values would correspond to application-specific mnemonic names (e.g., NN_h might be written as the Ada identifier "FileServer"). Also, P\$1, ..., P\$k denote the particular procedural services which that individual node provides.

This first specification consists primarily of package specifications for the constituent nodes of the overall configuration. The logical interface of each separate node comprises, in addition to various type and constant declarations, the declaration for a *generic package* Service, which will ultimately be instantiated within the definition of other (caller) nodes.

The associated body for the package Config, shown below, serves to establish the overall conventions which are common to all nodes. As such, it is primarily concerned with defining the underlying communications interface, by which information will be physically interchanged between distinct (virtual) machines within the configuration. These conventions are embodied firstly in a series of data type definitions, including:

- XREC, corresponding to a "transaction record" that contains at least an indication of the respective source and destination nodes for each transmission, as well as an encodement of the particular "operation code" for that particular transmission;

- XMIT, corresponding to a complete transmission, as delivered to or received from a local communications interface, which includes both an XREC component and an associated buffer (whereby argument or result data may be forwarded).

Two different types of transmission are distinguished at the communications level, namely Transmit Call (XC) and Transmit Response (XR), and the corresponding subtypes of XMIT are also defined (CALL and RESP, respectively).

Finally, the actual communications interface is specified in the form of two distinct generic packages, ChnDriver and ChnServer. Each of these have a number of generic parameters, in particular, an operation Request and an operation Deliver which will be bound in the context of their subsequent instantiations in order to carry out the necessary acquisition and disposition of transmissions over the underlying medium. This interface is assumed to take full responsibility for setting and using the Orig and Dest Fields of the transaction record part of such transmissions. The details of these interfaces will not be further specified here.

```
with Medium;
package body Config is

   function Card(N:in NSET) return INTEGER range
                  0..NODE'POS(NODE'LAST)+1...;

   subtype OPID is INTEGER range 0.....;
                  -- Max Op Code

   type XREC is record
      Orig, Dest: NODE;
      ...
      Code: OPID;
      ...
   end record;

   type BUFF is ... ;
   type XTYP is (XC, XR);

   type XMIT(T: XTYP) is record
      X: XREC;
      B: BUFF;
   end record;

   subtype CALL is XMIT(XC);
   subtype RESP is XMIT(XR);

   generic
      From, To: in NODE;
      with procedure Request(C: in out CALL);
      with procedure Deliver(R: in RESP);
   package ChnDriver;

   generic
      From: in NSET;
      To  : in NODE;
      with procedure Request(R: in out RESP);
      with procedure Deliver(C: in CALL);
   package ChnServer;

   package body ChnDriver is ... use Medium;
     ... end;
   package body ChnServer is ... use Medium;
     ... end;

   package body Node$1 is separate;
   ...
   package body Node$n is separate;

end Config;
```

We now introduce analogous definitions for each separate node of our distributed configuration (the outline for that representing the Node$h is shown below). In this instance, however, such a step no longer constitutes an "extra" level of abstraction; rather, it is essential -- for this is the first place in which we permit actual instantiations (of code or data), since we have only now reached a level that corresponds to some physical machine environment.

The definition of such a shell serves to establish what might be construed as an "Application Virtual Machine," in terms of which the constituent subsystems of the actual application (e.g., the modules A$1...A$m) may then be programmed without further regard to the distributed nature of the underlying target configuration. This definition serves to provide:

- An indication of the target environment for this particular node (pragma SYSTEM);
- The specification of the application modules to be hosted within this node (the package declarations for A$1...A$m);
- A mapping of the remotely callable services provided by this node onto the operations defined by those modules (e.g., renaming of P$i);
- Definition of *both* sides of the higher-level protocol required to support such remote calls, namely the driver side (the body of the *generic* package Service) and the server side (the body of the non-generic package Support);
- Finally, instantiations of the remote services needed to *implement* the application modules of this node (package Node$u, Node$v, etc.).

```
separate (Config)
package body Node$h is

  pragma SYSTEM(...);

  -- Specify local application modules:

  package A$1 is
     procedure Q$1(...);
     ...
     procedure Q$f(...);
  end A$1
  ...
  package A$m is
     procedure Q$1(...);
     ...
     procedure Q$g(...);
  end A$m;

  -- Local (re)definition of services:
  ...
  procedure P$i(...) renames A$a.Q$b;
  ...

  -- Support services called remotely:

  package Support;
  package body Support is -- Server side of Protocol

     ...
  end Support;

  package body Service is -- Driver side of Protocol

     ...
  end Service;

  -- Provide services needed locally:

  package Node$u is new Config.Node$u.Service
                                (Site => Host);
  ...
  package Node$v is new Config.Node$v.Service
                                (Site => Host);
```

```
   package body A$1 is separate;
   ...
   package body A$m is separate;

end Node$h;
```

Within the framework of this shell, the application modules would again be defined as separately compiled subunits:

```
     separate (Config.Node$h)
     package body A$1 is

        ... Node$u.P$i(...) ...

     end A$1;

     ...

     separate (Config.Node$h)
     package body A$m is

        ... Node$v.P$j(...) ...

     end A$m;
```

The approach outlined above effectively makes use of the Ada "Program Library" to establish the context in which individual components of a distributed application may be defined in terms of a purely procedural interface to services which are nonetheless hosted on different nodes of a distributed target configuration. The possible protocols by which such an "interprocessor procedure call" capability might be realized are the subject of Section 4 of this paper.

It must be pointed out, however, that the usage of the Ada separate compilation facilities described above, while legitimate in every respect, may nonetheless cause a potential problem in the context of overly "naive" implementations of those facilities. Specifically, the issue arises in conjunction with circular dependencies (wherein Node_1 calls Node_2, and so must instantiate its Service package which is defined in the body of Node_2, and vice versa). Whereas this, too, could be "programmed around" (at the cost of considerable effort and obscurity), in this instance it would seem preferable to wait for more mature implementations.

4. Possible Protocols

In this section, we shall be concerned with possible protocols by which the desired interprocessor procedure call capability might be implemented for a particular distributed application. Thus, at this point, we shall elaborate upon actual definitions for the driver side (which serves to map such calls onto the communications interface) and the server side (which acts to carry out such calls on behalf of any remote caller); these implementations correspond to the bodies of the packages Service and Support, respectively, which are defined within the body for the node wherein those remotely callable services are to be hosted.

For purposes of exposition, we shall consider only one instance of such a definition, that associated with the virtual machine Node$h (which makes available the operations P$1...P$k) and, moreover, we shall sketch out the detailed implementation for only one of the operations in question, identified throughout as P$i. This involves no loss of generality, since the structure for all other operations and nodes is essentially the same. Accordingly, the overall goal for the implementations that will be described here is to provide the capability suggested by Fig. 4-1, namely to permit application processes such as A_1, A_2, B...C, residing on separate (virtual) machines, to invoke the operation P_i hosted by Node_h (corresponding to yet another such virtual machine) as though by a simple (local) procedure call.

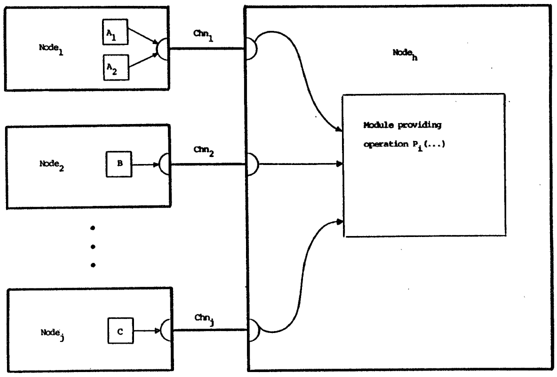


FIGURE 4-1: Overview of the Required Capability, to Support Remote Calls on the operation P_I

To simplify the presentation, we shall assume that the operation of interest has the following specification:

procedure P$i (A1:in TA1;...;Ax:in TAx; R1:out TR1;...;Ry:out TRy); where Aj stands out for the jth input argument (of type TAj) and Rk stands for the kth output result (of type TRk); formal parameters of mode "in out" are thus presumed to have been decomposed into separate input and output objects. We note that some restrictions must be imposed upon the types of parameters in the present context. Specifically, it must be possible to *copy* the associated objects from one machine to another, which apparently precludes the passage of task or "limited private" types (for which assignment is not defined). Similarly, it must be possible to meaningfully *refer* to such objects both locally and remotely, which precludes the passage of access types (except when declared as "private").

In the subsections which follow, we shall develop two alternative definitions for the desired protocol, referred to as the *Remote Entry Call* and the *Remote Procedure Call*, respectively.

In the first (and simpler) version, we impose the property that, from each distinct caller node, there is at most one remote call to any given operation in progress at a time. Such an implementation would be appropriate, for example, in cases where the operations to be invoked are known to be entries (i.e., serviced in a purely sequential fashion), whence there is no advantage to be gained by forwarding more than one potentially concurrent call from some particular node (since these would then have either to be buffered within the communications medium or enqueued by the corresponding server node).

The second version relaxes this restriction, allowing a (bounded) number of calls on the same operation to proceed concurrently from within each separate caller node. This somewhat more complicated strategy might be adopted in situations where there is some optimization to be achieved (on the server side) by recognizing new calls before all previous ones have been completely serviced (as for instance in the context of a disk scheduler).

It must be stressed that there is no *semantic* distinction between these alternative implementation strategies. The choice affects only system throughput and thus the overall performance of the application in question; it should therefore be made on that basis alone.

We shall now proceed to develop Ada definitions for these two alternative protocols, expressed primarily in terms of the synchronous communication primitives embodied in the tasking facilities of that language. Each of the implementations to be described consists of the driver side (the body of the generic package Service, which is to be instantiated within one or more remote caller nodes), and the corresponding server side (the body of the package Support, which resides within the Ada Virtual Machine that hosts the operations in question).

4.1 The Remote Entry Call

As stated above, the first strategy is based on the property that no more than one remote call on each operation is in progress from the same node at any given time, so as to avoid saturation of the communications medium or overloading of the corresponding server node. As such, this property is necessarily established on the driver side of the protocol defined below.

4.1.1. Driver Side. The overall structure and associated data-flow for the driver side are depicted in Fig. 4-2. Calls on the operation P$i, originating from application tasks Ta...Tz are fielded by an Agent which is specific to that operation (AGTi); this latter acts to acquire the input arguments for each individual call (Al...Ax) and to subsequently deliver the corresponding output results (Rl...Ry). These two separate transactions for every operation hosted by Node$_h$ (P$1...P$k) are dispatched via distinct processes, the Driver Call Handler (DCH) and the Driver Response Handler (DRH), which respectively act to forward calls and retrieve responses from the Local Channel Driver (LCD) for Node$_h$. These handlers are formulated as independent (concurrent) processes so that the order in which LCD requests calls or delivers responses will not be unnecessarily constrained by this protocol.

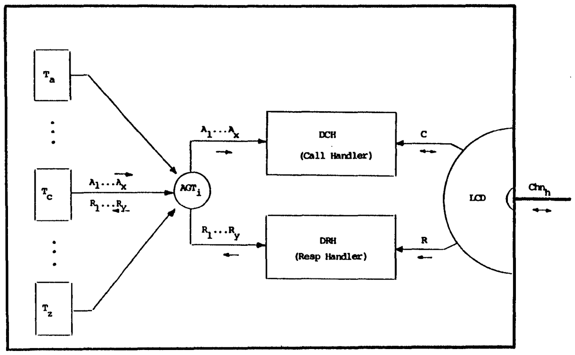


FIGURE 4-2: Overall Structure and Data-Flow on the Driver Side for the Remote Entry Call Protocol.

The outline of (generic) package body for the driver side is shown below:

```
package body Service is
        -- Driver Side, defined in Config.Node$h:

  task DCH is
    entry ReqCall(C: in out CALL);
    entry DC$1(...);
    ...
    entry DC$i(Al: in TAl; ...; Ax: in TAx);
    ...
    entry DC$k(...);
  end;

  task DRH is
    entry DelResp(R: in RESP);
    entry RR$1(...);
    ...
    entry RR$i($1: out Trl; ...; Ry: out TRy);
    ...
    entry RR$k(...);
  end;

  package LCD is new ChnDriver(
    From => Site, To => Host,
    Request => DCH.ReqCall,
    Deliver => DRH.DelResp);

  package D$1 is ... end;
  ...
  package D$i is
    procedure P(Al: in TAl;...;Ax: in TAx;
                   Rl: out TRl;...;Ry: out TRy);
    procedure PutArg(B: in out BUFF;
                   Al: in TAl: ...; Ax: in TAx);
    procedure GetRes(B: in BUFF; Rl: out TRl;
                                 ...; Ry: out TRy);
  end D$i;
  ...
  package D$k is .. end;

  procedure P$1 (...) renames D$1.P;
  ...
  procedure P$k (...) renames D$k.P;

  ... + bodies of DCH, DRH, D$1, ..., D$k

end Service;
```

The handler processes DCH and DRH are directly specified in terms of Ada tasks, with entries to be called by the channel driver and by the agents for the remote operations to be invoked. LCD is obtained by instantiation of the generic definition associated with the overall configuration. For each operation, there is then a corresponding Driver package, D$1...D$k, which provides an operation P to be called by an application process (as P$i) along with operations for moving arguments into and results out of the actual transmission buffers.

The definition of the Driver Call Handler is as follows:

```
task body DCH is
begin
  loop
    accept ReqCall(C: in out CALL) do
      select
        accept DC$1(...) do ... end;
      ...
      or
        accept DC$i(A1:in TA1;...; Ax:in TAx) do
          C.X.Code := OPER'POS(OP$i);
          D$i.PutArg(C.B, A1,..., Ax);
        end DC$i;
      ...
      or
        accept DC$k(...) do ... end;
      end select;
    end ReqCall;
  end loop;
end DCH;
```

Each time the channel driver requests a call (entry ReqCall), DCH makes a (non-deterministic) choice among the Agents waiting to deliver a call for one particular operation (entry DC$i), whereupon it sets the OpCode of the transaction record for that CALL and transfers the arguments into the associated data buffer.

The definition of the Server Response Handler shows the other side of this interface with the Local Channel Driver for Node$_h$:

```
task body DRH is
begin
  loop
    accept DelResp(R: in RESP) do
      case OPER'VAL(R.X.Code) is
        when OP$1 => ...;
        ...
        when OP$i =>
          accept RR$i(R1: out TR1,...,
                                         Ry: out TRy) do
            D$i.GetRes(R.B, R1,..., Ry);
          end RR$i;
        ...
        when OP$k =  ...;
      end case;
    end DelResp;
  end loop;
end DRH;
```

Each time LCD delivers a response (entry DelResp), DRH decodes the Opcode appearing in the transaction record of that RESP and then accepts the pending response request from the agent for that operation (entry RR$i), transferring the corresponding result data.

The outline of the body for a Driver package is shown below:

```
package body D$i is

  task AGT is
    entry Exec(A1: in TA1;...;Ax: in TAx;
                      R1: out TR1;...; Ry: out TRy);
  end;

  procedure P(A1: in TA1;...;Ax: in TAx;
                      R1: out TR1;...; Ry: out TRy)
    renames AGT.Exec;

  procedure PutArg(...) is ... end;
  procedure GetRes(...) is ... end;

  ... + body of AGT

end D$i;
```

The (sole) Agent for the operation P$i is simply defined as a task having an entry Exec (with the same signature), and the operation is renamed to be a call to this entry (which is sufficient to ensure the desired property--that calls from the application tasks of each node will be serviced sequentially). In addition, the low-level operations PutArg and GetRes are defined herein (presumably in terms of representation specifications and/or untyped conversions).

Finally the body of the agent task for P$i is defined as follows:

```
task body AGT is
begin
  loop
    accept Exec(A1:in TA1;...;Ax:in TAx;
                       R1:out TR1;...;Ry:out TRy) do
      DCH.DC$i(A1,..., Ax);
      DRH.RR$i(R1,..., Ry);
    end Exec;
  end loop;
end AGT;
```

For each successive external call to the entry Exec (while the calling process is held in rendezvous), the Agent first delivers the call to DCH and then requests the response from DRH. Because these transactions take place within the rendezvous itself, arguments and results need only be copied once (via the operations PutArg and GetRes) upon actual transmission.

4.1.2. The Server Side. The server side of the Remote Entry Call protocol is essentially symmetric to the driver side. The overall structure and associated data-flow for this side are shown in Fig. 4-3. The Local Channel Server (LCS) forwards incoming calls from connected nodes to the Server Call Handler (SCH), and transmits the corresponding responses as dispatched by the Server Response Handler (SRH). As before, these handlers are formulated as independent processes (so as not to constrain the order of transactions with the underlying communications medium) and play a purely intermediary role. The actual calls to a locally supported operation P$i are performed by one of a

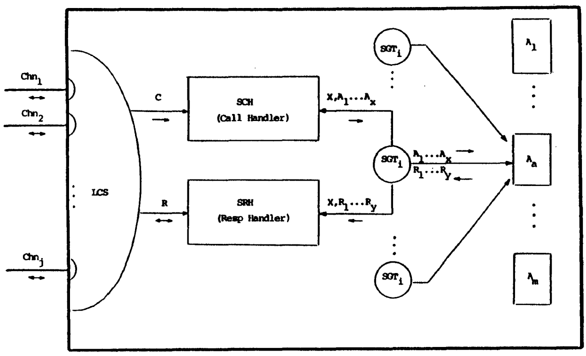


FIGURE 4-3: Overall Structure and Data-Flow on the Server Side for the Remote Entry Call Protocol.

number of Surrogate processes (SGTi), which act as stand-ins for the original calling processes within some other node. Thus, there exist *multiple* surrogates for each remotely callable operation, which serve both to "buffer" incoming calls and outgoing responses (along with their associated transaction records) as well as to invoke the actual operation in question (as provided by one of the application modules A1...Am supported by Node$_h$).

The implementation of the server side for Node$_h$ is defined in the (non-generic) package body Support, shown in outline form below:

```
package body Support is
        -- Server Side, defined in Config.Node$h;

  task SCH is
    entry DelCall(C: in CALL);
    entry RC$1(...);
    ...
    entry RC$i(XR: out XREC; A1: out TA1;...;
                                  Ax: out TAx);
    ...
    entry RC$k(...);
  end;

  task SRH is
    entry ReqResp(R: in out RESP);
    entry DR$1(...);
    ...
    entry DR$i(XR: in XREC; R1: in TR1;...;
                                  Ry: in TRy);
    ...
    entry DR$k(...);
  end;

  package LCS is new ChnServer(
    From => Conn, To => Host,
    Deliver => SCH.DelCall,
    Request => SRH.ReqResp);

  package S$1 is ... end;
  ...
  package S$i is
    procedure GetArg(B: in BUFF; A1: out TA1;...,
                                  Ax: out TAx);
    procedure PutRes(B: in out BUFF;
                     R1: in TR1;...; Ry: in TRy);
  end S$i;
  ...
  package S$k is ... end;

    ... + bodies of SCH, SRH, S$1, ..., S$k

end Support;
```

The handler processes are again directly specified as Ada tasks (SCH and SRH) and the communications interface is obtained by generic instantiation of the definition ChnServer for the overall configuration. As on the driver side, separate Server packages S$1...S$k are introduced here for each individual operation P$1...P$k that can be called remotely.

The definition of the Server Call Handler is as follows:

```
task body SCH is
begin
  loop
    accept DelCall(C: in CALL) do
      case OPER'VAL(C.X.Code) is
        when OP$1 =  ...;
        ...
          accept RC$i(XR:out XREC; A1:out TA1;...;
                                   Ax:out TAx) do
            XR := C.X;
            S$i.PutArg(C.B, A1, ..., Ax);
          end RC$i;
        ...
        when OP$k =  ...;
      end case;
    end DelCall;
  end loop;
end SCH;
```

Upon delivery of a new call from LCS (entry DelCall), SCH switches on the OpCode and accepts a request for a call to the specified operation (entry RC$i) from the next of the (possibly many) Surrogates which are queued up on the corresponding entry. This dispatching consists simply of copying the transaction record contained within this particular CALL and transferring the associated arguments (via the operation PutArg provided by S$i).

The definition of the Server Response Handler is like that of the Call Handler on the driver side:

```
task body SRH is
begin
  loop
    accept ReqResp(R: in out RESP) do
      select
        accept DR$1(...) do...end;
      ...
      or
        accept DR$i(XR: in XREC; R1: in TR1;...;
                                   Ry: in TRy) do
          R.X := XR;
          PutRes(R.B, R1,..., Ry);
        end DR$i;
      ...
      or
        accept DR$k(...) do...end;
      end select;
    end ReqResp;
  end loop;
end SRH;
```

Each time LCS requests a new response (entry ReqResp), SRH makes an arbitrary choice among pending responses ready to be delivered for any operation (entries DR$1...DR$k), whereupon the original transaction record and corresponding output results are copied into the RESP, to be transmitted back to the node from which that particular call originated.

The definition of a Server package S$i has the following form:

```
package body S$i is

  subtype SID is NATURAL range 1..Card(Conn);

  task type SGT;

  ST: array (SID) of SGT; -- surrogate tasks

  procedure GetArg(...) is ... end;
  procedure PutRes(...) is ... end;

  ... + body of SGT

end S$i;
```

The Surrogates for the operation P$i are introduced as an array of tasks, the range of which is set to the cardinality of the incoming connections (which would be the maximum number needed if every connected node did indeed call the operation in question). The operations GetArg and PutRes are presumably the inverses of PutArg and GetRes, which were present on the driver side.

Finally, each individual surrogate for P$i is defined as follows:

```
task body SGT is
  XR: XREC;
  A1: TA1
  ...
  Ax: TAx;
  R1: TR1;
  ...
  Ry: TRy;
begin
  loop
    SCH.RC$i(XR, A1,..., Ax);
    Node$h.P$i(A1,..., Ax, R1,..., Ry);
    SRH.DR$i(XR, R1,..., Ry);
  end loop;
end SGT;
```

In a cyclic fashion they simply request a call from SCH, invoke the local operation provided by Node$_h$, and deliver the corresponding response (along with the original transaction record) to be dispatched by SRH. Once again, because the dispatching is handled within a rendezvous, information is copied directly between the individual Surrogates and an incoming CALL or outgoing RESP.

It should be noted that no special precautions are taken on the server side to ensure the basic property of the Remote Entry Call protocol (at most one call in progress to each operation from any given node); this is solely a concern on the driver side. The servers simply invoke the local operations in question. If these have been specified as entries, then those calls will indeed be serviced sequentially; otherwise they will proceed concurrently.

What is of significance on the server side, however, is the fact that there are exactly as many Surrogates for each operation as there are Agents in total (distributed among the possible caller nodes). This property, referred to as *load balancing*, is fundamental to the solutions developed here, in that it ensures that this protocol does not require any additional storage capacity within the underlying communications medium nor any other form of buffering than that provided by the Surrogates themselves. This same property also guarantees that the communications interface will never be unduly tied up (since there will always be an available Surrogate ready to proceed).

4.2 The Remote Procedure Call

In this section, we develop an alternative to the Remote Entry Call protocol, wherein we allow a (bounded) number of calls to the same operation to be in progress concurrently within a given caller node (while still maintaining the overall load balancing that characterized our first solution). This somewhat more general strategy is described as a modification to the approach developed initially.

The point of departure for this strategy is to slightly extend the initial specification for the application as a whole:

```
package Config is

  type NODE is (NN$1, NN$2, ..., NN$n);
  type NSET is array (NODE) of BOOLEAN;
  subtype CONC is INTEGER range 0.....;
                                 -- Max Concurrency
  package Node$1 is ... end;
  ...

  package Node$h is

    type OPER is (OP$1, OP$2, ..., OP$k);
    type MPLX is array (OPER) of CONC;
    -- other type definitions ...

    Host: constant NODE := NN$h;
    Conn: constant NSET := (... => True, others
                                          => False);
    Load: constant MPLX := ...;
    -- other constant declarations ...

    generic
      Site: in NODE;
      Usag. in MPLX;
    package Service is
      procedure P$1 (...);
      ...
      procedure P$k (...);
    end Service;

  end Node$h;

  ...
  package Node$n is ... end;

end Config;
```

The changes are wholly concerned with this added (potential) concurrency:

- A subtype CONC is introduced, whereby the maximum degree of concurrency anywhere within the system is specified;
- Within the package specifying each Node$_h$, a type MPLX is defined, values of which indicate a degree of concurrency on an operation-by-operation basis;
- A constant load (of type MPLX) is defined for each Node$_h$, whereby the limits on the overall concurrency (from all callers) are established for every such node;
- An additional generic parameter Usag (of type MPLX) is introduced for the Service package, so that the degree of concurrency for individual caller nodes may be set upon subsequent instantiation.

Minor modifications are also introduced into the body of the package Config, wherein the overall communications conventions are established:

```
with Medium;
package body Config is

   subtype OPID is INTEGER range 0.....;
   subtype RCID is CONC range 1..CONC'LAST;

   type XREC is record
     Orig, Dest: NODE;
     ...
     Code: OPID;
     Iden: RCID;
   end record;

   type BUFF is ... ;
   type XTYP is (XC, XR)
   type XMIT(T: XTYP) is record
     X: XREC;
     B: BUFF;
   end record;
   subtype CALL is XMIT(XC);
   subtype RESP is XMIT(XR);

   generic
     From, To: in NODE;
     with procedure Request(C: in out CALL);
     with procedure Deliver(R: in RESP);
   package ChnDriver;

   generic
     From: in NSET;
     To  : in NODE;
     with procedure Request(R: in out RESP);
     with procedure Deliver(C: in CALL);
   package ChnServer;

   package body ChnDriver is ... use Medium; ... end;

   package body ChnServer is ... use Medium; ... end;


   package body Node$1 is separate;
   ...
   package body Node$n is separate;
end Config;
```

The changes are to define an additional subtype RCID, which will serve to identify a particular remote call originating from a given node (since the OpCode alone will no longer be sufficient for this purpose), and to add a new component Iden (of type RCID) to all transaction records.

The only changes within the definitions of the separate nodes of the application would be to suitably set the generic parameter Usag upon each instantiation of the package Service:

```
separate (Config)
package body Node$h is

   pragma SYSTEM(...);

   --- Specify local application modules:

   package A$1 is
      procedure Q$1(...);
      ...
      procedure Q$f(...);
   end A$1
   ...
   package A$m is
      procedure Q$1(...);
      ...
      procedure Q$g(...);
   end A$m;

   -- Local (re)definition of services:
   ...
   procedure P$i(...) renames A$a.Q$b;
   ...

   -- Support services called remotely:

   package Support;
   package body Support is -- Server side of Protocol

     ...
   end Support;

   package body Service is -- Driver side of Protocol

     ...
   end Service;

   -- Provide services needed locally:

   package Node$u is new Config.Node$u.Service
                       (Site => Host, Usag => ...);
   ...
   package Node$v is new Config.Node$v.Service
                       (Site => Host, Usag => ...);

   package body A$1 is separate;
   ...
   package body A$m is separate;
end Node$h;
```

4.2.1. The Driver Side. The changes on the driver side in going from the Remote Entry Call to the Remote Procedure Call are concerned with keeping track of the identity of calls in progress. At the first level, this involves adding and additional ID parameter to the DC$i entries of the Driver Call Handler (DCH), and of introducing a Post Response procedure (PR) to each of the Driver packages D$1...D$k:

```
package body Service is
-- Driver Side, defined in Config.Node$h:

  task DCH is
    entry ReqCall(C: in out CALL);
    entry DC$1(...);
    ...
    entry DC$i(ID: in RCID; Al: in TA1; ...;
                                    Ax: in TAx);
    ...
    entry DC$k(...);
  end;

  task DRH is
    entry DelResp(R: in RESP);
    entry RR$1(...);
    ...
    entry RR$i(R1: out TR1; ...; Ry: out TRy);
    ...
    entry RR$k(...);
  end;

  package LCD is new ChnDriver(
    From => Site, To => Host,
    Request => DCH.ReqCall,
    Deliver => DRH.DelResp);

  package D$1 is ... end;
  ...
  package D$i is
    procedure P(Al: in TA1;...;Ax: in TAx;
                    R1: out TR1;...;Ry: out TRy)
    procedure PutArg(B: in out BUFF;
                    Al: in TA1: ...; Ax: in TAx);
    procedure GetRes(B: in BUFF; R1: out TR1;...;
                                    Ry: out TRy);
    procedure PR(ID: in RCID)
  end D$i;
  ...
  package D$k is .. end;

  procedure P$1 (...) renames D$1.P;
  ...
  procedure P$k (...) renames D$k.P;

  ... + bodies of DCH, DRH, D$1, ..., D$k

end Service;
```

The definition of DCH is then modified to store the identity of each call as part of the transaction record which it forwards:

```
task body DCH is
begin
  loop
    accept ReqCall(C: in out CALL) do
      select
        accept DC$1(...) do ... end;
      ...
      or
        accept DC$i(ID:in RCID; Al:in TA1;...;
                                    Ax:in TAx) do
          C.X.Code := OPER'POS(OP$i);
          C.X.Iden := ID;
          D$i.PutArg(C.B, Al,..., Ax);
        end DC$i;
      ...
      or
        accept DC$k(...) do ... end;
      end select;
    end ReqCall;
  end loop;
end DCH;
```

The corresponding modifications to DRH involve its passing that identity to the appropriate PR procedure prior to accepting a request to dispose of each incoming response:

```
task body DRH is
begin
  loop
    accept DelResp(R: in RESP) do
      case OPER'VAL(R.X.Code) is
        when OP$1 => ...;
        ...
        when OP$i =>
          D$i.PR(R.X.Iden);
          accept RR$i(R1: out TR1,...,
                                    Ry: out TRy) do
            D$i.GetRes(R.B, R1,..., Ry);
          end RR$i;
        ...
        when OP$k => ...;
      end case;
    end DelResp;
  end loop;
end DRH;
```

Within a Driver package D$i, the modifications consist primarily of introducing a multiplicity of Agents for the same operation (whereas there was only one heretofore). As shown on the next page, this is accomplished by defining an array of agent tasks (AT), the range of which is established by the Usag generic parameters. Thus, the index in this array (of type AID) will serve to uniquely identify a particular call-in-progress for the operation P$i. At the same time, additional entries have to be provided for the AGT task: these are Init (whereby an Agent acquires its own identity) and Done (whereby it may be notified that the response for the call it is carrying out has been received). The procedure PR is essentially a call to this latter entry. A further task, the Agent Manager (AM) is now needed to establish the initial correspondence between the original call (from some application process) and the particular agent which will perform that transaction. This correspondence is created by the procedure P, which is called (concurrently) by every application process seeking to invoke the remote operation P$i.

```
package body D$i is

  subtype AID is RCID range 1..Usag(OP$i);

  task type AGT is
    entry Init(A: in AID);
    entry Exec(Al: in TA1;...;Ax: in TAx;
                    R1: out TR1;...; Ry: out TRy);
    entry Done;
  end;

  AT: array(AID) of AGT;
  task AM is
    entry Ready(A: out AID);
    entry Avail(ID: in AID);
  end;

  procedure P(A1: in TA1;...;Ax: in TAx;
                R1: out TR1;...; Ry: out TRy) is
    A: AID;
  begin
    AM.Ready(A);
    AT(A).Exec(A1,...,Ax, R1,...,Ry);
  end;

  procedure PutArg(...) is ... end;
  procedure GetRes(...) is ... end;

  procedure PR(ID: in RCID) is
  begin
    AT(AID;(ID)).Done;
  end;

  ... + bodies of AGT, AM

end D$i;
```

The initialization and actual allocation of agents is handled by the Agent Manager:

```
task body AM is
begin
  for A in AID loop
    AT(A).Init(A);
  end loop;
-- main cycle:
  loop
    accept Ready(A: out AID) do
      accept Avail(ID: in AID) do
        A := ID;
      end;
    end;
  end loop;
end AM;
```

Each of the agent tasks of the array AT is then defined as follows:

```
task body AGT is
  ID: AID;
begin
  accept Init (A: in AID) do
    ID := A;
  end;
-- main cycle:
  loop
    AM.Avail(ID);
    accept Exec(A1:in TA1;...;Ax:in TAx;
                  R1:out TR1;...;Ry:out TRy) do
      DCH.DC$i(ID, A1,..., Ax);
      accept Done;
      DRH.RR$i(R1,..., Ry);
    end Exec;
  end loop;
end AGT;
```

After initialization an Agent enters its main cycle, wherein it first makes itself available to AM prior to accepting the resultant call via its entry Exec. Within the corresponding rendezvous, it delivers its own identity to SCH along with the arguments for the call in progress, it then awaits notification (via the entry Done) that the response for that particular call has been received before proceeding to request the results on behalf of the original caller.

4.2.2. The Server Side. In passing from the Remote Entry Call to the Remote Procedure Call protocol, essentially no modifications are required on the server side (since this latter already provided for some degree of concurrency, insofar as it had to handle incoming calls from more than one caller node). The only provision that must be made is to possibly increase the number of Surrogates for each operation P$i, which would be specified within the corresponding Server package S$i as follows:

```
    subtype SID is CONC range 1..Load (OP$i);
```

thereby fixing the number of elements in the array of surrogate tasks. This will presumably preserve the overall load balancing (number of Surrogates = total number of Agents, for each operation Pi) upon which both of the protocols developed in this section have been based.

6. Conclusion

This paper has addressed the problem of programming distributed applications in Ada and outlined a first approach in this area. Essentially two aspects have been considered: the provision of a suitable compile-time framework for defining such applications in the first place (which was achieved by exploiting the possibilities of the separate compilation facilities in Ada); and the support of a suitable "interprocessor procedure call" protocol, whereby the application itself could then be programmed without further regard to the distributed nature of the underlying hardware configuration (a capability which was defined in terms of the multi-tasking facilities of Ada). Several such protocols were in fact developed here, beginning with the relatively simple Remote Entry Call, which was then extended to yield the Remote Procedure Call strategy. In [4] we further extended this approach so as to take into account the unreliability of the transmission medium in question, while still assuming that the nodes within the overall configuration were perfectly reliable.

References

[1] Hoare, C.A.R., Communicating Sequential Processes, CACM, August 1978, Vol. 21, 8.

[2] Ichbiah, J.D. *et al.*, Rationale for the Design of the ADA Programming Language, SIGPLAN Notices, June 1979, Vol. 14, 6, B.

[3] -- Reference Manual for the ADA programming language, U.S. Dept. of Defense, July 1980.

[4] Schuman, S.A., Clarke, E.M., Nikolaou, C.N., Programming Distributed Applications in ADA: A First Approach, Massachusetts Computer Associates, Inc., CADD-8103-3102.

[5] Schuman, S.A., Tutorial on ADA Tasking, Vol.I: Basic Interprocess Communication, Massachusetts Computer Associates, Inc., CADD-8103-3101.

SALAD - A DISTRIBUTED COMPILER FOR DISTRIBUTED SYSTEMS*

T. Christopher, O. El-Dessouki, M. Evens,
H. Harr, H. Klawans, P. Krystosek, R. Mirchandani, Y. Tarhan
Computer Science Department, Illinois Institute of Technology
Chicago, Illinois 60616

Abstract -- A procedural single assignment language, SALAD, is presented, and its implementation on a distributed, multicomputer system is discussed. A procedure is executed not by a single task, but by a collection of cooperating tasks (threads of control) that share the procedure's activation record and synchronize with event variables and semaphores. Procedure calls and returns are handled with message passing, permitting the called procedures to be executed on remote machines. SALAD includes state-maintaining objects, e.g. queues, which violate the spirit of single assignment languages, but provide more usual multitasking facilities. Not only is a SALAD program to be able to run distributed over a computer network, but the compiler itself is to be able to execute on such a system. The compiler tries to optimize both the code for SALAD procedures and the distribution of the code over the network.

Introduction

The proliferation of distributed computing systems has intensified the software crisis. The development of distributed hardware has far outstripped the development of the necessary software. We are primarily concerned with systems composed of microcomputers connected in a network. There is a strong need for compilers which can both run on a distributed system and generate code for that same system. The development of compilers for such systems presents two novel problems: distributed compiler organization and distributed code generation. Since different components of the compiler will run on separate microcomputers it must be organized into small modules which can function independently and exchange information only by messages. This compiler must also be capable of generating distributed code. First it must partition the code into clusters of tasks small enough to fit on the separate computers of the distributed system. Then run time routines are added to control the run time system, to handle error conditions, and to make each cluster capable of standing alone. Finally, the compiler inserts message-passing primitives in each cluster to provide the exchange of values between the clusters at run time. The problem of automatic partitioning is a central issue in the design of these new compilers, but the problems of partitioning conventional general-purpose programming languages are tremendously complex. Single-assignment languages (languages in which no variable is assigned a value more than once) seem to be much easier to handle. This paper describes the design and the implementation of a simple but nontrivial single assignment language (SALAD). The compiler is organized as a pipeline of small modules each of which can reside on a separate microcomputer; it includes a crude partitioning module which divides the intermediate code into self-contained clusters which are then converted into separate load modules for separate computers.

There is a strong relationship between our work and current work on dataflow computers [1, 2]. In fact, we are motivated by the belief that it is possible to obtain the advantages of data-flow architecture without the expense of specialized hardware. There are important differences, however, in our methods. We have not simply programmed microcomputers to behave like data-flow computer components. Instead we have encoded data-flow operations as multiple communicating processes.

Our model of a distributed computing system is a collection of many independent computers with no shared memory, so that all communication is by means of messages only. Why assume that there is no shared memory? While systems exist with multiple processors accessing a common memory, systems without shared memory are easier and cheaper to build, since they can be produced by adding communication channels to existing machines. When those with shared memory systems want to link them together, they will be subject to these constraints as well. Furthermore, compilers designed for systems without common memory can be made to run on shared memory systems, but the reverse is not true.

Our interest in the twin problems of distributed compiler organization and the generation of distributed code began with the TECHNEC project, a very successful project which was funded by the National Science Foundation (NSF-MCS76-01310). TECHNEC, the Illinois Institute of Technology Network Computer, is a ring network of LSI/11's [3, 4]. It was designed to support Greene's experiments in heuristic control [5]. The first step was a distributed operating system [6, 7]. We have also designed and implemented a demon language for TECHNEC, but the current implementation is not truly a

*This research was supported by the National Science Foundation under grant MCS-80-04114.

0190-3918/81/0000/0050$00.75 © 1981 IEEE

distributed compiler, it is a cross-compiler running on the PRIME 400 but generating code for TECHNEC. Demons do, however, present particularly exciting problems in the design of the run time environment.

Our first true distributed compiler was the DYNAMO compiler [8] and [9]. This language was chosen because we needed a continuous simulation language for work on robotics and we were fascinated with the challenge of simulating parallel processes on a network of parallel processors. This compiler was designed from the beginning to run on our distributed system and to generate automatically partitioned code for that system.

We have experimented with four different partitioning algorithms for this compiler [10, 11]. DYNAMO is a nonprocedural language with no explicit control structures; this makes it relatively easy to partition. It can even be viewed as a single assignment language, although it does not resemble pure LISP and those applicative languages which are usually called single assignment languages [12, 13]. We felt that the next step toward our goal of eventually developing mechanisms for conventional programming languages ought to be a compiler for a language that combines the constraints of a single assignment language with procedures, explicit control structures, and at least some multiple data structures. Since we did not know of such a language we decided to design one ourselves; the result is SALAD.

Description of the Language SALAD

A program consists of a collection of procedures. (See Figure 1 for grammar.) A procedure declaration consists of a header line, zero or more declaration lines, one or more command lines, and an END line. The procedure header is of the form

PROC outputs = procedure-name inputs

where either outputs or inputs may be a single identifier or a list of identifiers in parentheses.

The declarations in SALAD are optional. If an identifier is not declared, it may be of any type, and its type may be different on different executions of the procedure. If a name is declared, the code will check that the type of the value assigned to it is correct at run time. The form of a declaration is an identifier or a list of identifiers in parentheses followed by a colon followed by the name of the type they are being declared to be.

There are three primitive types in SALAD: integer, real and Boolean. There are two pure structured types: strings and tuples. A string is a sequence of characters. Once created, it cannot be modified. There is no theoretical limit on the length of a character string.

```
proc ::= prochd declare*  command+  END newline
prochd ::= PROC  lhs = procid lhs newline
declare ::= lhs  : type newline
type is one of INTEGER, REAL, BOOLEAN, TUPLE,
               STRING, FILE, QUEUE, ANY
command ::= lhs = rhs newline
lhs ::= id
lhs ::= ( idlist )
idlist  ::= id
idlist  ::= id , idlist
rhs ::= e3
rhs ::= IF id THEN e3 ELSE e3
e3 ::= e2
e3 ::= op e2
e2 ::= e1
e2 ::= ( e1list )
e2 ::= ( )
e1list ::= e1
e1list ::= e1 , e1list
e1 ::= id
e1 ::= constant
```

Figure 1. SALAD Grammar.

A tuple is a sequence of values. Each value in a tuple may be of any type. As with strings, once created, a tuple cannot be modified. Tuples play a central role in the SALAD language.

There are two kinds of executable statements in SALAD: simple assignments and conditional assignments. Simple assignments have the form

lhs = rhs

The left hand side, lhs, can be either a single identifier or a list of identifiers in parentheses. If the lhs is a single identifier, then the value produced by the rhs may be of any type. If, however, the lhs is of the form (id1, id2, ..., idn) then the value of the rhs must be a tuple of length n. Each element of the tuple is assigned to the corresponding identifier in the lhs.

If the rhs is a single identifier, then its value is used. If the rhs is a list of identifiers (id1, id2, ..., idm), then the value of the rhs is a tuple of length m with the value of identifier idj in position j.

The right hand side could also be an operator or function applied to either a single identifier or a list of identifiers in parentheses. The single identifier form, F A, causes the function F to be applied to A's value. The form F (A1, A2, ..., An) causes function F to be applied to a tuple with the values of the identifiers A1 through An.

A conditional assignment has the form

```
lhs = IF b THEN rhs1 ELSE rhs2
```

where b is an identifier that will have a Boolean value at run time, and lhs and rhs1 and rhs2 have the forms discussed immediately above for simple assignment statements.

```
a = IF B THEN c ELSE d
```

requires B be a Boolean-valued identifier. If B is true, the statement behaves as if it were

```
a = c
```

If, however, B is false, then the statement behaves as if it were written

```
a = d
```

The simple and conditional assignment statements are required to obey the single assignment nature of the language. An identifier may appear only once in a lhs within a procedure. An identifier in the inputs section of a procedure header may not appear in the lhs of any assignment in the procedure.

Like conditional assignments, procedure calls may be defined by substitution rules. The definition of procedure calls in SALAD is very similar to the copy rule for Algol 60 procedures. Given the procedure definition

```
PROC plhs = procid prhs
body
END
```

and the call of the procedure

```
clhs = procid crhs
```

the call behaves as if it had been written

```
prhs' = crhs
body'
clhs = plhs'
```

where the prime (') indicates that all the identifiers are renamed uniquely to avoid conflict with the identifiers in use at the place of call.

There are two other data types in SALAD that have not been mentioned before. They violate the spirit of single assignment languages in that they maintain an internal state, can be modified, and are not pure values.

One of the two types is FILE. Since files encapsulate the interface to the outside, state-maintaining world, they must be forgiven for being that way themselves. The other type is QUEUE. Queues are the objects used for synchronization. Any kind of object may be added to a queue. An attempt to remove an object from an empty queue will cause a delay until an object becomes present. There is no such thing as a full queue. Queues are generally handled in FIFO order, but simultaneous attempts to add or remove items from a queue will be serviced in an unspecified order.

The operations on queues are as follows:

```
q = QUEUE ( )
q1 = PUT(q,val)
(q2, val1) = GET q1
```

The operation q = QUEUE ( ) creates a new queue object and return a pointer to it in q.
The operation q1 = PUT(q, val) puts the value of val in the queue pointed to by q and returns a new pointer to q in q1.
The operation (q2, val1) = GET q1 removes an item from the queue pointed to by q1 and returns that value as val1. It also returns another pointer to the queue in q2.
The reason for returning new pointers to a queue is to permit sequencing of queue operations in the calling program. For example,

```
q = QUEUE ( )
q1 = PUT (q,a)
q2 = PUT (q1,b)
(q3, c) = GET q2
(q4, d) = GET q3
```

will accomplish much the same as

```
q = QUEUE ( )
(q1, q2, q3, q4) = (q, q, q, q)
(c, d) = (a, b)
```

However,

```
q = QUEUE ( )
q1 = PUT (q,a)
q2 = PUT(q,b)
(q3, c) = GET q
(q4, d) = GET q
```

will not necessarily accomplish the same thing. Sometimes it will behave as the example above; sometimes it will do the assignment

```
(c,d)=(b,a)
```

Calls and Messages

Since the network hardware we envision has no shared memory, the implementation must use message passing to pass data and coordinate the execution. Procedures are called by sending a call message. The results of a procedure call are returned in a return message. If a procedure needs to examine a tuple located on another machine, it sends a message requesting a copy of the tuple. Operations on state-maintaining objects, files and queues, are handled via a message to the computer where the object is located.

Node Structure

The software structure on each computer consists of the programs for the procedures located on that computer; tables used by the system; input queues for receipt of messages from the computers connected to this one; output queues to the connected computers; an "available operations queue" containing call messages for procedures that could be involked here; and a heap, or dynamic storage area, that contains tuples, strings, queues, and local storage for active procedures.

Activation Records and Threads of Control

An "activation record" contains the local storage for a procedure. In a conventional language, a single process, or task, would have a stack of activation records. Every procedure call would push an activation record on the stack. Every return would pop one off. Our implementation does just the reverse. There are not multiple activation records per task. There are multiple tasks, which we call threads, per activation record.

Each thread is associated with a thread control block which contains only 1) a pointer to the activation record the thread is associated with, 2) the address of the next instruction the thread is to execute, and 3) a link field so the thread control block can be placed on queues.

The system in each separate computer in the network maintains a run queue of threads. A piece of code, the dispatcher, removes the first thread control block from the run queue, loads a register with the activation record pointer, and jumps to the next instruction the thread is to execute.

A simple form of coordination between threads uses "event variables." An event variable is initialized to require a particular number of "signals" before the event occurs. Only a single thread may wait on an event variable. If the event has not yet occurred, the waiting thread is suspended until it does occur. If the event has already occurred, the thread continues executing. Another thread may signal the event variable. If the signal is the last one required before the event occurs, and another thread is waiting for the event, the waiting thread is linked on the run queue.

We have implemented the synchronization primitives in PDP-11 assembly language. To wait on an event variable requires at most five instructions, whether or not the thread executing the wait must be suspended. To signal an event variable without waking up a thread requires only two instructions. If a signal wakes up a thread, that thread must be linked on the run queue. Linking a thread on the run queue requires less than a dozen instructions.

The system also provides semaphores, which in addition to the usual synchronization and mutual exclusion functions, are used to permit control of the degree of concurrency at run time. This is mentioned again below in the section on optimizations the compiler can perform.
See Figure 2.

```
THREAD CONTROL BLOCK
LNK : link field
PC : program counter field
FP : frame pointer field

EVENT VARIABLE
CNT : count field
THRP : thread pointer (to THCB of Waiting Thread)

READY QUEUE
RQLCK : lock byte on ready queue (init 1)
RQ1ST : pointer to head of ready queue
RQLST : pointer to end of ready queue

READY LIST MANIPULATION
dispatch:  -- Just a normal label
 while RQ1ST = NULL do diddle
 seize_rq
 if RQ1ST = NULL then
  { release_rq
    goto dispatch}
 else
  { T := RQ1ST
    RQ1ST := RQ1ST@.LNK
    release_rq
    FP_REG := T@.FP
    JUMP T@.PC@}

procedure  ready1 (t) =
  { t@.LNK := NULL
   seize_rq
   if RQ1ST = NULL then
    { RQ1ST := t
     RQLST :=t}
  else
   { RQLST@.LNK := t
    RQLST := t}
  release_rq}
 return
end ready1

SIGNAL OPERATION ON EVENT VARIABLE
signalevent s
-- is translated into
   decr s.CNT
   if zero then
    ready1(s.THRP)

WAIT OPERATION ON EVENT VARIABLE s BY THREAD t
waitevent s
-- by a thread with thread with
--   thread control block t
-- is translated into
t.PC := @L
s.THRP :=@t
decr s.CNT
if positive then
  goto dispatch
L:

INITIALIZE EVENT VARIABLE
 initevent s,c
-- is translated into
s.CNT := c+1
```

Figure 2. Threads and Events

Distributed Garbage Collection

Tuples, strings, queues and files are dynamically created during the course of the program. Their bodies occupy memory on the heap on the computer they were created on. When they are no longer needed, the storage they occupy must be reclaimed for other uses. Tuples and strings cannot be created with cycles of containment, e.g. if tuple A is created with tuple B as a component, then since B was created first, it cannot contain A as a component. Moreover, B cannot be modified to contain A. At most, a copy, C, of B can be created with a component changed to be A.

Thus, the containment graphs for tuples are directed, acyclic graphs, and reference count storage reclamation is adequate. The state-maintaining objects, files and queues, might cause problems.

Files do not, in fact, cause problems since neither files nor queues nor tuples containing files or queues may be written into them. Queues may cause problems; a queue can be placed into itself, e.g.:

```
q = QUEUE ( )
q1 = PUT (q,q)
```

It is, therefore, possible to creat circularly linked, inaccessible structures which will not be reclaimed with a reference count scheme.

Such structures are likely to be rare. So we are considering implementing a reference count storage management algorithm. We do have, in addition, a full garbage collection algorithm that will mark all accessible structures and reclaim those that are inaccessible. It works in two phases: 1) it marks all accessible structures by having each computer mark those that are accessible locally and send messages to the computers containing those that are remote; after all computers have finished the mark phase. 2) it has each computer reclaim the unused storage in its own heap.

We have no plans to make the garbage collection algorithm run concurrently with normal processing. There are thoughts that a single computer can do some local garbage collection independently of the others. Objects on the heap can be marked when pointers to them are sent to other computers in messages. Unmarked objects are only pointed to locally, if at all, and can be collected by the computer on which they are contained.

Structure of the Compiler

The compiler for SALAD is composed of four sections. The first section translates from the source language into intermediate code. The second section optimizes the intermediate code and translates procedures into parallel cooperating threads of control. The third section allocates the procedures to separate computers. The fourth section converts into assembly-like code which it optimizes and assembles. Each section is composed of several phases which can be run as separate passes or, in some cases, as a pipeline. Code optimization and allocation are described in more detail below.

Translating Procedures into Parallel Threads

When control enters a procedure it is executing a single, main thread associated with the procedure. The main thread creates the other threads in the procedure to execute concurrently with it. See Figure 3a for a collection of procedures and Figure 3b for an example of the code that could be generated for that collection.

Figure 3a below:

```
PROC C = F N
C = F1 (1,1,N)
END

PROC D = F1(I,J,M)
T1 = +(I,J)
B1 = GT(T1,M)
D = IF B1 THEN I ELSE F2(I,J,M)
END

PROC E = F2(X,Y,Z)
T2 = *(Y,2)
T3 = +(X,Y)
T4 = F1(X,T2,Z)
T5 = F1(T3,T2,Z)
E= *(T4,T5)
END
```

Figure 3a. SALAD code for factorial Function F.

The compiler must partition the operations in a procedure into a collection of threads. There are two main rules for placing operations into threads: 1) Two operations may be placed in the same thread only if one is dependent on data from the other; 2) The operations must be placed in the thread in the order they must be executed. The first rule is to prevent one operation that could be executed from being delayed waiting for completion of an operation that does not have to precede it. The second rule is obvious.

If an operation, A, in one thread requires as input a name computed by a operation, B, in another thread, the first thread must wait on a event variable, EA, before executing A, and the other thread must signal EA after finishing operation B.

The compiler performs the following optimizations on threads where applicable:

1) Recursion removal -- a tail-end recursive call to the procedure that includes the call can sometimes be replaced with an assignment to the input parameters and a jump back to the beginning.

2) Code incorporation -- a procedure called in only one place may be incorporated into the place of the call. Recursion removal, by eliminating a place of call may make more code incorporation possible, which may in turn permit further recursion removal. Also, small, non-recur-

sive procedures may be incorporated into several places of call.

3) Within the constraints mentioned above about the placement of operations in threads of control, the compiler tries to minimize the number of threads generated for a procedure. The constraint that two operations may be placed in the same thread only if there is a data dependency between them makes the minimization of the number of the threads an NP-hard problem: it can be shown equivalent to graph coloring.

4) It will often be possible to eliminate some event variables and signals. In particular, an operation A that defines a value used by operation B need not signal an event for B if there is an operation C that uses a value supplied by A and supplies data to B.

5) For those procedures where it is permissible, the compiler will generate code that will choose at run time whether to execute only one operation in the procedure at a time or to execute with as much concurrency as possible. We hope this code will keep the system from becoming swamped by concurrent operations. The trick is to protect the activation record with a semaphore and let the threads compete for it. See Figure 4 for a picture of the desired behaviour.

6) "Sending off" tail-end calls -- Sometimes it is possible for a procedure call as the last operation of a procedure A to tell the called procedure to return its values to the caller of A, not to A itself. After sending off this call, A's activation record may be deleted.

The Partitioning Module

The purpose of the partitioning module is to specify how the code should be partitioned and assigned to the different computers of the network in order to achieve load balancing, while minimizing the communication overhead. The output from the partitioning module includes not only code clusters but enough information about runtime data flow so that the code generation routines can produce runtime modules capable of standing alone on separate computers and communicating via messages.

The basic requirements for partitioning module design are considered to be:

1. The module can be incorporated as an integral part of a pipelined compiler i.e. it should accept information concerning source program in the form of a stream of messages each of them containing information regarding one source statement. The messages arrive one at a time. The format of the messages and their contents should be compatible with the output of the front end of the compiler. The last stage of the partitioning module should produce groups of statements of intermediate language (partially compiled code) compatible with the input language of the back end stage(s) of the compiler.

2. The partitioning module can run on the same network computer for which the distributed compiler is designed. So the same limitations on the size of every compiler phase apply to the partitioning module also. However, the partitioning module may consist of a number of phases distributed over the network and cooperating together to perform the function of that module. In this case, the general requirements for distributed software apply also to the distributed partitioning e.g. minimizing the communication overhead and balancing the network load.

Figure 3b below:

```
; code for F
Fmain:    send_off_call F1(1,1,N)
          terminate

:code for F1's main thread
F1main:   T1 = +(I,J)
          B1 = GT(T1,M)
          if not B1 goto L1
          return I
L1:       (X,Y,Z) = (I,J,M)
;code for F2's main thread, incorporated into F1
F2main:   Choose_concurrency CS
             ;initialize semaphore CS
             ; to 1 or infinity
          initevent e1,1
          initevent e2,1
          fork  F2T2, F2ThCB2
          waitsema  CS
          T2 = *(Y,2)
          signalevent e1
          T4 = F1(X,T2,Z)
          signalsema CS
          waitevent e2
          waitsema CS
          E =*(T4,T5)
          return E
;code for F2's other thread, T2
F2T2:     waitsema CS
          T3 = +(X,Y)
          signalsema CS
          waitevent e1
          waitsema CS
          T5 = F1(T3,T2,Z)
          signalevent e2
          signalsema CS
          die
```

Figure 3b. Translation of code for factorial into optimized pseudo-code.

In our design, the partitioning module (PM) is composed of three phases:

<u>Phase I</u>: Data structure builder

<u>Phase II</u>: Partitioner

<u>Phase III</u>: Allocator

The data structure building phase serves as a functioning part of the front end pipe during compile time, i.e. it processes one statement at a time. It builds a standard representation of programs using such things as data dependency among statements. The kind of data structure built during this phase will be discussed in detail in the following paragraphs. Transforming known programming constructs to this standard representation is a major part of this research and will be discussed in detail in the next sections. The partitioner and the allocator deal with the representation of the whole program in two separate passes. They must collect global information about dependency among statements and interactions among various parts of the program in order to segment that program and insert communication primitives in it. For example, when a program is partitioned into several nodes of the system, a value may be needed in a node other than the one in which it is calculated. The allocator must detect such a situation and provide for the sending and receiving of the required value. Since a node will probably be receiving values for many variables, the name of the variable must be included with the value so that the receiving node can identify it. The data structure builder stores the program representation on a file for the partitioner to operate on it in a new pass. The graphs which are used as standard representation for programs are described elsewhere [14, 15].

The partitioning problem can be divided into two basic strategies: verticle vs. horizontal partitioning. Imagine you have program listing and you draw horizontal lines on it. All the code between a pair of lines is placed in the same computer. This we call horizontal partitioning.

The easiest place to draw these lines is at subroutine boundaries. An entire subroutine is placed in a single computer. A subroutine is called by the arrival of a message providing its parameters, and it sends back a message when it's done. Since we assume parallelism is provided, or at least permitted, by the language being implemented, there may be several calls to the subroutine concurrently. Thus the activation record for the subroutine is allocated on a heap, rather than a stack, because the calls will not obey a strict LIFO discipline.

If the horizontal lines cut through the midst of subroutines, then when the flow of control reaches such a cut the activation record must be sent to the computer containing the next section of code. Note that addresses must have a computer name associated with them, and fetches or stores of anything other than components of the activation record require (potentially) message passing. Note also that an activation record cannot be sent to another computer while there is a reference to one of its components outstanding.

As an alternative to horizontal partitioning, imagine you take a program listing and draw vertical lines on it making several columns beside the program. Each column represents a different computer. Beside the statements you make check marks in one or more of the columns. A check mark in a column beside a statement indicates that the statement is to be placed in the computer the column represents. The statements of the program have been partitioned among the several computers so that the computers work in parallel on the program, sending messages to each other when a value is computed in one that is needed in the other. Some statements may be represented in all of the computers, e.g. loop control. Each computer contains a part of the activation record of the routine being executed. We call this vertical partitioning.

The partitioning heuristics developed for the DYNAMO compiler assigned each statement to precisely one computer. We noticed that in designing the DYNAMO compiler as a large distributed program we found that the high communication overhead made it more efficient to repeat some portions of the code in several different computers.

For the SALAD compiler we have developed a simple divide-and conquer style algorithm, where a procedure divides a problem into several smaller problems and calls itself recursively and in parallel for each. To get any speed-up from this, copies of the procedure must be located in several computers.

Our past work on partitioning has assumed that all partitioning should be done at compile time. Now that we have copies of some procedures in several computers, it seems that we may be able to balance the load better by delaying until run time the decision of where to execute a particular procedure call.

Future Plans

The obvious next step is further experiments with the partitioning of SALAD. We would like to write at least some part of the SALAD compiler in SALAD and compare the compiler output with what we have done by hand. In the long run more theoretical work on the partitioning is certainly necessary. In the next year we hope to design a distributed partitioning compiler for a subset of Pascal.

References

[1] Arvind, "A Dataflow Architecture with Tagged Tokens," Laboratory for Computer Science; MIT, June, 1980.

[2] J. Dennis and D. Misunas, "A Preliminary Architecture for a Basic Data-flow Processor," Proc. 2nd Annual Symposium on Computer Architecture, (December, 1974), pp. 126-132.

[3] W. Huen, P. Greene, R. Hochsprung, and O. El-Dessouki, "A Network Computer for Distributed Processing, COMPCON, (Fall, 1977), pp. 326-330.

[4] W. Huen, P. Greene, R. Hochsprung, and O. El-Dessouki, "TECHNEC, a Network Computer for Distributed Task Control," Proceedings of the First Rocky Mountain Symposium on Microcomputers: Systems, Software, and Architecture. Fort Collins, Colorado. 1977.

[5] P. Greene, "Strategies for Heterarchical Control," Computer Science Dept., Illinois Institute of Technology, Chicago, 1978.

[6] T. Christopher, O. El-Dessouki, M. Evens, P. Greene, A. Hazra, W. Huen, A. Rastogi, R. Robinson, W. Wojciecowski, "Uniprogramming a network Computer," Proceedings Eighth International Conference on Parallel Processing, (August, 1978), pp. 132-138.

[7] T. Christopher, "The Operating System for TECHNEC", COMPSAC, November 1979.

[8] W. Huen, O. El-Dessouki, E. Huske, and M. Evens, "A Pipelined DYNAMO Compiler, "Proceedings of the Seventh International Conference on Parallel Computing. Traverse City, Michigan. 1977.

[9] M. Evens, E. Huske, J. Pomes, O. El-Dessouki, C. Gerlach, M. Samanta, W. Huen, "Synchronization Issues in Network Compilers," Proc. 2nd Annual Rocky Mountain Symposium on Microcomputers, Fort Collins, Colorado, (August, 1978), pp. 358-397.

[10] O. El-Dessouki, and W. Huen, 1977. "Automatic Partitioning for a Network Computer," Technical Report 77-6, Computer Science Department, Illinois Institute of Technology.

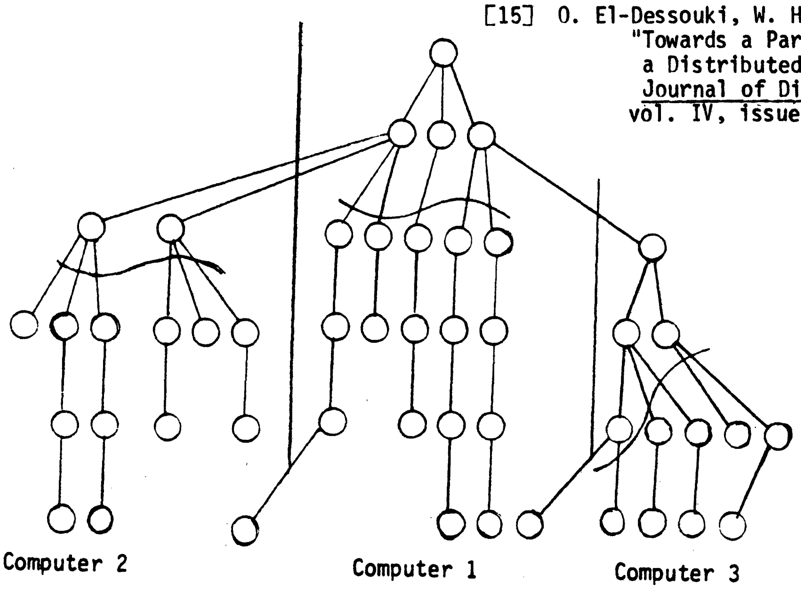
[11] J. Emrich, Partitioning Heuristics. M. S. Thesis, Illinois Institute of Technology, 1978.

[12] J. McCarthy, "Recursive Functions of Symbolic Expressions and their Computation by Machine," CACM 3, 4, (April, 1960) pp. 184-195.

[13] J. Backus, "Can Programming Be Liberated From the von Neumann Style? A Functional Style and its Algebra of Programs," CACM, 21, 8, (August, 1978), pp. 613-641.

[14] O. El-Dessouki, Program Partitioning and Load Balancing on Network Computers, Ph.D. Dissertation, Computer Science Department, Illinois Institute of Technology. Chicago, December, 1978/ pp. 160.

[15] O. El-Dessouki, W. Huen, and M. Evens, "Towards a Partitioning Compiler for a Distributed Computing System." Journal of Digital Systems, vol. IV, issue 4, 1981.



Circles -- activation records
Straight lines -- procedure calls
Curved lines -- density at which system chooses sequential execution

Figure 4. Procedure call tree resulting from dynamic choice of degree of concurrency.

MEASUREMENTS OF AN OPTIMIZING COMPILER FOR A VECTOR COMPUTER

John C. Knight
NASA Langley Research Center
Hampton, Virginia, 20606

Douglas D. Dunlop*
Department of Computer Science
University of Maryland
College Park, Maryland, 20742

Summary

The Control Data Corporation STAR-100 is a very-high performance vector processor[1]. A language known as SL/1 [2] that is oriented to scientific applications programming and which allows good program structure was designed and implemented by the authors for the STAR-100, and is now being used for many applications. SL/1 is also used with the CDC CYBER-203 but the work reported here was done using the STAR-100. In this paper we discuss the optimizations performed by the SL/1 compiler and report a series of measurements of the effects of these optimizations. The advent of vector processors and vector oriented languages such as SL/1 produces a new environment for scientific computation. Programs written for vector computers will be sufficiently different from their scalar counterparts that the effects of optimization in a compiler may be different. The primary optimizations of interest in the SL/1 compiler are common subexpression elimination, the movement of invariant code out of loops, and the elimination of unecessary vector temporaries. In order to get some information about the effect of optimizing programs written in a vector language, the performance of the optimizer in the SL/1 compiler was measured.

There are two hardware characteristics of the STAR-100 which are of importance in optimization. First, the hardware supports vector instructions with vector lengths between zero and 65,535, and the execution time of a vector instruction is proportional to its length after an initial start-up delay. For floating point addition, the longest vector instruction requires approximately one and one third milliseconds while the shortest requires only approximately three microseconds; a ratio of about 400 to one. Under ideal circumstances, a scalar floating point addition requires only 0.16 microseconds; a ratio of almost 10,000 to one compared to the longest vector instruction but only about 20 to one compared to the shortest. These ratios are important because the optimization techniques to be discussed are only applied to scalar operations. Vector operations are always included in SL/1 programs explicitly by programmers and there is usually nothing redundant that can be removed. Similarly, vector instructions are rarely inside loops in which they are invariant.

The second hardware characteristic of importance is the set of 256 general purpose registers. Variables which are used frequently can be stored in registers permanently [3], and the values of common subexpressions which appear in separate parts of a program can reside in registers between uses.

The SL/1 language structure is modelled after SIMPL_T [4]. Variables can be declared as scalars, vectors, or arrays. Arrays of scalars are not allowed and all array elements must be vectors. A matrix is therefore represented by a one dimensional array of vectors, and for a given matrix, the user may interpret these vectors as rows or columns.

As well as basic vector arithmetic, the STAR-100 hardware provides a variety of sophisticated macro operations. For example, forming the inner product of two vectors is a single machine instruction, as is the evaluation of a polynomial for a vector of coefficients and a vector of arguments. All of these macro instructions are available in SL/1 as special operators which can be used freely in building expressions. The compiler makes no attempt to recognize implicit vector operations in loops containing scalar computations since the language provides access to all the hardware vector facilites.

Key elements of the language are the array and vector referencing notations. Variables declared as vectors or arrays can be indexed in the normal way yielding a vector in the array case and a scalar in the vector case. It is also possible to select a range of elements, known as a subvector, from a vector variable or array element using notations which specify the index of the first element and length, or the indices of the first and last elements.

The SL/1 compiler is organised into three phases. The first phase translates the given SL/1 module into a series of quadruples (quads). The second phase optimizes the quads, and the third phase translates these optimized quads into a relocatable object module. In the rest of this paper, the term quad is used to mean an operator of the intermediate form and all (possibly zero) of its associated operands.

There are two important characteristics of the quadruple intermediate form. First the sequence contains quads which represent the control structure of the program in terms of the control statements of the language. This enables the optimizer to detect explicit program loops and control flow very easily. Secondly, some high-level operations such as indexing and forming subvectors translate into sequences of low-level quads which represent single instructions. This enables the optimizer to detect redundant computations in these high-level operations.

*Work performed under NASA contract number NAS1-14900 while the author was with Computer Sciences Corporation, Hampton, Virginia.

0190-3918/81/0000/0058$00.75 © 1981 IEEE

For common-subexpression analysis and code motion, the design of the optimizer is similar to the quad improver described by Hecht [5].

SL/1 allows arbitrarily complex vector expressions. This may result in the creation of temporary vectors, and these vector temporaries may be of different lengths. Building a temporary necessitates the execution of several scalar instructions to allocate space in virtual memory and increases the program's working set size by the size of the vector temporary. The compiler attempts to minimize the number of vector temporaries required to evaluate an expression in order to reduce this overhead. One technique employed is to use a single vector temporary in place of a number of equal length vector temporaries whose life spans are disjoint. This technique is a generalization of the algorithm described by Dantzig and Reynolds [6] which has been shown to minimize the necessary number of temporaries. A second technique is used only when the expression constitutes the right hand side in a vector assignment. In this case the compiler attempts to use the left hand side variable in place of one of the vector temporaries.

Five SL/1 programs which were considered typical were measured by an instrumented version of the SL/1 compiler. Table 1 shows the total number of quads and words of machine code with and without all optimizations, and the length of each program in lines. On average 27% of the quads, and 28% of the machine code were removed.

TABLE 1 - Overall Quad and Code Reductions

	Program Number				
	1	2	3	4	5
Source Lines	255	691	986	603	1647
Quads Without Opt.	641	2894	1910	1665	3741
Quads With Opt.	491	1710	1688	1196	2516
Reduction	23.4%	40.9%	11.6%	28.2%	32.7%
Code Without Opt.	554	3567	2638	1835	3680
Code With Opt.	462	2168	2229	1079	2656
Reduction	16.6%	39.2%	15.5%	41.2%	27.8%

Several quad operations had a relatively high probability of being redundant. Sixty-four percent of the scalar addition quads and 57% of the subvector quads were removed by common subexpression elimination. An optimizer which considered only these two quad operations would detect 86% of the total number of redundant quad operations for the five sample programs.

Table 2 shows the static frequency of occurrence of certain SL/1 statements. Assignment represents at least 74% of the total number of executable statements and the average proportion is 85%. As well as occurring in large numbers, assignment statements occur in groups and large basic blocks tended to dominate. Table 3 shows the largest basic block observed for each program and the proportion of each program which was made up of basic blocks which were ten or more lines long.

TABLE 2 - Statement Frequencies

	Program Number				
	1	2	3	4	5
Assignment	115	357	357	244	757
Procedure Call	9	3	34	8	69
IF Statement	13	1	11	7	26
FOR Statement	7	19	12	7	5
WHILE/REPEAT Statements	2	0	6	0	1
GO TO Statement	2	0	1	0	2

TABLE 3 - Basic Block Sizes

	Program Number				
	1	2	3	4	5
Largest (In Lines)	19	110	139	176	293
Ten or More Lines	40%	70%	77%	88%	66%

From Tables 2 and 3 it can be seen that relatively little use is made of control structures. A simpler optimizer is possible if common subexpression analysis is performed only across basic blocks. The SL/1 optimizer was modified to operate in this way and the five sample programs were recompiled. Table 4 shows the total number of quads and words of machine code with this less powerful optimization and with no optimization.

TABLE 4 - Quad and Code Reductions

	Program Number				
	1	2	3	4	5
Quads Without Opt.	641	2894	1910	1665	3741
Quads With Opt.	537	1710	1767	1200	2617
Reduction	16.2%	40.9%	7.5%	27.9%	30.0%
Code Without Opt.	554	3567	2638	1835	3680
Code With Opt.	491	2168	2320	1082	2721
Reduction	11.4%	39.2%	12.1%	41.0%	26.1%

The effectiveness of eliminating unnecessary vector temporaries was measured and the results are shown in Table 5. The average reduction in code volume is 10.3%. These measurements were made without common subexpression elimination. By comparing Table 5 with Table 1 it can be seen that in terms of code volume reduction, eliminating unnecessary vector temporaries made a large contribution to the total optimizer's performance on three of the sample programs.

TABLE 5 - Vector Temporary Elimination

	Program Number				
	1	2	3	4	5
Code Without	554	3567	2638	1835	3680
Code With	549	3061	2384	1372	3623
Reduction	0.9%	14.2%	9.6%	25.2%	1.5%

Table 6 shows the measurements of code motion on the sample programs. No candidate quad was found to be invariant inside two or more nested loops in any of the programs. The performance of code motion is rather poor due partly to the caution which is exercised in selecting operations to move, and partly to the relatively small numbers of explicit program loops.

TABLE 6 - Code Motion Effect

	Program Number				
	1	2	3	4	5
Total Quads	491	1710	1688	1196	2516
Quads Considered	60	74	230	50	31
Quads Moved	10	7	194	2	20

For the majority of users, the most important benefit from optimization is the reduction in program execution time which it is expected to produce. The five SL/1 programs used in this study were each executed with no optimization and with full optimization using data supplied by the programmer and regarded as typical. The percentage reductions in execution times produced by the optimizations were:

Program Number				
1	2	3	4	5
1.0%	28.87%	1.79%	3.6%	0.4%

Except for program 2, these reductions are hardly of any value. Notice that program 2 also experienced the largest quad volume reduction. The reason for these poor results is that the optimizer removes scalar operations only and the five programs were heavily vectorized; their execution times were dominated by very long duration vector instructions. The execution of 10,000 scalar instructions must be prevented in order to have an effect comparable in execution time with a single vector instruction operating on long vectors. The performance of the optimizer in the critical area of execution time is thus very dependent on the vector lengths used and the degree of vectorization of the program.

In order to assess the effect of different vector lengths on optimizer performance, program 1 was executed with vectors ranging in length from 64 to 16,128. The percentage reduction in execution time for the various vector lengths resulting from use of the optimizer are:

Length	Reduction	Length	Reduction
64	31.0%	5888	3.0%
128	26.0%	7168	2.5%
256	25.0%	8448	3.6%
320	20.0%	9278	2.2%
640	17.0%	11008	1.7%
768	15.6%	12288	1.8%
2048	10.2%	13568	0.7%
3328	5.0%	14848	1.3%
4608	3.7%	16128	1.0%

For very short vectors, the optimizer's performance is considerably better than with long vectors.

In the SL/1 optimizer, a very small subset of the quad operators was responsible for most of the code removal, and analysis of common subexpressions across control structures and code motion both proved relatively ineffective. In addition, the ratio of instruction execution times means that the effects of optimization are extremely program dependent and in terms of execution time, optimization was of almost no benefit in many cases because of the dominance of long vector instructions. This problem is significantly worse with the CYBER-203 where the instruction execution time ratio is much higher. The CRAY-1 [7], on the other hand, has a maximum vector length of 64 and the instruction execution time ratio is orders of magnitude less than that of the STAR-100 and CYBER-203. The optimizations attempted by the SL/1 compiler would probably be much more effective on programs which are executed on the CRAY-1.

A very simple optimizer is probably most appropriate for vector-oriented languages on machines like the STAR-100.

References

1. CDC STAR-100 Hardware Reference Manual, CDC Publication Number 60256000, Control Data Corporation, Minneapolis, Minnesota.

2. SL/1 Language Reference Manual, Analysis and Computation Division, NASA Langley Research Center, Hampton, Virginia 23665.

3. Dunlop, D. D., J. C. Knight, "Register Allocation in the SL/1 Compiler", Proceedings of a Workshop on Vector Processors, Los Alamos, New Mexico, 1978.

4. Basili, V. R., A. J. Turner, "Simpl T, A Structured Programming Language", Computer Note CN-14.1, University of Maryland, College Park, Maryland.

5. Hecht, M. S., "Data Flow Analysis of Computer Programs", American Elsevier, New York, N. Y.

6. Dantzig, G. B., G. Reynolds, "Optimal Assignment of Computer Storage By Chain Decomposition of Partially Ordered Sets", Report No ORC-66-6, University of California at Berkeley, O. R. Center, March 1966.

7. Cray-1 S Series Hardware Reference Manual, CRAY Publication No HR-0808, CRAY Research Inc., Mendota Heights, Minnesota.

THE SYMBOLIC, HIGH-LEVEL LANGUAGE PROGRAMMING OF AN MIMD MACHINE

David Klappholz
Department of EE/CS
Polytechnic Institute of New York
Brooklyn, NY 11201

1. Introduction

The present work is concerned with the high level language programming of a large-scale, tightly-coupled, speedup-oriented MIMD machine of the type proposed in [1] or [2].

We will assume a high level language which differs from traditional high level languages only in that it contains constructs for:

- dynamic (run-time) spawning of parallel processes
- run-time communication between processes
- dynamic (run-time) identification of one process by another for the purpose of establishing communication.

What we have in mind for the first of these constructs is something on the order of a SPAWN statement of the following type:

SPAWN <name of code>(<parameter 1>,...,<parameter k>)

What we have in mind for the second of these constructs is statements of the following type:

a) WRITE BUFFER <buffer name> FROM
<private variable>

b) READ BUFFER <buffer name> INTO
<private variable>

where <buffer name> is the identifier of a shared variable.

We will be concerned with:

i) showing that if direct interprocess communication is limited to communication between processes which bear the parent-child relationship, with all other communication constructed indirectly from parent-child communications, then the speedup promised by parallelism will, in general, be vitiated

ii) proposing a construct for the dynamic (run-time) identification of an arbitrary process by another process for the purpose of establishing communication.

2. Parent-Child Communication

We define the "spawning tree" of a system of cooperating sequential processes to be that graph:

- whose nodes represent the processes
- whose directed arcs represent the spawning relation. I.e., a directed arc from node A to node B represents the fact that A spawned B.

Different systems of cooperating processes will, of course, have their own idiosyncratic communication patterns. In order to get a handle on the general case, we will assume a random pattern of necessary communication. I.e., we will assume that wherever a process might "sit" on the spawning tree:

- each time a process needs to communicate with another process it will choose the latter process at random from a uniform distribution over all the processes (including itself for the sake of simplicity) in the system of cooperating processes
- different communications from the same (source) process will be directed at destination processes chosen independently of one another
- different (source) processes will choose the destinations of their communications independently of one another.

These assumptions, are, in one important sense, very optimistic. That is, in the long run they ensure uniformity of spread of the total volume of communication traffic over the set of all pairs of processes rather than possibly skewing that same total volume of traffic. What we will see, however, is that the spawning tree, because of its structure, will still form a very inefficient base for carrying communications; i.e., we will see that uniformity of traffic over the set of all pairs of processes when superimposed on the hierarchical structure of the spawning tree creates intolerable speedup-vitiating bottle-necking.

To start, then, let us take as our unit of time the time within which a process - on the average - sends a communication to some (randomly chosen) process. To simplify matters, and without loss of generality, we will assume that each process sends exactly one communication to some process during each unit of time.

We will assume, then, that once per unit of time each node (process) of a full binary (spawning) tree containing $N = 2^m - 1$ nodes will generate one communication addressed to some node chosen at random from among all N nodes. For each arc, α, of the tree we will be interested in the amount of traffic, T_α, - i.e., the number of communications - generated during one unit of time and destined to traverse the arc α at some point in its journey from its source to its destination. More precisely we will be interested in $E(T_\alpha)$, the expectation of T_α.

0190-3918/81/0000/0061$00.75 © 1981 IEEE

Now a full binary tree with $N = 2^m - 1$ nodes is, of course, of depth $d = m - 1$. For some ℓ then, $1 \leq \ell \leq m-1$, let the arc α be ℓ levels up from the leaf nodes of the tree as in Figure 1. If we let:

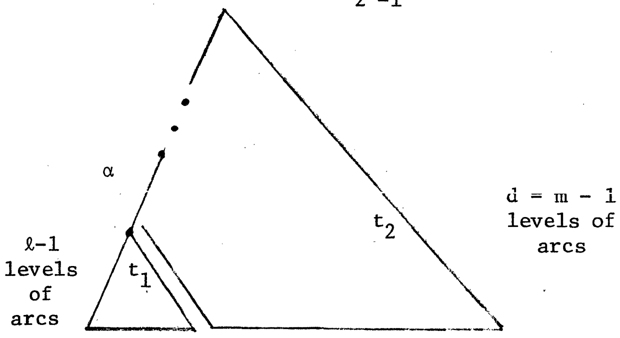
- a be the number of nodes in the subtree t_1, of Figure 1 (including the root node of t_1)
- b be the number of nodes in that part of the tree of Figure 1 (clearly not a subtree) labeled t_2 (i.e., all the nodes of the entire tree except those in t_1)
- T_α^{up} be the number of communications (generated during one unit of time) destined to traverse α in the upward direction
- T_α^{down} be the number of communications (generated during one unit of time) destined to traverse α in the downward direction

then:

$$E\left(T_\alpha^{up}\right) = \frac{ab}{a+b} = E\left(T_\alpha^{down}\right)$$

But $a = 2^\ell - 1$, and $b = 2^m - 1 - a = 2^m - 2^\ell$. We thus have:

$$E\left(T_\alpha^{up}\right) = \frac{(2^\ell-1)(2^m-2^\ell)}{2^m-1}$$



(FULL BINARY SPAWNING TREE OF DEPTH d)

Figure 1

or

$$E(T_\alpha) = \frac{2(2^\ell-1)(2^m-2^\ell)}{2^m-1}$$

If we now let α be either of the arcs for which $\ell = m - 1$, i.e., either of the arcs directly emanating from the root of the full binary tree we see that

$$E(T_\alpha) = \frac{2(2^{m-1}-1)(2^m-2^{m-1})}{2^{m-1}-1}$$
$$= (2^m-2)\left(\frac{2^{m-1}}{2^{m-1}-1}\right)$$
$$= (N-1)\left(\frac{2^{m-1}}{2^{m-1}-1}\right)$$

What this means is that, subject to our optimistic statistical assumptions, each period during which every process generates one communication causes the process at the root of the tree to perform O(N) units of work sequentially. (The root process is, after all, as are all the processes, a sequential process, and it is expected to have to handle $(N-1)(2^{m-1}/2^{m-1}-1)$ communications.) What this means, among other things, is that an N-process system of parallel processes is not expected to terminate in less than O(N) time.

3. Process Identification

Given, then, that in general implementing an arbitrary inter-process communication as a sequence of parent-child communications leads to intolerable loss of speedup, it is necessary for communicating processes to be able to directly identify one another for the purpose of establishing direct communication.

In the simplest case, i.e., that of two specific processes which are known at compile-time (actually, at the time the program is written) to have to communicate with one another, there is no problem. For example, suppose that procedures PROCA and PROCB are each to be activated exactly once, and that the one activation of PROCA is to communicate to the one activation of PROCB a result which the former will compute and store in its private variable RESULTA; the programmer need simply invent a buffer name, say BUFFAB, and a name for a private variable, say RESULTFROMA, and then write the code for PROCA and PROCB as in Figure 2.

```
PROCEDURE PROCA;
SHARED BUFFAB;
    .
    .
    .
RESULTA: = ... ;
WRITE BUFFER BUFFAB FROM RESULTA;
    .
    .
    .
END;

PROCEDURE PROCB;
SHARED BUFFAB;
    .
    .
    .
READ BUFFER BUFFAB INTO RESULTFROMA;
    .
    .
    .
END;
```

(EXAMPLE OF COMMUNICATION CODE WHEN COMMUNICATION PATTERN IS KNOWN EXPLICITLY AT TIME OF PROGRAM-WRITING)

Figure 2

Suppose, though, that the situation is more complicated, i.e., suppose that for the application of interest, processes must dynamically - i.e., on the basis of results which they will compute rather than on the basis of criteria explicitly known at compile time - "develop the need" to communicate with one another. How, in this case, are procedures to be coded in such way that processes which "develop the need" may establish a means of communication with one another?

For the purpose of enabling such communication, we propose constructs for the dynamic creation of variable names. To wit, the notion of a schematic variable name is defined as follows:

```
<schematic variable name>:: = (<schema>) ;
<schema>:: = <character> |
             <arithmetic expression> |

             <schema><schema>;
```

Note that in the above definition of <schema>, <arithmetic expression> denotes an arithmetic expression each of whose characters is underlined.

The semantics of schematic variable names is as follows:

- an underlined arithmetic expression is to be evaluated, and the numeric value translated into the character string representing that value

- a character not underlined represents itself.

Thus, for example, if

WRITE BUFFER (JOE/I**2/J+8)FROM <private-variable>

is executed at a time at which I has the value 5 and J has the value 3, then the statement which will effectively be executed will be

WRITE BUFFER (JOE/25/11)FROM <private-variable>

The manner in which processes which dynamically develop the need to communicate establish a means of communication is clear. Before the introduction of dynamically-created names two processes communicates with one another if and only if one executes a statement of the form READ BUFFER <buffer-name 1> INTO <private-variable 1>, the other executes a statement of the form WRITE BUFFER <buffer-name 2> FROM <private-variable 2> and <buffer-name 1> happens to be identical to <buffer-name 2>. This is still of course true, but now the name of the buffer may itself be computed at run time.

The specific details of the proposed construct for the dynamic creation of buffer names, however, is not the important point. Rather, what is of consequence is that once large-scale, tightly-coupled, speedup-oriented MIMD computation becomes a widespread reality, algorithms will be developed which will require the dynamic establishment of communication on the basis of computed results. This will be the case, for example, in the solution of PDE's over dynamically varying grid structures and in such AI applications as natural language understanding. In such applications, some means for the dynamic creation of buffer names or some alternative means for the dynamic identification of one process by another will be of critical importance.

References

[1] Sullivan, H. and Bashkow, T. R., "A Large Scale Homogeneous, Fully Distributed Parallel Machine, I" in Proc. Fourth Annual Symposium on Computer Architecture, March, 1977.

[2] Klappholz, D., "An Improved Design for a Stochastically Conflict-Free Memory/Interconnection System," in Proc. Fourteenth Asilomar Conference on Circuits, Systems, and Computers, Nov., 1980.

A PARALLEL HETERARCHICAL MACHINE
FOR HIGH LEVEL LANGUAGE PROCESSING

Adolfo Guzmán

Computing Systems Dept.,IIMAS
National University of Mexico
Apdo. Postal 20-726
México 20, D.F.

Abstract

A computer architecture is presented that processes in parallel programs written in high level languages capable of being expressed in the lambda notation (applicative languages).

Internally, it is a collection of weakly-coupled general purpose processors, without a hierarchy among them. Each processor evaluates a part of a program, thus permiting asynchronous computation.

The architecture here exposed has been developed for the Lisp language, although other applicative languages are also possible. The hardware implements the function calls, argument passing and sequencing of tasks. Each processor is a Z-80 microprocessor that is programmed to execute the Lisp primitive operations.

The AHR machine operates as a slave of a general purpose minicomputer. This avoids doing I/O in the AHR machine. In addition, all interactions with the user(s) are done by the normal operating systems of the mini.

The machine is being built at the Computing Systems Dept. (IIMAS).

I. Introduction and Project Status

This paper presents the architecture of a parallel general purpose computer that has Lisp as its main programming language. It is built of several dozens of microprocessors (Z-80's), each of them executing a part of the program.

Goals

The goals of the Project AHR (Arquitecturas Heterárquicas Reconfigurables) are:

* To explore new ways to perform parallel processing.
* To have a machine in which it will be possible to develop parallel processing languages and software
* To have a tool for students to learn and practice parallel concepts in hardware and software.

Project Status

Version 0 [3] of the machine has been designed and simulated. This produced Version 1[12] which was simulated using SIMULA. Results of the simulation are not to be found here, but in [8, 9, 12] instead.

We are building Version 1 of the machine, expected to be operational [5] in 1981. Subsequently, a faster version will be built, possibly incorporating changes and ideas sprung from our experience with the first machine. Finally, this fast version will be used to try to attain the goals mentioned above.

About six people full time are involved in the project.

The expected uses of the machine also include picture processing, finite element methods, engineering calculations, and distributed processing.

Main Features

The AHR machine has the following characteristics:

* general purpose.
* parallel processor.
* heterarchical. It means that there is no hierarchy among the processors; there is no "master" processor, or controller. All the processors are at the same level.
* asynchronous operation.
* it has Lisp as its main programming language.
* processors do not communicate directly among themselves. They only "leave work" for somebody else to do it.
* no input/output. This is handled by a minicomputer to which the AHR machine is attached.
* no operating system (software). Most of the Lisp operations, as well as the garbage collector, are written in Z-80 machine language
* the AHR machine works as a slave of a general purpose computer (a mini or micro).
* gradually expandible. More microprocessors can be added as additional computing power is needed. [9]

Functional Notation

The AHR machine obtains its parallelism by parallel evaluation of the arguments of functions. For instance, in f(a,b, g(u,g(x,b))), first x and b are evaluated; then g of them, in parallel with u; then g of the result, in parallel with a and b. That is, evaluation occurs from bottom up, or from the inside to the outside of the expression. This is in accordance with the rule for evaluation of a

0190-3918/81/0000/0064$00.75 © 1981 IEEE

function: "to evaluate a function, the arguments have to be already evaluated".

Recursion is handled [3] by substituting the function name ("FACTORIAL") by its function definition (LAMBDA (N) (IF (EQ N 0) 1...)) when evaluating it.

The machine works with pure Lisp, without SETQ's, GOTO's, Label's, RPLACA.

II. The Parts of the AHR Machine

In this section the constituents of the machine are described; section III explains how the machine works. Refer to figure 2.

Passive Memory

This memory holds lists and atoms; it holds partial results and parts of programs that are not being executed at the moment.

Originally, the programs to be executed reside here, and they are copied to the grill for their execution. As new data structures are built as partial results of the evaluation, they come to the passive memory to reside.

The Grill

This memory holds the programs that are being executed. A program, once in the grill, is being transformed into results, as the result of its evaluation.

Programs reside in the grill in the form of nodes, as figure 1 illustrates. Each node is pointed at by its sons (its arguments), and its nane field contains the number of nonevaluated arguments. Nodes with nane = 0 are ready for evaluation.

The Lisp Processors

These active units are microprocessors (about several dozens of Z-80's) that obtain from the grill nodes ready for evaluation, and, after evaluation, return results (s-expressions) to the grill. Each Lisp processor knows how to execute every Lisp primitive. Each of them works asynchronously, without communicating with other processors.

The processors obtain new work to be done from the distributor, through the high speed bus. This work comes as a node ready to be evaluated.

Only nodes with nane = 0 come up to the Lisp processors for evaluation. So, for instance, (CAR '(A B C)') will evaluate to A. The node (CAR '(A B C)') has become the result A. The Lisp processor has to do, after evaluation, the following things:

1.- Insert the new result A in the cell (in the grill) pointed to by the node (CAR '(A B C)'). That is, insert such result in a slot of the father of the evaluated node (see such slots in figure 1).
2.- Release the grill space occupied by node (CAR '(A B C)').
3.- Substract 1 to the nane of the father.
4.- If the new nane (of the father) is zero, inscribe the father in the fifo: the father is now ready for evaluation.

```
(LIST (CONS (CAR A)
            (CDR B) )
      X
      Y )
```

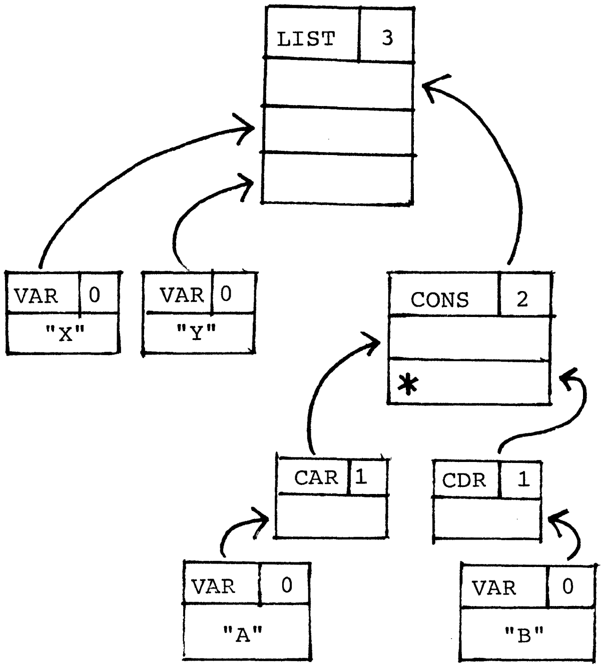


Figure 1
NODES IN THE GRILL

Above, the Lisp expression to be evaluated. Below, how it is structured into nodes, each node being a function or a variable. Each node shows a number: its nane, or number of non-evaluated arguments. When a node has a nane of zero, it means that such node is ready for evaluation.

Empty words are slots where the results of evaluation will be inserted. For instance, the results of (CDR B) will be inserted in the slot marked "".*

These steps are initiated by the processor simply by signaling to the distributor that the processor has finished, and that its results should be handled in mode "normal end" (burocracia de salida, in Spanish [12]); the distributor itself performs the requested steps.

Notice that in this form nobody has to search the grill looking for nodes with nane=0, because as soon as they appear, they are inserted into the tail of the fifo.

The Lisp processors have access to the passive memory (where lists and atoms reside), and to the variable memory, where we have the values of variables.

A Lisp processor is either busy (evaluating a node) for it is ready to accept more work (another node).

The high speed bus

Connecting each Lisp processor with the distributor is a high speed bus that goes into the private memory of each processor. The new node that the distributor throws is inserted (through the high speed bus) into the memory of the selected processor. Then, the processor is signaled to proceed.

The slow speed bus

This bus runs from the i/o processor (the mini or micro to which the AHR machine is connected) to each bos. It is not shown in the diagrams, nor it is explained furthermore in this article (See [5]). Through this bus each processor is loaded with programs, prior to starting the machine. Also, in the debugging stage, the slow bus is used to pass statistical information to the i/o processor. The slow speed bus is not used during normal execution of Lisp programs.

Variable Memory

This memory contains pairs of (variable,value), and it is organized as a tree, or a collection of a-lists, where each pair (variable,value) points to older pairs. It is accessed by the Lisp processors, and it is augmented (a branch of the tree grows) after each LAMBDA binding.

Since the evaluations are made in parallel, the a-lists could grow in parallel, too. For instance, consider the following expression

BODYO: (list ((lambda(X) BODY1) 3) ((lambda(X) BODY2) 4)).

Then, if when evaluating BODYO the a-list is

ALISTO: ((X,A) (Y,B) (Z,9))

Then, when evaluating BODY1, the a-list is

ALIST1: ((X,3) (X,A) (Y,B) (Z,9));

and when evaluating BODY2, the a-list is

ALIST2: ((X,4) (X,A) (Y,B) (Z,9)).

But since the evaluation of BODY1 and BODY2 can be carried in parallel (by two different Lisp processors), this means that ALIST1 and ALIST2 coexist at the same time in variable memory, but BODY1 points to ALIST1 and BODY2 points to ALIST2. So, each processor has its "appropriate" a-list to work with.

Both ALIST1 and ALIST2 share ((X,A) (Y,B) (Z,9)) between them. That is, they "share" ALISTO. ALISTO grew in two directions, like a tree, giving rise to ALIST1 and ALIST2 simultaneously. This explains the affirmation that "the variable memory contains a tree of a-lists".

The Distributor

This piece of hardware communicates the grill with the Lisp processors. The distributor keeps in the fifo (a memory) an array of nodes ready to be evaluated; these nodes are thrown, one in each cycle of the distributor, to the Lisp processors that are ready to accept new work. An arbiter decides which Lisp processor obtains the node; an exchange is done (through the high speed bus) between that Lisp processor and the distributor, the processor accepting the node and releasing the result of the previous evaluation. The distributor stores the result in the grill, in the address indicated within the result. Generally, this result is stored in a slot of the node which is father of the node just evaluated.

An overall view of the machine is shown in figure 2.

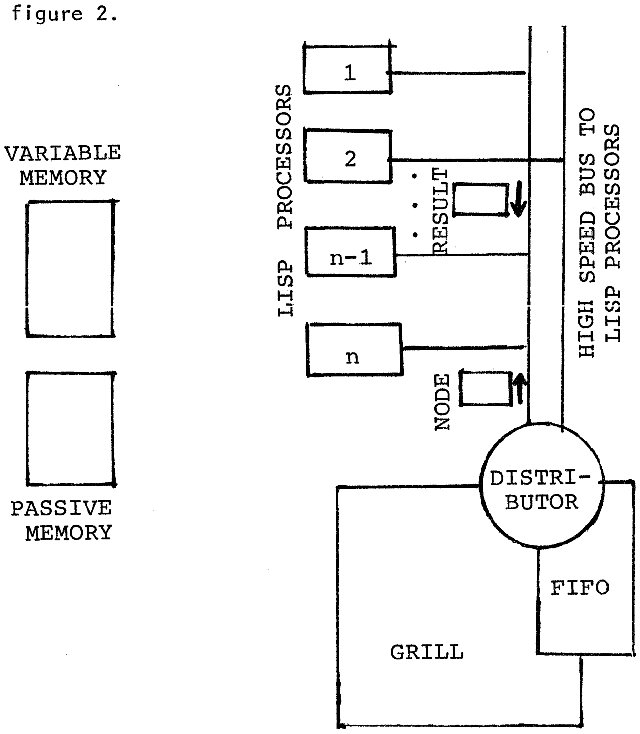


FIGURE 2

THE AHR MACHINE

Lisp processor 2 is ready to accept more work. The distributor fetches a node (to be evaluated) from the fifo and sends it to processor 2, while accepting the results of the previous evaluation performed by such processor. That result is stored in the grill, in a place indicated in the *destination address of the result.*

Such exchange of new work--previous result is performed at each cycle of the distributor.

Version 2 of the AHR machine will gain speed over Version 1, mainly by building a faster distributor.

The Lisp processors also have access (connections not shown) to the variable and passive memories.

The Fifo.

The fifo is a first in-first out memory that holds pointers to nodes (in the the grill) ready to be evaluated. The distributor fetches such nodes through the head of the fifo, while new nodes to be evaluated are inserted through its tail [5].

The arbiter.

If several Lisp processors become ready to accept more work, the arbiter (a hardware) selects one of them, which will receive the node thrown by the distributor.

If every processor is busy, the cycle of the distributor is wasted, since no processor accepts the node that the distributor is offering.

The I/O Processor

It has been said that the AHR machine can be seen as a peripheral of a general purpose minicomputer. But this mini can also be considered as a peripheral of the AHR machine; we thus talk of such mini as the I/O processor.

Input/output will be described in next section.

III. How The Machine Works

Input

The user uses a terminal of the mini or micro (i/o processor) which is master of the AHR machine. He uses a common editor, disks and the normal operating system of the mini. When he is ready to run a program, he loads it from disk into a part of the address space of the mini which is really the passive memory of the AHR machine (see figure 3 . In this way, the program is loaded (already as list cells) in the passive memory. A signal from the i/o processor to the AHR machine signifies that Lisp execution should begin. Together with this signal an address is passed, indicating where in passive memory resides the program to be evaluated.

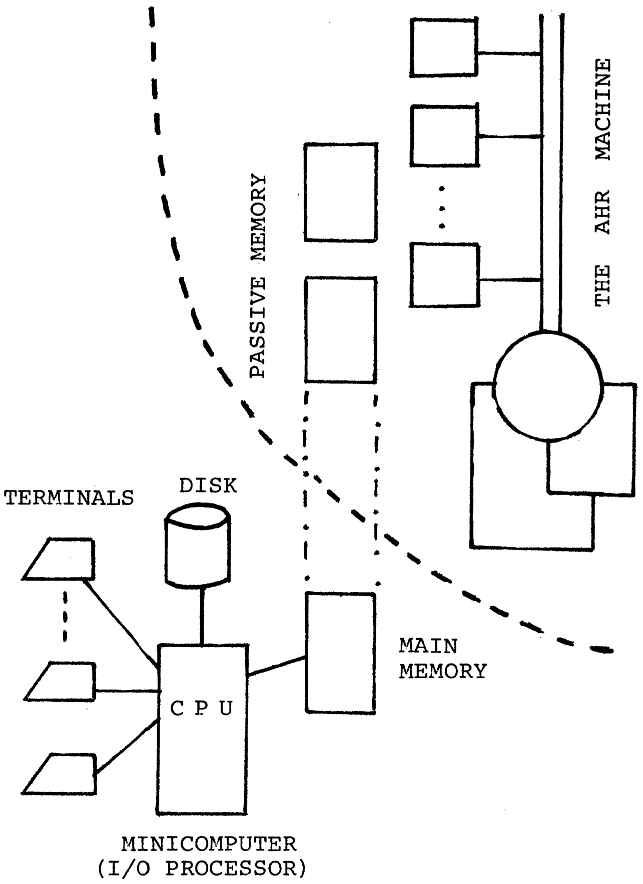


FIGURE 3

"THE AHR MACHINE AS A SLAVE"

The AHR computer is shown as another peripheral of a general purpose minicomputer. The address space of the mini comprises the passive memory of AHR, through a movable window of 4k addresses.

Starting

It is assumed that each Lisp processor already has its programs loaded in its private memory.

When the AHR machine receives the "start" signal, the distributor throws a node (called the RUN node) to some Lisp processor. This node points to the program which will start.

The program (in passive memory) is copied (i. e., transformed from its passive-memory representation, which is in list notation, to its grill-representation, which is composed of nodes) by more and more Lisp processors (the more leaves or branches a program has, the more processors help to copy it. Each processor copies a branch of the program)into the grill. Nodes with nane=0 are inserted by the Lisp processors into the fifo, so that some other Lisp processors will execute them.

Finally, the program has been copied into the grill. Notice that at the same time of copying, some nodes with nane=0 could have been evaluated by some other Lisp processors.

Evaluation

When a Lisp processor is idle, it signals to the distributor, meaning that it is ready to accept more work.

The distributor chooses (with the help of an arbiter) one of several idle processors, and through the high speed bus it injects a new node [taken from the grill through the head of the fifo] into its private memory. It then signals such processor to start.

The Lisp processor "discovers" the node in its own memory, with all the arguments already evaluated. The Lisp processor proceeds to perform the evaluation that the node demands. Suppose it is LIST, and its arguments are (A B), M and N. It then has to address the passive memroy in the mode "give a new cell". Such cell is given by a cell dispatcher (hardware attached to passive memory). Three new cells have to be requested. Then the Lisp processor forms the result: ((A B) M N). For this, it has to store pointers to (A B), to M and to N, into passive memory, in the new cells already obtained. Then, it stores the result (which is a pointer to passive memory) into a special place ("results place") of its private memory. It has finished. It signals to the distributor that it is ready to accept more work. The distributor will insert new work (another node with nane=0) into the private memory of the processor, but it will also collect (through the high speed bus; see figure 2) from the "results place" in private memory, the result ((A B) M N). The distributor will store this result into a slot in a node in the grill. The address in the grill of this slot was known to the (LIST (A B) M N) node, because each node points to its father. Thus, the distributorhas no problem in finding where to store the result: such address is found also in the "results place", together with the result ((A B) M N).

The distributor has to do one more thing: it has to substract one from the nane of the father (which has just received the result ((A B) M N). And if such nane becomes zero, then a pointer to the father is inserted by the distributor into the fifo through its tail.

One last thing: the distributor has to free the cell of the node (LIST (A B) M N), so that this grill space could be reused [10] .

The distributor is very fast compared with the speed of the Lisp processor. This will be even more true if we code "Complicated" Lisp functions (such as MEMBER OF FACTORIAL) in Z-80 machine language, instead of "simple" Lisp functions, such as CDR.

Due to such difference in speed, the distributor can keep many Lisp processors working; if the distributor is 100 times faster than the (average) Lisp function, it could keep 100 Lisp processors functioning. It pays to make a fast distributor.

Output

Finally, the whole program has been converted into a single result (let us say, a list) deposited in passive memory. The AHR machine now signals the mini (or i/o processor), giving it also the address in passive memory where the result lays. The mini now accesses the passive memory as if it were part of its own memory (remember, their address spaces overlap), and proceeds to the (serial) printing process.

Execution has finished.

IV. Hardware Considerations

Lisp Processors

The first version of the machine will have 5 Lisp processors, and the i/o processor is another Z-80. Each Lisp processor will have 4K bytes of private memory, where a pure-Lisp interpreter will reside [8].

The maximum number of Lisp processors is 64. It could be increased further, but a new arbiter needs to be designed in that case.

The high speed bus

The distributor inserts a node (7 words of 32 bits) into the private address space of the selected Lisp processor, through the high speed bus. It does this in 0.5 microseconds. The high speed bus runs from the distributor to all Lisp processors. It carries nodes and results.

The low speed bus

A 16 bits low speed bus; 8 of them indicate which Lisp processor is addressed, the other 8 bits carry data. It runs from the i/o processor to the Lisp processors.

An additional use of the low speed bus is to broadcast to the Lisp processors the number of a program that needs to be stopped or aborted.

Passive Memory

It consists of up to 2^{20} words of 22 bits; it contains the input ports, list space, output ports and atom space.

Version 1 will have only 64K words.

Access time is 150 nanoseconds. It has a parity bit.

The Grill

It consists of up to 2^{19}words of 32 bits. It is divided logically in nodes, each with 7 words.

Version 1 will have 8K words. Access time is 55 nanoseconds. The grill contains the nodes that are about to be evaluated.

Variable Memory

It consists of up to 2^{19} words of 32 bits. This memory contains names of variables and their values at a given time. The variable memory contains also real numbers, in its lower half. In its upper half it has "environments", which are lists of cells of 5 words each.

Version 1 will have 16K words. Access time is 150 nanoseconds.

The Distributor

The distributor passes nodes from the grill to the Lisp processors, and stores in the grill the results coming from the Lisp processors. There are two versions of the distributor.

First version of the distributor:

This first version [10] is implemented through a Z-80, using a program that performs all the functions of the distributor. It runs slowly, in the sense that distributes nodes at low speed. It is further described in Section V-Software considerations.

Second version: fast distributor:

Not yet built, it will become part of version 1 of the machine. It will be built either from bit-slice microprocessors, of from PAL's.

The fifo

Of a maximum size of 2^{19} words of 19 bits, it contains pointers to the nodes in the grill. Version 1 will be of 4K words. Its access time is 55 nanoseconds.

The arbiter

There are really three arbiters, for passive memory, variable memory and for the grill.

Each arbiter takes 400 nanoseconds to respond, and it may handle up to 64 processors. Each processor has a fixed priority, varying from 1 to 64. Each processor has a different (unique) priority. The assignment of priorities to processor really does not matter, since all of them are equal (they are able to perform exactly the same tasks). Of course, if there are too may processors, those with lowest priorities will never obtain work (nodes) to do.

The I/O Processor

It is actually built around a Z-80 that works as a general purpose computer. Its main functions are:

* to talk to the users; to read their input and to print their results.
* to store user files in its disk.
* to initialize the AHR machine.
* to load into passive memroy, through the window, the programs loaded from disk.
* To begin garbage collection.
* To end garbage collection.
* Actually, the garbage collector runs in the i/o processor.

V. Software Considerations

The Lisp Interpreter

A Lisp interpreter runs in each Lisp processor. It interprets pure Lisp (only evaluations; no setq's, rplacd's or other operators). The garbage collection is not done, at this moment, by the Lisp processors.

For the first version, the Lisp interpreter will do argument checking of the Lisp functions. This will reamin as an option in the second version of the AHR machine.

The Garbage Collector

For the first version of the machine, it will be a "normal" serial garbage collector, running in the i/o processor. While it works, the Lisp processors remain idle. For the second version, it will be a parallel incremental garbage collector, running in the Lisp processors.

Garbage collection is done for passive memory (list cells) and for the real numbers region of variable memory (where it compactifies memory). In the "environments" zone of variable memory and in the grill (nodes), there is no need to recollect garbage, because used space, as soon as it is abandoned in these two places, it is inserted (by hardware) into a list of free environment cells (for variable memory) or into a list of free nodes (for the grill).

The Distributor (First Version)

This is a piece of software [10] running in a Z-80, that emulates all the functions that the "real" (hardware) distributor performs. It is slow in this sense, but it is flexible and helps in the debugging of the AHR machine; it may be run "step by step" to see the flow of information. It also keeps statistics of use of hardware and software.

Editing

Editing of Lisp programs is done outside the AHR machine, using the operating system and editor of the i/o processor. After editing, the program is filed on disk. From here, a loader (running in the i/o processor) converts it into list cells and brings the program to passive memory. See figure 3.

Performance of the Machine

No figures can be given at this time, since the AHR machine is not yet completed.

New Advances as of June 1981.

The hardware is now working; the software is about to be completed.

VI. Related Work and Machines

Greenblatt's Lisp Machine

This is a single processor machine [14] built for high speed Lisp computations. It does not pretend to be an experiment in parallel hardware; it gains its speed and power from careful design of the software and machine architecture, as well as from the experience of the builders with the Lisp language.

Parallel Lisp Machine

The machine [7] is a loosely coupled multiprocessor for applicative languages such as Lisp. It is the machine most closely resembling ours, in its application.

Data Flow Machines

These machines [13] resemble the AHR architecture in that data is directed through "boxes" that process them. The flow of executions is controlled, like in our design, by what previous results are ready (available). The cited article describes a machine that uses different colors of tokens to mark "this result", "previous result", and so on.

Zmob

A collection of Z-80's around a conveyor belt, this machine [11] may be applied to image processing and numerical calculations. Each microprocessor has its own private memory. They do not have direct access to a common memory (as AHR does), but behind one of the micros, a huge central memory or mass memory may reside.

PM4

This is a machine [2] suitable for iamge processing. It is a dynamically reconfigurable multimicroprocessor-based machine. It can be partitioned into several groups of processors which may be assigned to execute multiple independent SIMD processes and MIMD processes.

The Language "L" for Image Processing

"L" is a language suitable for processing of images. It is mentioned here because it may be implemented in a parallel machine [4] , such as the AHR computer. The language is described elsewhere [1], · It was designed mainly as a result of our experience in picture processing of multispectral images [6] . "L" has not been implemented.

VII. Conclusions

The architecture of the AHR computer shows that it is possible to build a multiprocessor of the MIMD type, where each processor does not explicitly communicate with other processors. In The AHR design, a processor does not know how many other processors are there, or what they are doing. It is not possible to address a processor: "here I have a message for processor number 4."

The construction of new software has been kept low by connecting the machine to a general purpose computer, thus being able to use already available operating systems for time sharing, text editors and loaders.

Once the machine is built, experimentation will begin in the design of parallel languages and ways to express "powerful" commands in heterarchical fashion. Also, if the amount of access to memories for each processor is low, it may be possible to place each micro in a remote place, thus achieving some class of distributed computing. That is, a micro can process local work (through Basic, for instance) as well as remote (Lisp) work.

Finally, the AHR machine shows how it is possible to design a heterarchical system, where none of the processors tells the others what to do, in what order to do it, or what resources are available to whom.

Acknowledgements

The AHR machine is being built by the members of the AHR Project, to whom I express my appreciation for their time, effort and enthusiasm.

Work herein described has been partially supported by Grant 1632 from CONACYT, the National Council for Science and Technology (México).

References

1. Barrera, R., Guzmán, A., Jinich, A., and Radhakrishnan, T. Design of a highlevel language for image processing. 1979. Technical Report PR-78-22, IIMAS, National Univ. of Mexico.

2. Briggs, F.A., Fu, K.S., Hwang, K., and Patel, J.H. PM4: a reconfigurable multiprocessor system for pattern recognition and image processing. 1979. Technical report TH-EE-79-11. School of Electr. Eng., Purdue University (USA)

3. Guzmán, A., and Segovia, R. A parallel reconfigurable LISP machine. 1976. Proceedings of the International Conference on Information Sciences and Systems. Univ. of Patras, Greece. 207-211.

4. Guzmán, A. Heterarchical architectures for parallel processing of digital images. 1979. Technical report AHR-79-3, IIMAS, National University of Mexico.

5. Guzmán, A., Lyons, L., et al. The AHR Computer: construction of a multiprocessor with LISP as its main language. (in Spanish). 1980. Technical report AHR-80-10. IIMAS, National University of Mexico.

6. Guzmán, A., Seco, R., and Sânchez, V. Computer Analysis of LANDSAT images for crop identification in Mexico. 1976. Proceedings of the International Conference on Information Sciences and Systems. University of Patras, Greece. 361-366.

7. Keller, R.M., Lindstrom, G., and Patil, S. A loosely-coupled applicative multi-processing system. AFIPS 1979 Conference Proceedings, Vol. 48, 613-622.

8. Norkin, K., and Gômez, D. A new description for data transformations in the AHR computer. 1979. Technical report AHR-79-4, IIMAS, National University of Mexico.

9. Norkin, K., and Rosenblueth, D. Towards optimization in AHR. Technical report AHR-79-5, IIMAS, National Univ. of Mexico. 1979

10. Peñarrieta, L. Error detection in the AHR computer. (In Spanish). 1980. Technical report AHR-80-9. IIMAS, National Univesity of Mexico.

11. Rieger, C., Bane, J., and Trigg, R. ZMOB: a highly parallel multiprocessor. 1980. Technical report TR-911, Dept. of Comp. Science, Univ. of Maryland (USA).

12. Rosenblueth, D., and Velarde, C. The AHR machine for parallel processing: 1st stage. (In Spanish). 1979. Technical Report AHR-79-2, IIMAS, National University of Mexico.

13. Watson, Ian, and Gurd, John. A prototype dataflow computer with token labeling. AFIPS 1979 Conference Proceedings, 48, 623-628.

14. Weinreb, D., and Moon, D. Lisp machine manual. 1979. M.I.T.A.I. Laboratory, Cambridge, Mass. (USA)

DISTRIBUTED PROCESSING APPROACH FOR THE INTERNATIONAL PUBLIC TELEGRAMS MESSAGE SWITCHING SYSTEM

Jin-tuu Wang
Yen-son Lee
International Telecommunications Administration
Taipei, Taiwan, Republic of China

Abstract -- International Telecommunications Adminstration (ITA) in Taipei, Taiwan, Republic of China, recently has completed the application software development for its International Telegram Automatic Processing System (ITAPS). This system adopts an in-house computer network architecture that includes four closely coupled minicomputers and more than two dozens of microprocessors. Two of the minis serve as the front-end communication processors and others as the host message switching processors. These minis are interconnected using the SDLC protocol. The microprocessors are connected to the front of the communication processors using the RS-232C protocol to handle Telex signalling for those telegrams to be delivered/accepted to/from the Telex network. The ITAPS is configured to provide full redundancy so that the hot-standby processors will take over the on-line task should any failure occur in the on-line system. One of the special characteristics of the ITAPS is to print-out Chinese address information automatically on the received international telegrams to facilitate messenger's delivery. Besides hardware architecture of the system, this paper also describes the functional characteristics of the system, software design and the integrating testing result. This system is one of the large scale software development projects that are carried on in this country.

Introduction

The recent advent of minicomputer technology prompted the prevailing applications of using minicomputer systems for various types of transaction processing [1,2,3]. Message switching is one of such applications to automize the handling of message records. Although CCITT has set certain recommendations for these type of services, such as F.31 message format, various systems very often differ from one another due to different operational requirements of record carriers.
The International Telecommunications Administration (ITA) has called an international open tender for the international telegram message switching system in 1974, however, the bid was unsuccessful because none of the venders could propose a system that could meet the user's operational requirements. Furthermore, a non-standard project always requires tremendous man-hours to write the specific application software in order to meet these requirements, and the cost for developing such a non-standard software package is always very high. After few times of unsuccessful open tender on the turn-key basis, ITA decided to develop the necessary application software to meet its own operational requirements. A system appended with on-line handling of Chinese address information and with inter-connection to the Telex network is probably not available in the market. Therefore, it is worthwhile to develop such a non-standard system by yourselves not only to meet your own requirements but also to gain some practical experiences in the field of software engineering technology.

Hardware Architecture

Fig. 1 shows the hardware configuration of ITAPS. Two GA-16/440 minicomputers with 112 KW core memory and Memory Management System (MMS) serve as the host message switching processors, while two GA-16/440 minicomputers with 64 KW core memory and Memory Parity and Protection (MPP) option serve as the front-end communication processors. These four processors are connected with SDLC links in such a way that each host has a front-end processor through a link and is the standby of the other on-line host.

In order to facilitate automatic delivery/acceptance of incoming/outgoing telegrams to/from the Telex subscribers, Z-80 microprocessors are used to handle Telex signalling information with the Telex exchange. Each microprocessor is designed to handle four trunks of call setup and clear down signalling to/from the Telex exchange using 2K-byte EPROM and 256-byte RAM. These intelligent hardware interface boards were designed and manufactured locally to response to the CCITT No. 2 signalling protocol. Therefore, they serve as the protocol converters between the CCITT No. 2 and the RS-232C protocols.

All the peripheral devices are attached to the message switching host processors. These peripheral devices include 2 head-per-tracks (drums), 4 moving head disks, 8 magnetic tape drives, 2 card readers, 2 line printers and 2 CRT terminals as console. 16 CRT terminals are attached to the on-line host processor for manual assistance of the intercepted messages while 10 CRT terminals are connected in distance through modem to facilitate telegram entering directly from ITA branch offices. These CRT terminals are always connected to the on-line processors through Automatic Bus Transfer Unit (ABTU). The function of the head-per-track is to serve as the transit storage for each telegram entering the system, while that of the moving head disk is to serve as the short-term journaling of telegrams for later retrieval and as the storage space for operational files and programs. The magnetic tape drives are for the long-term journaling of telegrams, automatic ticketing of outgoing telegrams, system and file backup. The card reader and line printer are for system software development. The head-per-tracks are all connected to the on-line host through ABTU while moving head disks are connected to the dual port disk formatters and can be

0190-3918/81/0000/0072$00.75 © 1981 IEEE

accessed by either host processor. The magnetic tape drives, card reader and line printer are all dedicatedly connected to each host processor.

All the communication lines are connected to the ITAPS communication processors via two types of asynchronous communication multiplexors, one for the slow speed trunk-lines or teletype terminals, and the other to the modems for the remote CRTs. The former is the GA-1595 multiplexors that provides 64 lines PIO capability to input/output message character and line status one at a time after interrupt request. The latter is the GA-1535 multiplexors that provide 16 lines DMA capability to input/output message characters, line or page depending on the operational mode of CRT terminal. These multiplexors generate three types of interrupt to the communication processor, namely, the input buffer full, the output buffer empty and the status change of line so that the CPU can serve the respective type of interrupt to input, output character or sense the status of lines. The 1595 multiplexors are further connected to two types of line adaptors, one is the current loop line adaptors that provide neutral current loop interfaces to trunk-lines, the other is the RS-232C line adaptors that provide EIA interface to trunk-lines for the Telex exchange. Portion of the current loop line adaptors are connected to a neutral/bipolar current converter for those lines that are in bipolar characteristics. Four serial-type graphic printers are connected via RS-232C line adaptor to the 1535 multiplexor for printing Chinese address information on the received telegrams. A line monitor and patch panel is also installed to provide signal monitoring, line cross-patching, and trunk line interfacing for all the low speed lines and trunks.

Major Functions and Special Characteristics

The major functions of the ITAPS are to perform a store-and-forward message switching which automatically processes and routes both international incoming/outgoing telegrams to/from this country, stores the processed telegrams for later retrieval, and provides traffic relevant reports. Remote and local CRT positions are also provided to facilitate direct editing of outgoing telegrams at branch offices and manual assistance of the intercepted telegrams at the telegraph operation center (see Fig. 2). Detail functions are described as follows:

Automatic Classification of Incoming Telegrams

For the incoming telegrams, the ITAPS automatically classifies the telegrams into ten delivery classes. Four major classes are: (1) to be delivered through the Telex network; (2) to be delivered by messengers, (3) to be routed to the domestic network, and (4) to be printed on the local teletype terminals.

By using the cable address in the received telegram as keyword, the ITAPS looks up the Telex number and the Automatic Answer Back (AAB) from the database, if there is any, and gives the number to the microprocessor interface for automatic dial-out. If the circuit connection is successful, the communication front-end processor will send "WHO ARE YOU" (Figure D) signal to the connected Telex terminal for obtaining an Automatic Answer Back Code. If a complete match occured between the returned AAB and that gotten out from the database, the ITAPS sends out the incoming telegrams to the Telex subscriber who has registered using this cable address. If a Telex number is not found under this cable address, the ITAPS looks up the Chinese address information in another file. This file consists of over 20,000 records, each of which contains the Chinese address of the telegram recipient in terms of Chinese character internal codes. Each one-word internal code is then translated into its binary graphic pattern. A group of these binary graphic patterns, lead by a graphic control code, are then sent down to a graphic printer which will print the Chinese address information including the company's full name and address in front of the English (ASCII) telegrams (see Fig. 3).

For those telegrams routed to the domestic network, the city name on the telegrams will be verified against the city-name file. It will be routed to the respective line based on the information from the file. Local Teletype terminals include "Full Address" positions, "Service Telegrams" positions, "Inter-office Communication" positions, and the "Intercept" positions for the abnormal telegrams that require manual assistance.

Automatic Editing and Routing of Outgoing Telegrams

For the outgoing telegrams, the ITAPS accepts the telegrams from the following four major sources: (1) ITA's branch offices can send telegrams either by Teletype keyboard/paper tape reader, or by remote CRTs; (2) Telex subscribers can send public telegrams using simple format; (3) ITA's Telegram Operation Center can send telegrams either by Teletypes keyboard/paper tape reader, or by local CRTs; (4) domestic network can handover its international outgoing telegrams to ITAPS.

Telex subscribers can dial up "923" requesting a direct connection to the ITAPS through the microprocessor interface. If there is buffer available in the Communication Processor, the ITAPS, after obtaining subscriber's ID, will send "GO AHEAD CABLE". The subscriber then send his/her prepared paper tape or type in telegrams. After receiving End of Message (NNNN), the ITAPS will again verify the same ID to make sure that the same circuit has been connected throughout the entire period of telegram transmission. If a match occurred in the verification, the sending subscriber is then given a receipt number on which a later inquiry of the telegram may be made. For the convenience of the customers, the telegrams sent by the Telex subscribers are in simple format. The ITAPS will edit the simple format into the CCITT F.31 format by automatically filling in the numbering line, pilot line, word count, destination indicator, and origin indicator, etc. to become an internationally compatible interchange format. Based on the destination indicators or geographical indicator, the telegram is then routed onto the required destination international trunks.

For those telegrams sent from ITA's branch offices, there is no dialing-up procedure needed, instead, the prepared paper tape can be sent directly from Teletype paper tape reader or keyboard into the ITAPS, or telegrams can be edited on the CRT screen and sent to ITAPS by a single key action. The remote CRTs are connected to the Communication Processor through modems using asynchronous RS-232C protocol at 1200 bauds. Telegrams may also be input to the ITAPS using local CRTs which are directly connected to the Message Switching Processor using asynchronous protocol at 9600 bauds.

Journaling and Retrieval of Telegrams

Telegrams input/output into/from ITAPS are properly recorded or journaled into the short-term input/output journal file in the moving head disks. Input journal file contains telegrams that are originally input into the system with their arrival time stamps and system numbers from which the respective telegram can be retrieved. Output journal file contains telegrams that may have been automatically edited into the F.31 format or manually corrected some erroneous fields in the message header, together with their leaving time stamps, system numbers and other information extracted from the telegram header. For retrieval and report-printing purpose, many inverted files are built at the time of output journaling such as DELINV (delivery number), ICPINV/OCPINV (Input/Output Circuit Prefix), TIMINV (Time), TIGINV (Telegram ID Group) to facilitate multi-directional retrieval from other keys. The on-line retrieval commands can be entered from 5-unit Teletypes locally and remotely, and from local CRTs.

Two 80-megbyte disks are installed to allow telegrams in two days to be journaled, while magnetic tapes are used to transcribe telegrams for long-term filing. The retrieval of telegrams from magnetic tape can be made off-line.

System Switchover and Recovery

ITAPS is designed to have dual configuration. During normal operation one system serves as on-line and the other as hot-standby. The on-line system does all work including telegram reception, assembly, storing, analysis, routing, dispatching and disassembly while the standby system does only the telegram reception and assembly. The on-line message processor continously sends information to the standby message processor ordering it to release those buffers whose contents have been safely written (stored) into the transit storage by the on-line Message Processor. Also, in the on-line system, a snapshot program periodically saves system operational data and tables onto the drum (head-per-track) snapshot area including the current queue transactions, data and tables in common area. If the on-line system fails, or either side receives a switchover command, the hot-standby system will immediately loads the last snapshot area into its core memory, changes its own processor state to on-line, and then take-over the on-line task without having to load the on-line programs from the system disk. During normal operation, the same set of real-time programs are running or stationary both in the on-line and in the standby system respectively, but the input processing program is running in on-line state or in standby state depending on whether the respective processor is in which state. This arrangement allows fast switchover action to be taken place.

After the switchover, the standby processor backs up its processing starting from the last snapshot of the system which preserves all the necessary information to start over from the last mile-stone record. Such arrangement will guarantee that no telegram message or character will be lost during the switchover transition.

The system can also be restarted using a restart procedure LJ(RESTART) to recover all the necessary data which have been saved in the snapshot area.

A Chinese Computer System for File Building

One of the requirements of ITAPS is to print Chinese address information on the received telegrams to facilitate messenger's delivery. This requirement motivated the invention of a Chinese computer system for file building purpose. This system uses an ordinary graphic CRT terminal to input and display the selected Chinese characters. This is accomplished by assigning key positions for Chinese character roots and any Chinese character can be defined according to the normal writing sequence as a one-dimensional spelling sequence of its constituent roots. Such a system can be built in a general purpose computer system as part of the file handling process. Besides building cable address file with Chinese address information, the system can be used for general purpose Chinese information storage and retrieval purposes. The special characteristics of the input method are described as follows:

1. Use ordinary small CRT keyboard without special interface,
2. The number of roots approaches theoretical optimum value, which means minimum average key strokes, speedy operation and high uniqueness of the selected characters.
3. The arrangement of roots is on the one hand according to the statistical occurent frequency of roots, which makes the average operation speed faster; and on the other hand according to the connotational meaning of root to key-position alphabets, thus to facilitate beginners' memorizing.
4. Spelling sequence can be defined dynamically by users, thus make the selection operation more flexible and multi-directional, for example, normal character being selected from the abbreviated writing sequence; multiplication of a number and a root to denote the repetition of the same root; subtraction of roots can be performed for similar roots.
5. Processing program is very simple, within 2K words, and the additions of spelling sequences and character patterns are independant of the processing program.
6. It may also be used for English, Japanese, Korean and other ideographic languages.

Local and Remote CRTs

Local positions are installed at the Telegraph Operation Center to manually assist the system for the handling of the intercepted telegrams either having format, routing, spelling errors, or unidentifiable name or field in the telegram header, owing to which the telegrams cannot be properly routed or delivered. After human intervention for the proper correction, these intercepted telegrams will be routed to their proper destination or be diverted to a specified printer for further analysis.

The functions of the local CRTs are categorized by pressing the different function keys. These functions keys are built-in to each CRT in the right hand neighboring of the normal keyboard area. Each of them can trigger a pre-defined process in the CRT Processing Module.

Another block of key area further right hand side of the special function keys allows operator to edit the telegram in a page mode. By "page" mode, it means that the purchased CRTs have a local buffer and the limited intelligence to allow operator editing telegrams locally or without intervening the host computer, thus leaving the host computer with more CPU time to process other on-line tasks.

The remote CRTs have limited functional capabilities. The use of function keys are limited to send out a newly-edited telegram and to log-in and log-off.

Operational Commands and Reports

Various user's designed commands can be entered at the console CRTs, local CRTs, and the remote Teletype positions either to control or regulate the system operation, or to obtain the current operational status of the system. Commands entered from console CRTs are honored by the Executive of the Operating System while those entered from local CRTs and the 5-unit-code TTYs are interpreted and executed by a user's designed command interpreter and its associated subroutines working in the foreground environment.

The command entered from the remote TTY positions resemble those used in the network access operation because they are relayed by the Communication Processor to the Message Switching Processor for the proper responses. Each command response must be returned to the respective TTY that has issued the command. The technique used is in fact a "packet switching" type transmission of both command and the response. These commands can be used to obain operational reports or retrieve telegrams.

Software Design and the Parallel Running Result

The design of the application software for ITAPS uses the top-down and modular concepts [4]. Each module has its pre-defined functions and the related mudules have their interfaces. In Message Switching Processor, there are Dispatching Module, Input Processing Module, Message Analysis and Route Selection Module, Output Processing Module, CRT Processing Module, Telex Editing Module, Journaling Module, and Command Processing Module. These modules are assigned different priority level according to the degree of urgency of each module. The main interfaces are previded using the "adminnistrative block" appended to the first sector of each telegram in the transit storage, and the interface tables in high core common area. Transactions are passed around in core memory using the self-implemented queue manager. There are two types of queue, one is the cyclic type FIFO queue and the other is the multi-line multi-priority queue for output processing module. The same approach is also carried on in the Communication Processor in which three receiving modules and three transmitting modules are implemented except without using the drum transit storage as the interface among modules, instead, the packet buffers being used as such. The receiving modules include Input Interrupt Handling Routing, Input Processing Program, and SDLC Output Interrupt Handling Routine. The transmitting modules include SDLC Input Interrupt Handling Routing, Output Processing Program, and the Output Interrupt Handling Routines (see Fig. 4).

The software for the communication lines, such as the SDLC, CRT multiplexor, low speed multiplexor, and RS-232C multiplexor, is implemented inside the Input Output System (IOS) as Handling Routines (Handlers) which is device dependant and user-oriented portion of programs linking closely to the respective drivers.

ITAPS has passed various phases of testing including the modular testing, integrating testing, functional verification testing, and the stability testing. Up to now (June 1981), the system has been running as the parallel running with the manual processing system for more than ten weeks. During the testing period, the real traffic as well as the simulated traffic are both applied to the system. For the modular testing and the integrating testing, the simulated traffic helped prove the correctness of the normal processing path of each module, while for the functional verification testing, stability testing and parallel running, the real traffic helped prove the correct treatment of the abnormal cases. The overall availability of the system within the parallel running period is above 99.9%.

This system is one of the large scale software development project in this country. This is one way of achieving self-reliance in the brain-intensive industry in this country.

References

[1] Philips, Message Data Switching System -- DS 714, Engineering Consideration, 1973.

[2] Rockwell International, C900/180 Message Switching System, Product Description, 1974.

[3] Kokusai Denshin Denwa, Co. Ltd., The Telegraph Automation System -- TAS, Mar. 1973.

[4] James Martin, Programming Real-time Computer System, Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1973.

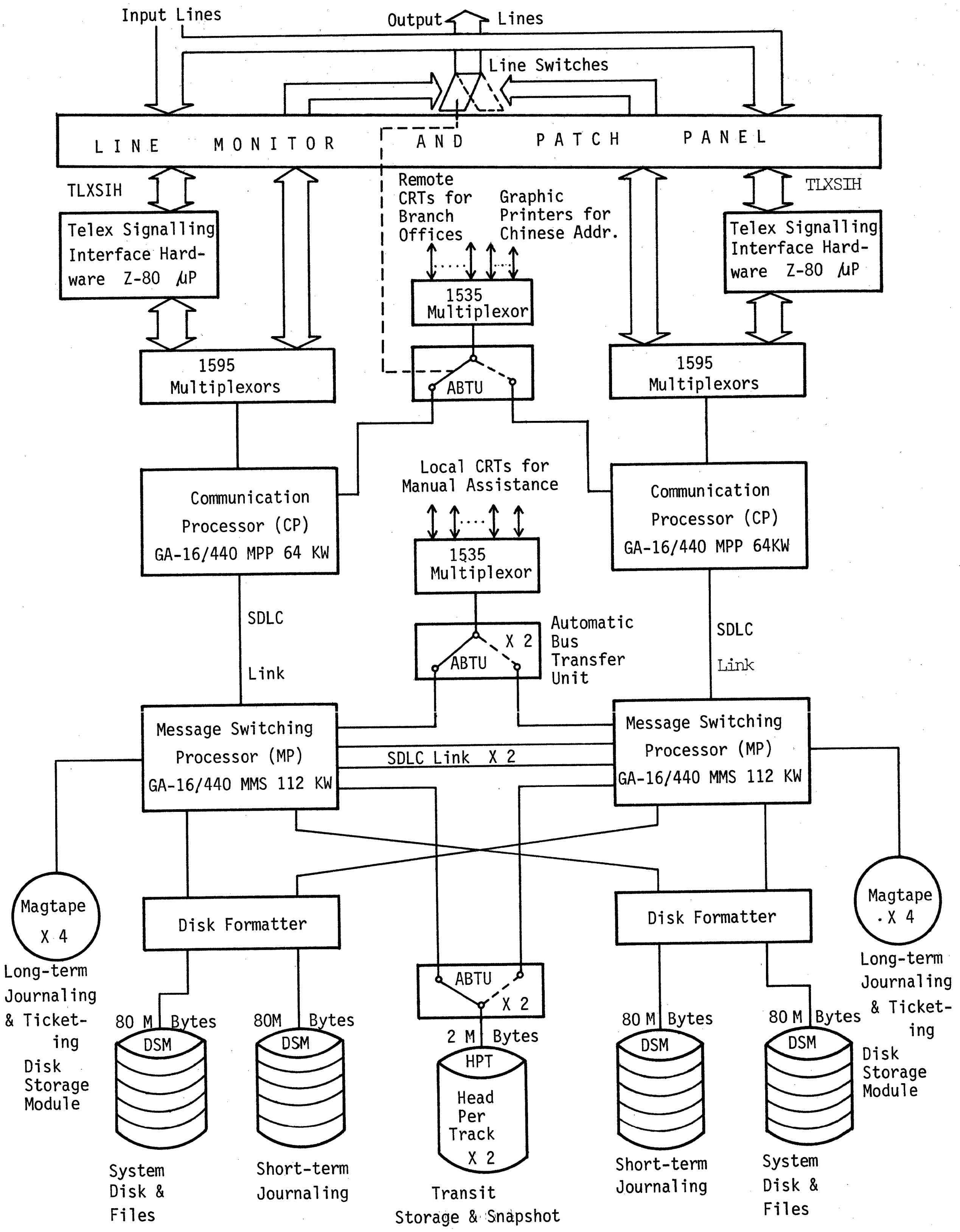

Input Lines
Output Lines
Line Switches
LINE MONITOR AND PATCH PANEL
TLXSIH
Telex Signalling Interface Hardware Z-80 µP
TLXSIH
Telex Signalling Interface Hardware Z-80 µP
Remote CRTs for Branch Offices
Graphic Printers for Chinese Addr.
1535 Multiplexor
ABTU
1595 Multiplexors
1595 Multiplexors
Local CRTs for Manual Assistance
Communication Processor (CP) GA-16/440 MPP 64 KW
Communication Processor (CP) GA-16/440 MPP 64KW
1535 Multiplexor
ABTU
X 2
Automatic Bus Transfer Unit
SDLC
Link
SDLC
Link
Message Switching Processor (MP) GA-16/440 MMS 112 KW
SDLC Link X 2
Message Switching Processor (MP) GA-16/440 MMS 112 KW
Magtape X 4
Long-term Journaling & Ticketing
Disk Formatter
Disk Formatter
Magtape X 4
Long-term Journaling & Ticketing
ABTU
X 2
80 M Bytes
DSM
80M Bytes
DSM
2 M Bytes
HPT
Head Per Track X 2
80 M Bytes
DSM
80 M Bytes
DSM
Disk Storage Module
Disk Storage Module
System Disk & Files
Short-term Journaling
Transit Storage & Snapshot
Short-term Journaling
System Disk & Files

Fig. 1. Hardware Configuration of the ITAPS.

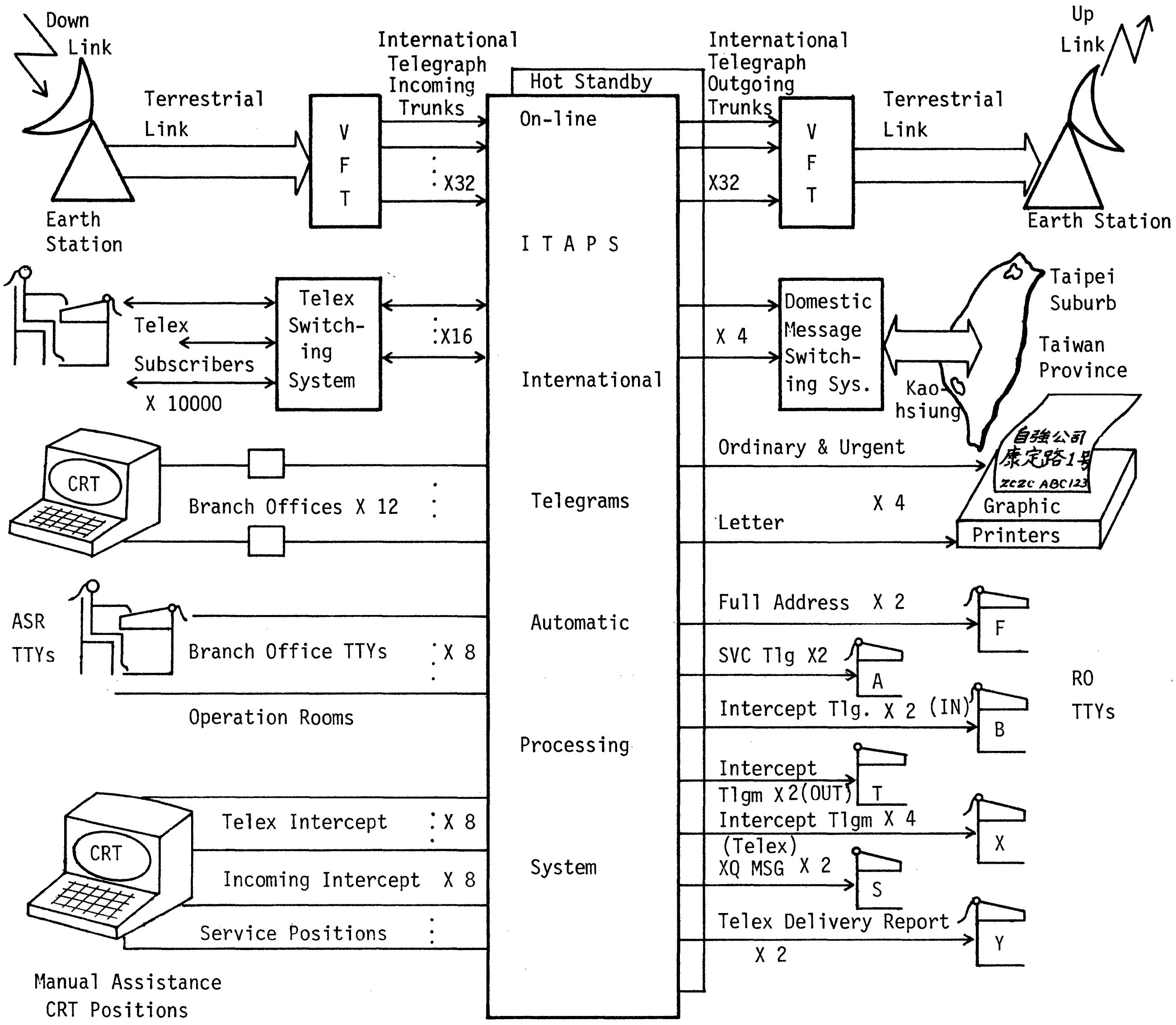
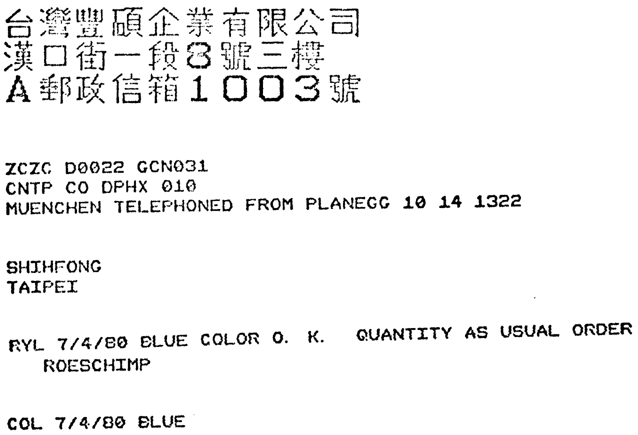


Fig. 2 Functional Block Diagram of ITAPS.



台灣豐碩企業有限公司
漢口街一段8號三樓
A郵政信箱1003號

```
ZCZC D0022 GCN031
CNTP CO DPHX 010
MUENCHEN TELEPHONED FROM PLANEGG 10 14 1322


SHIHFONG
TAIPEI


RYL 7/4/80 BLUE COLOR O. K.   QUANTITY AS USUAL ORDER
   ROESCHIMP


COL 7/4/80 BLUE
```

Fig. 3. Chinese Address Information Is Automatically Printed on the Received International Telegrams to Facilitate Messenger's Delivery.

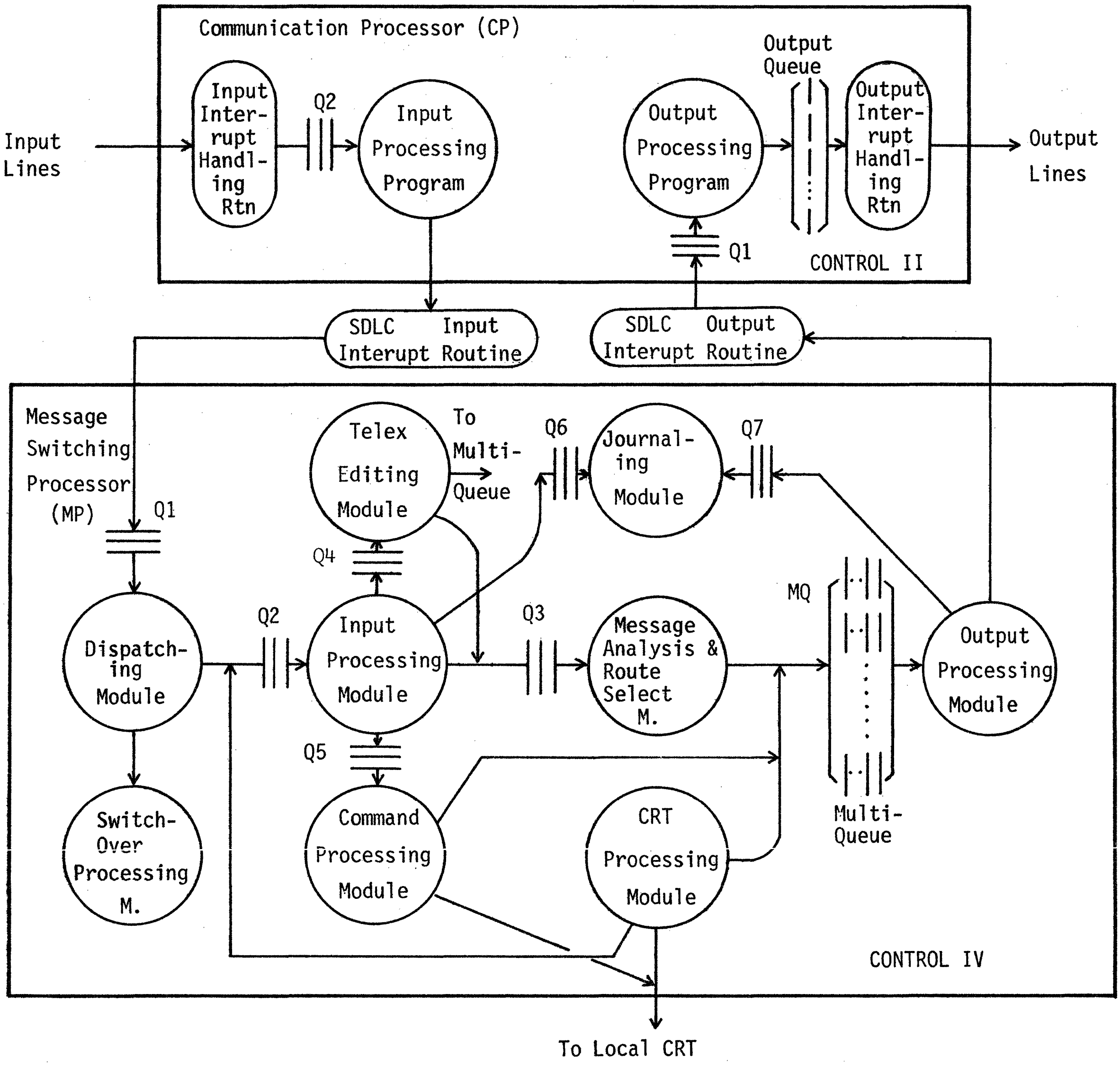

Communication Processor (CP)
Input Lines
Input Inter-rupt Handl-ing Rtn
Q2
Input Processing Program
Output Processing Program
Output Queue
Output Inter-rupt Handl-ing Rtn
Output Lines
Q1
CONTROL II
SDLC Input Interupt Routine
SDLC Output Interupt Routine
Message Switching Processor (MP)
Q1
Telex Editing Module
To Multi-Queue
Q6
Journal-ing Module
Q7
Q4
MQ
Dispatch-ing Module
Q2
Input Processing Module
Q3
Message Analysis & Route Select M.
Output Processing Module
Q5
Switch-Over Processing M.
Command Processing Module
CRT Processing Module
Multi-Queue
CONTROL IV
To Local CRT

Fig. 4. Software Configuration of ITAPS.

MULTITERMINAL RELIABILITY ANALYSIS OF DISTRIBUTED PROCESSING SYSTEMS*

Aksenti Grnarov and Mario Gerla
Computer Science Department
University of California
Los Angeles, California 90024

Abstract -- Distributed processing system reliability has been measured in the past in terms of point-to-point terminal reliability, or more recently, in terms of the 'survivability index' or 'team behavior.' While the first approach leads to oversimplified models, the latter approaches imply excessive computational effort. A novel, computationally more attractive measure based on multiterminal reliability is proposed. The measure is the probability of true value of a Boolean expression whose terms denote the existence of connections between subsets of resources. The expression is relatively straightforward to derive, and reflects fairly accurately the survivability of distributed systems with redundant processor, data base and communications resources. Moreover, the probability of such Boolean expression to be true can be computed using a very efficient algorithm. This paper describes the algorithm in some detail, and applies it to the reliability evaluation of a simple distributed file system.

1. Introduction

Distributed processing has become increasingly popular in recent years, mainly because of the advancement in computer network technology and the falling cost of hardware, particularly of microprocessors. Intrinsic advantages of distributed processing include high throughput due to parallel operation, modular growth, fault resilience and load leveling.

In a distributed processing system (DPS), computing facilities and communications subnetwork are interdependent of each other. Therefore, a failure of a particular DPS computer site will have a negative effect on the overall DP system. Similarly, failure of the communication subsystem will lead to overall performance degradation.

Recently, considerable attempts have been made to systematically investigate the survival attributes of distributed processing systems which are subject to failures or losses of processing or communication components. Two main approaches to DPS survivability evaluation have emerged:

a) In [MER 80] the term *survivability index* is used as a performance parameter of a DDP (distributed data processing) system. An objective function is defined to provide a measure of survivability in terms of node and link failure probabilities, data file distribution, and weighting factors for network nodes and computer programs. This objective function allows the comparison of alternative data file distributions and network architectures. Criteria can be included such as the addition or deletion of communication links, allocation of programs to nodes, duplication of data sets, etc. Constraints can be introduced which limit the number and size of files and programs that can be stored at a node. The main disadvantage of the survivability index is its computational complexity, which makes it practical only to DDP systems with, say, less than 20 nodes or links.

b) The second approach is a 'team' approach in which the overall system performance is related to both the operability and the communication connectivity of its 'member' components [HIL 80]. The performance index, defined axiomatically on the connectivity state space of the graph, captures the essentials of the 'team effect' and allows survivability cost/performance trade-offs of alternate network architectures. The basic advantage of the team approach is that performance degradation beyond the connected/disconnected state is measured. One disadvantage of the approach is that of being restricted to the homogeneous case and of ignoring other important details of real DPS's.

In this paper we propose a novel measure of DPS survivability, namely *multiterminal reliability*. We recall that in a communications network terminal reliability relative to node pair (i,j) is the probability that node i is connected to node j. We extend this notion to DPS's by defining the multiterminal reliability as follows:

Definition 1. The multiterminal reliability of a DPS consisting of a set of nodes (processors) V=1,2,...,N is defined as

$$P_s = Prob\ C_{I_1,J_1} \oplus_1 C_{I_2,J_2} \oplus_2 \quad \oplus_{k-1} C_{I_k,J_k} \tag{1}$$

where:

$I_1,J_1,I_2,J_2,...,I_k,J_k$ are subsets of V

C_{I_j,J_j} denotes the existence of connections between all the nodes of the subset I_j and all the nodes of subset J_j

and

$\oplus_j$ has a meaning of OR or AND.

The choice of the subsets $I_1,J_1,...,I_k,J_k$ as well as the interpretation of the operator $\oplus_j$ $(j = 1, \cdots, K-1)$ depend on the event (task) whose survivability is being evaluated. Priority between operators is determined by parentheses in the same way as in standard logical expressions.

* This research was supported by the Office of Naval Research under contract N00014-79-C-0866. Aksenti Grnarov is currently on leave from the University of Skopje, Yugoslavia.

0190-3918/81/0000/0079$00.75 © 1981 IEEE

As an example, let us assume that the successful completion of a given task requires node A to communicate with node B *or* node C; and node D *and* E to communicate with node F *and* G. The multiterminal reliability of such task is given by

$$P_m = Prob\ (C_{I_1,J_1}\ OR\ C_{I_1,J_2})\ AND\ C_{I_3,J_3}$$

where $I_1 = \{A\}$, $J_1 = \{B\}$, $J_2 = \{C\}$, $I_3 = \{D,E\}$ and $J_3 = \{F,G\}$.

The general definition of multiterminal reliability can be specialized to characterize the survivability of the following systems:

(A) Distributed Data Base System: For given link and computer center reliabilities, determine the reliability of a specific file allocation including redundant copies.

(B) Teamwork: Given link and processing node reliability, determine what distribution of the members will result in highest probability of a connection.

(C) Distributed Data Processing System: Given link and processing node reliability and (redundant) distribution of programs and data, determine the probability of successfully completing a specific application.

(D) Computer-Communication Network: Given link and node reliability, determine the probability of the network becoming partitioned.

Note that in all the above applications, system (or application) survivability is best characterized by some multiterminal reliability measure. In fact, terminal reliabilities alone could not be used to compute systems survivability because of the dependencies existing between the various events.

In this paper, an efficient algorithm for multiterminal reliability analysis is presented. The algorithm can be applied to oriented and non-oriented graph models of DPS's and can produce numerical results as well as symbolic reliability expressions.

The paper is organized in five sections. In Section 2, the application of Boolean algebra to multiterminal reliability is considered. Derivation of the algorithm is presented in Section 3. An example for determination of the multiterminal reliability is given in Section 4 . Some comments and concluding remarks are presented in the final section.

2. Boolean Algebra Approach

For reliability analysis a DPS is usually represented by a probabilistic graph $G(V,E)$ where V=1,2,...,N and $E = a_1 a_2,...,a_E$ are respectively the set of nodes (representing the processing nodes) and the set of directed or undirected arcs representing the communication links. To every DPS component i (processing node or link), a stochastic variable y_i can be associated. The weight assigned to the i^{th} component represents the component reliability

$$p_i = Pr(y_i = 1)$$

i.e., the probability of the existence of the i^{th} component. Variables are supposed to be statistically independent.

There are two basic approaches for computing terminal reliability [FRA 74]. The first approach considers elementary events and the terminal reliability of a connection from source s to termination t, by definition, is given by

$$P_{st} = \sum_{F(e)=1} P_e$$

where P_e is probability which corresponds to the event e and $F(e)$=1 means that the event is favorable, i.e., it includes a path from s to t.

The second approach considers larger events corresponding to the simple paths between terminal nodes. These events however are no longer disjoint and the terminal reliability is given by the probability of the union of the events corresponding to the existence of the paths.

The complexity of these approaches is caused in the first case by the large number of elementary events (of the order 2^n where n = the number of elements which can fail) and in the second case by the difficult computation of the sum of the probabilities of nondisjoint events (the number of joint probabilities to be computed is of the order 2^m where m = the number of paths between node pairs).

Fratta and Montanari [FRA 74] chose to represent the connection between nodes s and t by a Boolean function. This Boolean function is defined in such a way that a value of 0 or 1 is associated with each event according to whether or not it is favorable (i.e., the connection $C_{s,t}$ exists). Since the Boolean function corresponding to the connection $C_{s,t}$ is unique, this means that the connection $C_{s,t}$ can be completely defined by its Boolean function. Representing a connection by its Boolean function, the problem of terminal reliability can be stated as follows: Given a Boolean function F_{ST}, find a minimal covering consisting of nonoverlapping implicants. Once the desired Boolean form is obtained, the arithmetic expression giving the terminal reliability is computed by means of the following correspondences

$$x_i \rightarrow p_i$$

$$\bar{x}_i \rightarrow q_i = 1 - p_i$$

Boolean sum → arithmetic sum
Boolean product → arithmetic product

A drawback of the algorithms based on the manipulation of implicants is the iterative application of certain Boolean operations and the fact that the Boolean function changes at every step (and may be clumsy). The Boolean function may be simplified using one of the following techniques: absorption law, prime implicant form, irredundant form or minimal form. Any one of these procedures however requires a considerable computational effort. Therefore, it can be concluded that these algorithms are applicable only to networks of small size.

Recently, efficient algorithms based on the application of Boolean algebra to terminal reliability computation and symbolic reliability analysis were proposed in [GRN 79] and [GRN 80a] respectively. The algorithms are based on the representation of simple paths by 'cubes' (instead of prime implicants), on the definition of a new operation for manipulating the cubes, and on the interpretation of resulting cubes in such a way that Boolean and arithmetic reduction are combined.

The proposed algorithm for multiterminal reliability analysis is based on the derivation of a Boolean function for multiterminal connectivity and the extension of the algorithm presented in [GRN 80b] to handle both multiterminal reliability computation and symbolic multiterminal reliability analysis.

3. Derivation of the Algorithm

Before presenting the algorithm for multiterminal reliability analysis, it is useful to recall the definition of the path identifier from [GRN 79]:

Definition 2. The path identifier IP_k for the path π_k is defined as a string of n binary variables

$$IP_k = x_1 x_2 ... x_i ... x_n$$

where

$x_i=1$ if the i^{th} component of the DPS is included in the path π_k

$x_i = x$ otherwise

and n is the number of DPS components that can fail, i.e:

$n = N$ in the case of perfect links and imperfect nodes

$n = E$ in the case of perfect nodes and imperfect links

$n = N+E$ in the case of imperfect links and nodes.

As an example, let us consider a four node, five link DPS given in Figure 1, in which nodes are perfectly reliable and links are subject to failures. The sets of path identifiers for the connections $C_{S,A}$ and $C_{S,T}$ are given in Table 1 and Table 2 respectively.

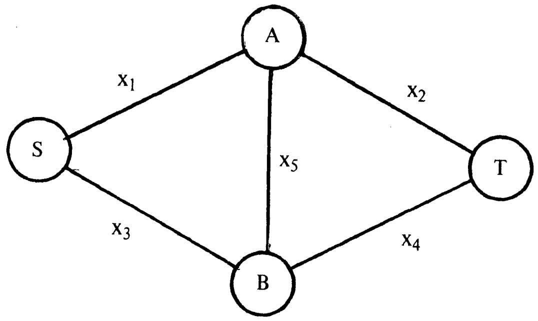


TABLE 1

PATH	IP
S x_1A	1xxxx
S x_3B x_5A	xx1x1
S x_3B x_4T x_2A	x111x

TABLE 2

PATH	IP
S x_1A x_2T	11xxx
S x_1A x_5B x_4T	1xx11
S x_3B x_4T	xx11x
S x_3B x_5A x_2T	x11x1

Figure 1. Example of DPS

Boolean functions corresponding to $C_{S,A}$ and $C_{S,T}$ given by their Karnaugh maps, are shown in Figure 2.

$x_5 = 0$

x_3x_4 \ x_1x_2	00	01	11	10
00			1	1
01			1	1
11		1	1	1
10			1	1

$x_5 = 1$

x_3x_4 \ x_1x_2	00	01	11	10
00			1	1
01			1	1
11	1	1	1	1
10	1	1	1	1

$F_{s,a}$

$x_5 = 0$

x_3x_4 \ x_1x_2	00	01	11	10
00			1	
01			1	
11	1	1	1	1
10			1	

$x_5 = 1$

x_3x_4 \ x_1x_2	00	01	11	10
00			1	
01			1	1
11	1	1	1	1
10		1	1	

$F_{s,t}$

Figure 2. Karnaugh Map Representation of the Connections $C_{S,A}$ and $C_{S,T}$

Instead of the cumbersome determination of elementary (or composite) events which correspond to a multiterminal connection, the multiterminal reliability can be determined from the Boolean function representing the connection. Moreover, the corresponding Boolean function can be obtained from path identifiers (Boolean functions) representing terminal connections. For example, the Boolean function corresponding to the multiterminal connection

$$C_{mor} = C_{S,A} \ OR \ C_{S,T}$$

can be obtained as

$$F_{mor} = F_{S,A} \ U \ F_{S,T}$$

where U is the logical operation union. Karnaugh map of F_{mor} is shown in Figure 3.

$x_5 = 0$

x_3x_4 \ x_1x_2	00	01	11	10
00			1	1
01			1	1
11	1	1	1	1
10			1	1

$x_5 = 1$

x_3x_4 \ x_1x_2	00	01	11	10
00			1	1
01			1	1
11	1	1	1	1
10	1	1	1	1

$F_{s,a}$

Figure 3. Karnaugh map representation of the connection $C_{mor} = C_{S,A}$ OR $C_{S,T}$

Covering the Karnaugh map with disjoint cubes, we can obtain F_{mor} as

$$F_{mor} = x_1 + \bar{x}_1x_3x_5 + \bar{x}_1x_3x_4\bar{x}_5$$

i.e., multiterminal reliability is given by

$$P_{mor} = p_1 + q_1p_3p_5 + q_1p_3p_4q_5$$

Analogously, the Boolean function corresponding to the multiterminal connection

$$C_{mand} = C_{S,A} \ AND \ C_{S,T}$$

can be obtained as

$$F_{mand} - F_{S,A} \wedge F_{S,T}$$

where Λ is the logical operation intersection.

According to the Karnaugh map representation (Figure 4), F_{mand} is given by

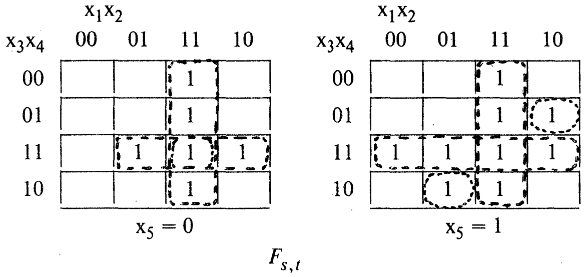


Figure 4. Karnaugh Map Representation of the connection $C_{mand} = C_{s,a}$ AND $C_{s,t}$

the following set of cubes

$$IP = (11xxx, xx111, 1xx11, x11x1, x111x, 1x11x)$$

Applying the algorithm REL [GRN 80b] we obtain that the multiterminal reliability is given by

$$P_{mand} = p_1p_2 + p_3p_4p_5(1-p_1p_2)$$
$$+ p_1p_4p_5q_2q_3 + q_1p_2p_3q_4p_5 + q_1p_2p_3p_4$$

Since the logical operations union and intersection satisfy the commutative and associative laws, previous results can be generalized as follows.

1) Multiterminal connection of OR type $C_{s,T}$ $(T=t_1, t_2, \cdots, t_k)$ is equal to

$$C_{s,T} = C_{s,t_1}\ OR\ C_{s,t_2}\ OR\ \cdots\ OR\ C_{s,t_k}$$

and the corresponding Boolean function $F_{s,T}$ can be obtained as

$$F_{s,T} = F_{s,t_1} \cup F_{s,t_2} \cup \cdots \cup F_{s,t_k}$$

2) Multiterminal connection of AND type $C_{s,T}$ $(T = \{t_1, t_2, \ldots, t_k\})$ is equal to

$$C_{s,T} = C_{s,t_1}\ AND\ C_{s,t_2}\ AND \cdots AND\ C_{s,t_k}$$

and the corresponding Boolean function $F_{s,T}$ can be obtained as

$$F_{s,T} = F_{s,t_1} \Lambda F_{s,t_2} \Lambda \ldots . \Lambda F_{s,t_k}$$

In the case when all nodes from the set S have connections of the same type with all nodes from the set T, multiterminal connection can be written as $C_{S,T}$.

4. Determination of $F_{S,T}$

The determination of $F_{S,T}$ by Boolean expression manipulation or by determination of elementary events is a cumbersome and time consuming task. Hence, these methods are limited to DPS's that are very small in size.

However, since path identifiers can be interpreted as cubes, the Boolean function $F_{S,T}$ can be more efficiently obtained by manipulating path identifiers. In the sequel we present the OR-Algorithm and the AND-Algorithm for the determination of $F_{s,T}$ of type OR and AND respectively. Both algorithms are based on the application of the intersection operation [MIL 65]. Since the path identifiers have only symbols x and 1 as components, the intersection operation can be modified as follows:

Definition 3: The intersection operation between two cubes, say $c^r = a_1a_2 \cdots a_i \cdots a_n$ and $c^s = b_1b_2 \cdots b_i \cdots b_n$, is defined as

$$c^r \Lambda c^s = [(a_1 \Lambda b_1),(a_2 \Lambda b_2), \ldots, (a_i \Lambda b_i), \ldots, (a_n \Lambda b_n)]$$

where the coordinate Λ operation is given by

Λ	1	x
1	1	1
x	1	x

It can be seen that the intersection operation between two cubes c^r and c^s produce a cube which is common to both c^r and c^s. If $c^r \Lambda c^s = c^r$ this means that the cube c^r is completely included in the cube c^s. The modified intersection operation produces a cube which has only symbols x and 1 as coordinates, so the modified intersection operation can be applied again and again. Also, the previous fact allows us to apply the REL-Algorithm on the set of cubes obtained by the application of the modified intersection operation.

Let us suppose that the cubes corresponding to connections C_{s_1,T_1} and C_{s_2,T_2} are stored in lists L_1 and L_2 of length k_1 and k_2 respectively. Let c_i^j denote the j^{th} element of the list L_i.

The OR-Algorithm for the computation of $F_{S,T}$ follows:

OR-Algorithm

```
STEP 1.

    for $i$ from 1 to $k_1$ do
        for $j$ from 1 to $k_2$ do
            begin
                $c = c_1^i \Lambda c_2^j$ ; if c = $c_1^i$ then
                    begin
                        delete $c_1^i$ from list $L_1$ ;
                    end
                else if c = $c_2^j$ then delete $c_2^j$ from list $L_2$ ;
            end
STEP 2.
    Store undeleted elements from the lists $L_1$ and $L_2$ as new list
    $L_1$

END
```

As an example, the OR - algorithm is applied to the determination of $F_{s,T} = F_{s,a} \cup F_{s,t}$ for the DPS given in Figure 1. The lists L_1 and L_2 are

	L_1		L_2
c_1^1	1xxxx	c_2^1	11xxx
c_1^2	xx1x1	c_2^2	1xx11
c_1^3	x111x	c_2^3	xx11x
		c_2^4	x11x1

STEP 1:

STEP 1: $c_1^1 \Lambda c_2^1 = c_2^1$ *delete* c_2^1

$c_1^1 \Lambda c_2^2 = c_2^2$ *delete* c_2^2

$c_1^1 \Lambda c_2^3 \neq c_1^1 \neq c_2^3$

$c_1^1 \Lambda c_2^4 \neq c_1^1 \neq c_2^4$

$c_1^2 \Lambda c_2^3 \neq c_1^2 \neq c_2^3$

$c_1^2 \Lambda c_2^4 = c_2^4$ *delete* c_2^4

$c_1^3 \Lambda c_2^3 = c_1^3$ *delete* c_1^3

STEP 2:

	L_1
c_1^1	1xxxx
c_1^2	xx1x1
c_1^3	xx11x

It can be seen that the OR - Algorithm produces a list with minimal number of elements which are cubes of the largest possible size. The same result could have been obtained from the identification of disjoint cubes directly in Fig. 3. Our method allows for the efficient generation of all disjoint cubes necessary for reliability analysis [GRN 79]. Next, we introduce the AND algorithm.

AND - Algorithm

```
STEP 1.
    for $i$ from 1 to $k_1$ do
        begin
            for $j$ from 1 to $k_2$ do

                $c_{i+2}^j = c_1^i \Lambda c_2^j$ ;
                  for $k$ from 1 to $k_2$-1 do
                      begin
                          m = k+1
                          while   $c_{i+2}^k \neq c_{i+2}^k \Lambda c_{i+2}^m$   and
                          $m \leqslant k_2$ do
                              begin
                                  c = $c_{i+2}^k \Lambda c_{i+2}^m$ ;
                                  if c = $c_2^m$ then delete $c_i^m$
                                  from list $L_i$
                                  m = m+1
                              end
                            if $m \leqslant k_2$ then delete $c_{i+2}^k$ from
                            list $L_{i+2}$
                          end
                  end
    STEP 2
        Store undeleted elements from lists $L_3, \ldots, L_{k_1+2}$
        as a new list $L_1$

END
```

As an example, the AND-Algorithm is applied to the determination of $F_{s,T} = F_{s,a} \Lambda F_{s,t}$ for the DPS in Fig. 1.

STEP 1.

i = 1

Step 1.1

$$L_3 = \begin{cases} c_3^1 = c_1^1 \Lambda c_2^1 = 11\text{xxx} \\ c_3^2 = c_1^1 \Lambda c_2^2 = 1\text{xx}11 \\ c_3^3 = c_1^1 \Lambda c_2^3 = 1\text{x}11\text{x} \\ c_3^4 = c_1^1 \Lambda c_2^4 = 111\text{x}1 \end{cases}$$

Step 1.2

$c_3^1 \Lambda c_3^2 \neq c_3^1 \neq c_3^2$

$c_3^1 \Lambda c_3^3 \neq c_3^1 \neq c_3^3$

$c_3^1 \Lambda c_3^4 = c_3^4$ *delete* c_3^4

$$L_3 = \begin{cases} 11\text{xxx} \\ 1\text{xx}11 \\ 1\text{x}11\text{x} \end{cases}$$

i = 2

Step 1.1

$$L_4 = \begin{cases} c_4^1 = 111\text{x}1 \\ c_4^2 = 1\text{x}111 \\ c_4^3 = \text{xx}111 \\ c_4^4 = \text{x}11\text{x}1 \end{cases}$$

Step 1.2

$$L_4 = \begin{cases} \text{xx}111 \\ \text{x}11\text{x}1 \end{cases}$$

i = 3

Step 1.1

$$L_5 = \begin{cases} 1111\text{x} \\ 11111 \\ \text{x}111\text{x} \\ \text{x}1111 \end{cases}$$

Step 1.2

$L_5 = \text{x}111\text{x}$

STEP 2.

	L_1
c_1^1	11xxx

c_1^2	1xx11
c_1^3	1x11x
c_1^4	xx111
c_1^5	x11x1
c_1^6	x111x

It can be seen that the AND - Algorithm also produces a list with minimal number of elements which are cubes of the largest possible size.

In the general case, the Boolean function corresponding to the connection $C_{S,T}$ where $S=\{s_1,s_2,...,s_k\}$ and $T=\{t_1,t_2,...,t_m\}$, can be obtained using the multiterminal Algorithm (m ⊕-Algorithm) described below:

m ⊕ - Algorithm

```
STEP 1:
Find the path identifiers for terminal connection $s_1,t_1$ and store them in the list $L_1$ ; $i \leftarrow 1$.

STEP 2:
Sort the path identifier in $L_1$ according to increasing number of symbols 1 (i.e. increasing path length);

STEP 3:
if $i \leqslant k$ continue. Otherwise go to step 5

STEP 4:
for $j = j_1,..., m$ ($j_1 = 2$ if $i = 1$, otherwise $j_1 = 1$)
        Step 4.1
        Find the path identifiers for terminal connection $s_i,t_j$ and store them in the list $L_2$
        Step 4.2
        Sort the path identifiers in $L_2$ according to the increasing number of symbols "1"
        Step 4.3
        Perform ⊕ -Algorithm on the lists $L_1$ and $L_2$
        Step 4.4
        $i \leqslant i+1$; go to step 2.

END
```

In the algorithm, ⊕ denotes OR or AND depending on the connection type. The sorting of the lists allows faster execution of the algorithm (starting with the largest cubes results in earlier deletion of covered cubes, i.e., faster reduction of the lists during the execution of Step 4.3).

Based on the previous results we can propose the following algorithm for multiterminal reliability analysis:

MUREL - Algorithm

```
STEP 1:
        Derive the multiterminal connection expression corresponding to the event which has to be analyzed.
STEP 2:
        Determine the Boolean function corresponding to the multiterminal connection by repetitive application of the m ⊕ - Algorithm.
STEP 3:
        Apply the REL -Algorithm to obtain the multiterminal reliability expression or value.
```

Regarding the computational complexity of the MUREL-Algorithm, the following observations can be made:

i) The ⊕ -Algorithm can be implemented using only logical operations which generally belong to the class of the fastest instructions in a computer system.

ii) The m ⊕ -Algorithm produces a minimal set of maximal cubes (i.e., minimal irredundant form of the Boolean function).

iii) The REL-Algorithm is the fastest algorithm for the determination of the reliability expression or for the reliability computation from the set of cubes (path identifiers).

From the above considerations we conclude that the proposed algorithm can be applied to DPS of significantly larger size than was possible with other existing techniques.

In the following section, the algorithm is illustrated with an application to a small distributed system.

5. Example of Application of the Algorithm

As an example of the application of the algorithm we compute the survivability index for the simple DPS system shown in Figure 5 (the example is taken from [MER 80]). Assignment of files and programs to nodes is shown in figure 5.

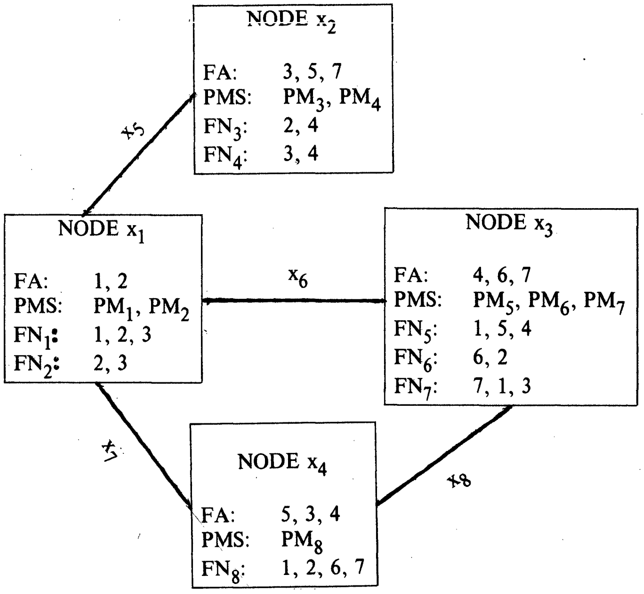


Figure 5. Four Node DDP

FA denotes the set of files available at a given node, FN_i denotes the files needed to execute program PM_i and PMS designates the set of programs to be executed at that node.

Let us assume that for a given application, we are interested in the survivability of program PM_3. Likewise, for another application, we need both programs PM_3 *and* PM_8 to be operational. We separately analyze these two cases using as a measure for survivability the multiterminal reliability (probability of program execution). The two problems can be stated as follows:

Given: node and link reliability, and file and program assignments to nodes.

Find: The survivability of:
1) Program PM_3

2) Both programs PM_3 and PM_8

Survivability of PM_3

The survivability PM_3 is equal to the multiterminal reliability of connection

$$C_{m3} = C_{2,I_1} \; OR \; C_{2,I_2}$$

where $I_1 = \{1,3\}$ and $I_2 = \{1,4\}$ The connections C_{2,I_1} and C_{2,I_2} are equal to

$$C_{2,I_1} = C_{2,1} \; AND \; C_{2,3}$$

$$C_{2,I_2} = C_{2,1} \; AND \; C_{2,4}$$

Paths and corresponding path identifiers for the connections $C_{2,1}$, $C_{2,3}$ and $C_{2,4}$ are shown in Figure 6.

$C_{2,1}$

paths	$F_{2,1}$
$x_1x_5x_2$	11xx1xxx

$C_{2,3}$

paths	$F_{2,3}$
$x_1x_5x_2x_6x_3$	111x11xx
$x_1x_5x_2x_7x_4x_8$	11x11x11

$C_{2,4}$

paths	$F_{2,4}$
$x_1x_5x_2x_7x_4$	11x11x1x
$x_1x_5x_2x_6x_3x_8x_4$	111111x1

Figure 6. Path and Path Identifiers Representing Connections $C_{2,1}$,$C_{2,3}$, and $C_{2,4}$

Applying the AND - Algorithm on $F_{2,1}$ and $F_{2,3}$, and $F_{2,1}$ and $F_{2,4}$ we obtain

F_{2,I_1}	F_{2,I_2}
111x11xx	11x11x1x
11x11x11	111111x1

Applying the OR - algorithm on F_{2,I_1} and F_{2,I_2} we obtain

F_{m3}

111x11xx
11x11x1x

Applying the REL - Algorithm on F_{m3} we obtain

$$P_{m3} = p_1p_2p_3p_4p_5p_6 + p_1p_2p_4p_5p_7(1 - p_3p_6)$$

Assuming $p_i = .95 \; \forall \; i$, we have: $p_{m3} = .85$

5.2. Survivability of both PM_3 and PM_8

The survivability of PM_8 is equal to the multiterminal reliability of connection

$$C_{m8} = C_{4,I_3}$$

where $I_3 = \{1,3\}$. The connection C_{4,I_3} is equal to

$$C_{4,I_3} = C_{4,1} \; AND \; C_{4,3}$$

Paths and corresponding path identifiers for the connections $C_{4,1}$ and $C_{4,3}$ are shown in Figure 7.

$C_{4,1}$

paths	$F_{4,1}$
$x_4x_7x_1$	1xx1xx1x
$x_4x_8x_3x_6x_1$	1x11x1x1

$C_{4,3}$

paths	$F_{4,3}$
$x_4x_8x_3$	xx11xxx1
$x_4x_7x_1x_6x_3$	1x11x11x

Figure 7. Paths and Path identifiers for Connections $C_{4,1}$ and $C_{4,3}$

Applying the AND - Algorithm on $F_{4,1}$ and $F_{4,3}$ we obtain

F_{m8}

1x11xx11
1x11x11x
1x11x1x1

Applying the AND - Algorithm on F_{m3} and F_{m8} we obtain

F_m

1111111x
111111x1
11111x11

Applying the REL - Algorithm on F_m we obtain

$$P_m = p_1p_2p_3p_4p_5p_6p_7 + p_1p_2p_3p_4p_5p_6q_7p_8 + p_1p_2p_3p_4p_5q_6p_7p_8$$

Assuming p_i=0.95 ∀i, we have: $p_m = 0.778$

6. Conclusion

In the paper, the multiterminal reliability is introduced as a measure of DPS survivability and the MUREL-Algorithm for multiterminal reliability analysis of DPS is proposed. First, the event under study is expressed in terms of its multiterminal connection. Then the m ⊕ -Algorithm is used to translate the multiterminal connection into a Boolean function involving all the relevant system components. Finally, the multiterminal reliability is obtained from the Boolean function by application of the REL-Algorithm.

Preliminary computational complexity considerations show that the MUREL-Algorithm permits the survivability analysis of DPS of considerably larger size than using currently available techniques.

References

[GRN 79] A. Grnarov, L. Kleinrock, M. Gerla, "A New Algorithm for Network Reliability Computation", *Computer Networking Symposium,* Gaithersburg, Maryland, December 1979.

[GRN 80a] A. Grnarov, L. Kleinrock, M. Gerla, "A New Algorithm for Symbolic Reliability Analysis of Computer Communication Networks", *Pacific Telecommunications Conference,* Honolulu, Hawaii, January 1980.

[GRN 80b] A. Grnarov, L. Kleinrock, M. Gerla, "A New Algorithm for Reliability Analysis of Computer Communication Networks", *UCLA Computer Science Quarterly,* Spring 1980.

[HIL 80] G. Hilborn, "Measures for Distributed Processing Network Survivability, *Proceedings of the 1980 National Computer Conference,* May 1980.

[MER 80] R. E. Merwin, M. Mirhakak, "Derivation and use of a survivability criterion for DDP systems", *Proceedings of the 1980 National Computer Conference,* May 1980.

[MIL 65] R. Miller, Switching Theory, Volume I: Combinational Circuits, New York, Wiley, 1965.

OPEN QUEUEING NETWORKS WITH FINITE CAPACITY QUEUES*

A. A. Nilsson and T. Altiok
North Carolina State University
Raleigh, North Carolina 27650

Abstract -- This paper discusses the problem of blocking in open exponential queueing networks. It is pointed out that such networks can be viewed as queueing network models of message-switched data communication networks with local flow- or congestion-control. Analysis is done by performing a node-by-node decomposition, and it is argued that an "off-line" analysis can be made, where the main problem is to analyze a single-server finite capacity queueing system with Markovian arrivals and a Coxian service time distribution. The method is applied to a number of example networks and evaluated by comparing the results obtained with those results obtained through exact analysis, simulation, or other approximate methods. We find that the method provides a good approximation procedure for obtaining system performance measures such as blocking-probabilities, throughput rates, etc.

Introduction

An open queueing network is a collection of nodes or servers that offer some form of service to customers in the network. A customer may enter the network at some node, receive service, and then immediately go to another node for additional service or he may leave the network. At any given time the number of customers in the network is a stochastic variable. In this paper we concentrate on exponential queueing networks where at any time the number of customers in a node may not exceed a certain number. This implies that customers currently not in that node and who want to go to that node are prohibited from doing so and will be held in their current nodes until the congestion is resolved. The interest in such a queueing network model was generated by an interest in gaining a better knowledge with regard to the influence of local flow control in a message- or packet-switched data communication network. Therefore, we prefer to present the detailed queueing network model through the terminology of data communication networks.

Flow control or congestion control in data communication networks are protocols that regulate the traffic flow input to the network or a switch node. The reason for introducing such control mechanisms is to try to minimize the impact of possible congestion and overflow due to the contention of a smaller number of resources by a large number of users [GERL 80]. Often flow control strategies are characterized as global control and local control. The global control refers to a control of the number of messages currently outstanding in the network between end users. The local control refers to a limit placed upon the number of messages currently residing in one node of the network. The impact of global flow control is fairly well understood, and there exists a number of excellent publications dealing with this subject, see the references in [GERL 80]. The local control strategy is much more difficult to analyze and very few results can be found in the existing literature. Some very useful results can however be found in [PENN 75]. We intend to present an approximate method that allows us to better understand the impact of a local flow control mechanism. For simplicity we will assume a network operating with a fixed routing algorithm. Consequently, it is possible to identify a number of fixed source-destination paths in the network. We will analyze a path consisting of M nodes that the messages have to pass through from the source to the destination. We will assume that the local control allows a maximum number N_i of messages in node i of our path.

The local control implies that a message arriving to the head of the queue in node i when there are N_{i+1} messages in node i+1, i.e., node i+1 is filled to its capacity, is blocked and has to wait until one of the messages in node i+1 is transmitted. When the blocking is resolved, the message can be transmitted immediately. In order to evaluate such a scheme, we need to analyze a queueing network model consisting of M finite capacity queues in tandem.

The tandem network of finite capacity queues is extremely difficult to analyze except for certain trivial cases. It is however always possible to use a numerical procedure in order to find interesting quantities. This is accomplished by generating a Markovian system possibly by approximating the arrival process and service process by Coxian processes [KLEI 75]. A Coxian arrival process is a stochastic process where the interarrival time distribution is Coxian and successive interarrival times are independent.

The numerical method has certain advantages, but it very quickly becomes impractical if the number of states in the Markov chain is large.

A purely analytical approach is very difficult again due to the fact that the dimensionality of the state-space is often too large.

Consequently, an approximate method that allows us to obtain almost accurate results for the steady-state probabilities and associated quantities such as network throughput and message delay seems to be a viable alternative.

* This work was supported by National Science Foundation Grant No. ECS 77-24110.

0190-3918/81/0000/0087$00.75 © 1981 IEEE

Approximate analyses of exponential queueing networks of the type we are interested in have appeared in the existing literature. The classical paper by Hunt [HUNT 56] provides the first introduction to this difficult problem. More recent papers are those by Hillier, et. al., [HILL 67] that concentrates on finding the network throughput, and the paper by Takahashi, et. al., [TAKA 80] in which an approximative method based upon an M/M/1 queueing model with adjusted arrival rates and effective service rates is given. In the next section we will present a method that we believe and also show to be better and more effective than other existing methods.

Approximate Analysis

Our queueing model of the logical link with local control is a tandem network with finite capacity queues. The service time distribution in the i:th node is exponential with parameter μ_i and the arrival process to the queueing network is Poissonian with rate λ. The capacity in the i:th node is N_i messages; included in this is the message currently under transmission if any and the messages waiting to be transmitted. We also assume that messages arriving to node 1 when the node is filled to its capacity are lost from the system and the last node, node M, cannot have any blocked messages.

Let n_k = the number of messages in node k and define $a_k = P\{n_k = N_k\}$ as the "blocking" probability for the k:th node. Clearly if a_1 is known, the total throughput for the tandem link is $\lambda(1 - a_1)$, since we do not allow messages to be lost or destroyed once they have been given access to the tandem link. This being the case, it follows that the throughput for each finite queue in the tandem is $\lambda(1 - a_1)$.

The idea behind our approximation is to decouple the tandem network into M individual queues with arrival rates and service times given such that the analyses of the individual queues, an off-line analysis, will give relatively accurate results for the total queueing model. The approach we follow is to define a new service time distribution and an effective arrival rate.

The service time distribution for the i:th node is found by observing that as long as the following node is not filled to capacity the service time is exponential with parameter μ_i. If the i+1:st node is filled to capacity, the effective service time of the message at the head of the queue in node i is taken to be the sum of two independent exponential random variables with rates μ_{i+1} and μ_i respectively. The probability for the latter event is a_{i+1}. The service time distribution can thus be represented as a two-stage Coxian distribution, see Figure 1.

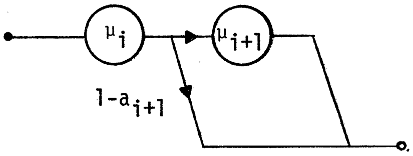


Figure 1: Service Mechanism for the i:th Node

In the above service time representation we have ignored the cases where subsequent nodes, i.e., nodes i+2, i+3, also are filled to capacity.

The arrival rate to the i:th node in the off-line analysis is set to

$$\lambda(1 - a_1) \;/\; (1 - a_i) \quad . \qquad (1)$$

The i:th node is a finite capacity single-server queueing system and according to our assumptions a fraction $1 - a_i$ of the arriving messages will be served by this queue. By using the arrival rate as defined in (1) we ensure that the throughput obtained by the off-line analysis of the node is the correct one, namely $\lambda(1 - a_1)$. We approximate the arrival process to the node with a Poisson process with the correct arrival rate. Each off-line single-server queue is now analyzed as an M/COX2/1 finite capacity queue.

An M/COXK/1 finite capacity queue can be analyzed by defining a Markov chain [MARI 80] with a state-space given by the number of customers, n, in the queue and the service stage j in which the customer in service, if any, is currently residing. The steady-state probability of this event is denoted by p(n,j) and the following balance equation can easily be written down.

$$\sum_{j=1}^{K} (1-r_j)\mu_j \; p(n,j) = \lambda \sum_{j=1}^{K} p(n-1,j) \qquad (2)$$

where μ_j is the exponential service rate in the j:th service stage, and r_j the Coxian branching probability. K is the number of stages in the Coxian distribution.

The conditional throughput of an M/COX-K/1 finite capacity system can be expressed as [MARI 80]

$$\nu(n) = \frac{\sum_{j=1}^{K} (1-r_j)\mu_j \; p(n,j)}{p(n)} \qquad (3)$$

where

$$p(n) = \sum_{j=1}^{K} p(n,j) \qquad (4)$$

provided that n ≠ 0. Using (3) the balance equation can be written as

$$\nu(n)p(n) = \lambda p(n-1) \ . \qquad (5)$$

For a two-stage Coxian distribution the conditional throughput $\nu(n)$ can be determined recursively by the following formulas [MARI 80]:

$$\nu(n) = \frac{\lambda \ \mu_1(1-r_1)+\mu_1\mu_2}{\lambda+\mu_1+\mu_2-\nu(n-1)} \qquad 1 < n < N \qquad (6)$$

$$\nu(N) = \frac{\mu_1\mu_2}{\mu_1+\mu_2-\nu(N-1)} \qquad n = N \qquad (7)$$

and

$$\nu(1) = \frac{\lambda \ \mu_1(1-r_1)+\mu_1\mu_2}{\lambda+\mu_2+r_1\mu_1} \qquad n = 1 \qquad (8)$$

where N is the system capacity. Using (6), (7), and (8) we easily find p(n).

In our off-line analysis every node except the last node is modelled as an M/COX-2/1 finite capacity queue. The last node is modelled as an M/M/1 finite capacity queue. This last queue is easily analyzed, and we find that

$$a_M = P\{n_M = N_M\} = \frac{\rho^{N_M}(1-\rho)}{1-\rho^{N_M+1}} \qquad (9)$$

where

$$\rho = \frac{\lambda(1-a_1)}{\mu_M(1-a_M)} \ . \qquad (10)$$

The blocking probabilities a_1, a_2, ..., a_M are not known but can be computed iteratively by using the following observation. If a_{i+1} is known, a_i can be computed, for $i = M-1, M-2, \ldots, 1$. a_M is by our construction a function of a_1 and by choosing an initial value for a_1 the blocking probabilities can be computed iteratively to any desired accuracy.

Numerical Examples

In this section we present numerical examples in which we compare the results by the approximate method to exact results if such are available or to other approximate methods or to results obtained by simulation.

In the first simple example we consider a tandem network consisting of two exponential servers each with the same service rate μ and each with the same finite capacity $N_i = 1$, Figure 2. The exact results for the blocking probabilities are easily obtained by solving the Markov chain problem that can be formulated. In Table I we compare the exact and approximate blocking probabilities for different values of the ratio λ/μ.

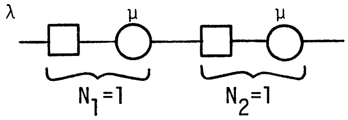


Figure 2: Network For Example 1

We note that the blocking probabilities in node 1 are consistently overestimated and the blocking probabilities in node 2 underestimated. Subsequent examples will show that this is always the case. The reason for this behavior is that we approximate the arrival process to the nodes with a Poisson process. This is not a serious problem, since it implies that the approximative method gives a lower bound for the network throughput.

Table I: Comparison of the Exact and Approximate Results for the Blocking Probabilities in Example 1

λ/μ	$P(n_1=1)$		$P(n_2=1)$	
	Exact	Approximate	Exact	Approximate
0.2	0.178	0.189	0.164	0.162
0.3	0.251	0.269	0.225	0.220
0.4	0.314	0.336	0.275	0.266
0.5	0.369	0.394	0.316	0.303
0.6	0.416	0.446	0.350	0.334
0.7	0.458	0.488	0.380	0.359
0.8	0.494	0.525	0.405	0.380
0.9	0.527	0.557	0.426	0.399

In our second example we investigate a three-node tandem queue where the first server is always kept busy, i.e., an overload situation. Obviously of interest here is to find the blocking probabilities of the second and third queue. In order to be able to make a comparison between exact results obtained from a Markovian analysis and the approximate method, we again select a fairly simple system with node capacities equal to one message, Figure 3.

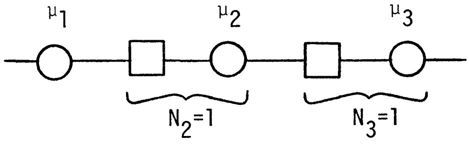


Figure 3: Tandem Network For Example 2

In Table II we show how the exact and approximate methods compare for different values of μ_1, μ_2, and μ_3.

Table II: Comparison of the Exact and Approximate Results for Blocking Probabilities in Example 2

Parameters			Blocking Probabilities	Exact	Approximate
μ_1	μ_2	μ_3			
1.1	1.2	1.3	$P(n_2=1)$	0.754	0.776
			$P(n_3=1)$	0.517	0.494
1.2	1.4	1.6	$P(n_2=1)$	0.723	0.735
			$P(n_3=1)$	0.484	0.460
1.3	1.6	1.9	$P(n_2=1)$	0.697	0.704
			$P(n_3=1)$	0.458	0.435

With minor modifications the approximate method can also be used in a network with random routing. In order to test the robustness of our method, we have used it on a simple sample network, see Figure 4.

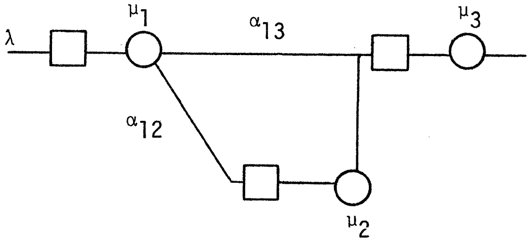


Figure 4: Three-node Network for Example 3

After completion of service at node 1, a message is routed to node 2 with probability α_{12} independent of the current state of the network and with probability α_{13} to node 3, $(\alpha_{12}+\alpha_{13}=1)$.

The reason for choosing this network is that it was used in [TAKA 80] to illustrate another approximate method for analyzing queueing networks with blocking. In Table III we compare the results for the blocking probability at node 1 obtained by an exact Markovian analysis, the approximate analysis due to Takahashi, et. al., and the method presented in this paper.

Our fourth and final example brings us to a data communication network with local control and also external traffic imposed on the tandem link. In order to be able to compare our results with others we choose exactly the same configuration as the one chosen in [PENN 75]. We assume accordingly that the external traffic is only allowed to use one server in the tandem network and then leave the tandem, see Figure 5

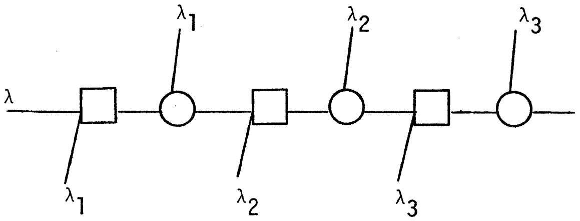


Figure 5: Queueing Network Model of a Path According to the Model Used in [PENN 75]

We use a similar approach as in [PENN 75] to account for the external traffic. We do however use our method for computing the blocking probabilities and the loading factor. The loading factor is by definition in [PENN 75] the fractional increase in queueing time suffered by external messages due to the presence of link

Table III: Comparison Between Exact and Approximate Methods for Computing the Blocking Probability (PB_1) at node 1 ($\alpha_{12} = \alpha_{13} = 0.5$)

Arrival Rates	Service Rates			Capacities			Exact	Takahashi Approximate	Our Method
λ_1	μ_1	μ_2	μ_3	N_1	N_2	N_3	PB_1	PB_1	PB_1
1.0	1.0	1.1	1.2	1	1	1	0.560	0.587	0.566
1.0	1.0	1.3	1.6	1	1	1	0.537	0.563	0.543
1.0	1.0	1.5	2.0	1	1	1	0.525	0.549	0.530
1.0	1.0	1.7	2.4	1	1	1	0.517	0.540	0.522
1.0	1.0	2.0	3.0	1	1	1	0.511	0.531	0.514

messages averaged over all external messages. The effect of local control as a function of the node capacities can be shown by plotting the loading factor as a function of blocking probability as in Figure 9, in [PENN 75]. We show in Figure 6 the loading versus blocking probability. If we had superimposed the curve presented in [PENN 75] in our diagram, the result would have been an almost overlap. Due to the fact that we do not have access to simulation data for this example, we cannot make a judgement about how accurate the method is. The previous examples have however shown that the method presented in this paper is more accurate than other approximate methods.

Conclusion

We have presented an approximate method for the analysis of open exponential queueing networks with finite capacity. It has been demonstrated through several examples that the method produces results that are quite accurate. In particular we showed that the impact of local flow control in message-switched data communication networks can be analyzed by this method. We have in this paper constrained ourselves to open exponential queueing networks. The results obtained are certainly useful, but we would like to be able to extend them to more general networks. This is possible as long as the arrival processes can be modelled with Coxian interarrival times and the service time distribution with a Coxian distribution. Note that the Poisson arrival process is a special case of a renewal input process with Coxian interarrival time distribution. The resulting "off-line" queueing system can then be modelled as a finite capacity queue with Coxian input and output. Further work in this area is currently being carried out. The main obstacle, however, is to check the accuracy of the result, since no exact results seem to be available and simulation results are scarce.

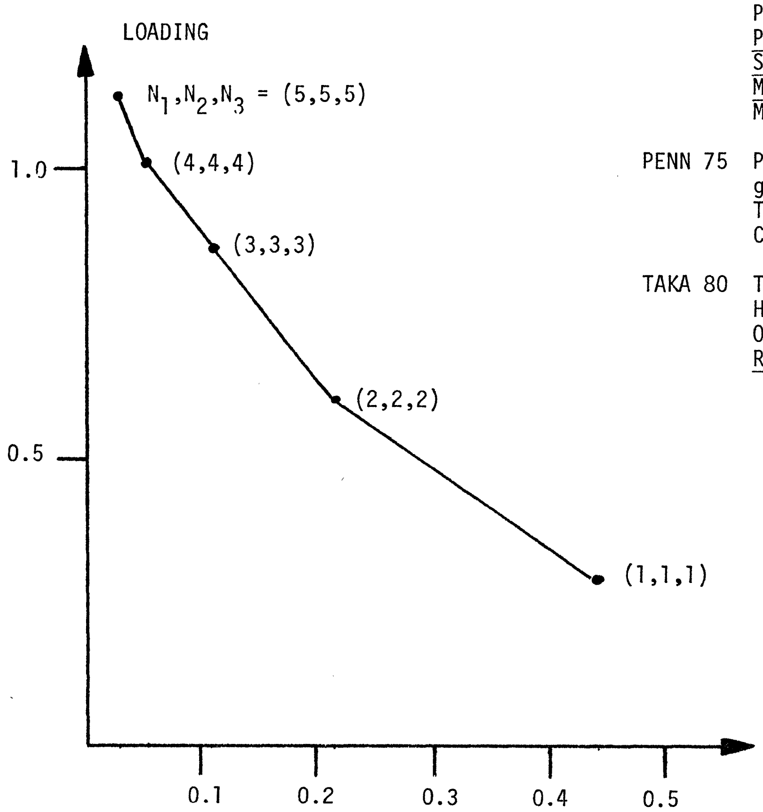


Figure 6: Local Control: Loading Versus Blocking Probability

References

GERL 80 Gerla, M. and L. Kleinrock, "Flow Control: A Comparative Survey," IEEE Trans. on Comm., COM-28, 1980, pp. 553-574.

HILL 67 Hillier, F. S. and R. W. Boling, "Finite Queues in Series With Exponential or Erlang Service Times - A Numerical Approach," Oper. Res., Vol. 15, 1967, pp. 286-303.

HUNT 56 Hunt, G. C., "Sequential Arrays of Waiting Lines," Oper. Res., Vol. 4, 1956, pp. 674, 683.

KLEI 75 Kleinrock, L., Queueing Systems, Vol. I: Theory, Wiley-Interscience (New York), 1975.

MARI 80 Marie, R., "Calculating Equilibrium Probabilities for $\lambda(n)/C_k/1/N$ Queues," Proc. of Performance 80, International Symposium on Computer Performance Modelling, Measurement and Evaluation, May 28-30, 1980, pp. 117-125.

PENN 75 Pennotti, M. C. and M. Schwartz, "Congestion Control in Store and Forward Tandem Links," IEEE Trans. on Comm., COM-23, 1975, pp. 1434-1443.

TAKA 80 Takahashi, Y., H. Miyahara, and T. Hasegawa, "An Approximation Method for Open Restricted Queueing Networks," Oper. Res., Vol. 28, 1980, pp. 594-602.

BLOCK TRIDIAGONAL SYSTEM SOLUTION ON RECONFIGURABLE ARRAY COMPUTERS

by

RAJAN N. KAPUR
DEPT. OF ELECT. ENGG.

JAMES C. BROWNE
DEPT. OF COMPUT. SCI.,

THE UNIVERSITY OF TEXAS AT AUSTIN,
TEXAS, 78712.

ABSTRACT

Reconfigurable array computer architectures give the application designer power to define an execution architecture or architectures and an interaction geometry appropriate to the computational architecture of the algorithm under consideration. Accurate estimation of execution times for reconfigurable architectures requires determination of appropriate computational structures for the algorithm and analysis of the cost of interprocess data movement, synchronization delays and reconfiguration faults as well as actual execution time for the algorithm in the architecture selected. This paper reports such a formulation of an algorithm whose instruction count has previously been well characterized, even/odd elimination of block tridiagonal linear systems. The algorithm naturally decomposes into three steps each of which requires a different computational structure and displays a different natural degree of parallelism. It gives a speed up linear in the number of processors when degrees of parallelism appropriate to each step are employed. Data movement , synchronization and reconfiguration fault costs are found to be about 10% of the computation costs.

1.0 INTRODUCTION

The practical formulation of parallel algorithms is limited by the interconnection geometry of the multi-processor architecture which is to host the computation. Any fixed geometry of processor interconnection limits the class of algorithms which can be implemented. A full cross bar network removes all restrictions on algorithm formulation, but is prohibitively expensive for even a moderately large number of processing elements. A common memory architecture will suffer performance degradation from interference as the number of processors becomes large. The solution to this dilemma is being sought with the development of reconfigurable interconnection networks to link arrays of processing elements (processors and memories). A variety of reconfigurable network architectures have been proposed [LIP77,SIE79]. The common elements of these interconnection networks include:

1. implementation costs which grow at a rate of n log n where n is the number of elements to be connected [GOK73].
2. the ability to establish resource partitions which execute independently except for interactions through specified communication and data channels.
3. the ability to implement a wide spectrum of interconnection geometry.

The availability of such reconfigurable architectures opens new dimensions for the formulation of parallel algorithms. The arrangement of computations can be based upon the structure of the algorithm rather than upon a specific available architecture. Resource partitions can be tailored to the computation and data movement requirement of the algorithm. The importance of problem specific interconnection geometry is noted by Gentleman [GEN78]. He demonstrates that fixed geometries can easily lead to data movements dominating execution time for matrix multiplication and matrix inversion.

Problems can be formulated as sets of tasks or sequences of sets of tasks (MIMD/SIMD mode of computation) rather than merely sequences of tasks as is the case on the uni-processor. Each task set may have a different degree of parallelism and/or a different interconnection geometry.

The execution time of an algorithm on a parallel computational structure depends not only upon the operation count of the computation, but also the time required for data movement and the time lost to synchronization delays. For an algorithm or process with disjoint phases which require different computational structures to give an optimal execution time for each phase, then the time for reconfiguration of the architecture must be included in the total execution time. This paper defines an algorithm for the odd/even elimination of block tridiagonal systems on a reconfigurable array computer. The algorithm is resolved into three distinct steps, each of which uses a different degree of parallelism and has a different interconnection geometry. This formulation displays advantages for the use of reconfigurable array systems with SIMD computers assigned to blocks for the odd-even algorithm. These are

1. Each SIMD machine operates independently, therefore independent pivoting is possible.
2. Each SIMD machine can be tailored to the size of the block it is handling so that synchronization waits are minimized.
3. The synchronized nature of the shared data access is well suited to intercommunication mechanisms characteristic of reconfigurable computers.

0190-3918/81/0000/0092$00.75 © 1981 IEEE

2.0 RECONFIGURATION

This section defines and describes the concepts of an MIMD/SIMD execution mode for reconfigurable arrays of processing elements and describes the modes of data movements which characterize reconfigurable network architecture computer systems.

2.1 MIMD/SIMD

Reconfigurable computers are generally implemented as arrays of processor modules and memory modules with a modular interconnection network. This definition of reconfiguration is quite different from the instruction set level reconfiguration as in the Burrough's B1700 [ORG78, RAU78]. The interconnection network can establish resource partitions consisting of a subset of processor modules and memory modules. A partition can be configured to implement SISD or SIMD modes of operation (figure 1). A job then consists of a number of partitions (a task is a partition) interconnected according to the data flow of the job. Processors within a partition are under lock-step control of one instruction stream; processors from different partitions are under the control of different instruction streams.

2.2 Communication And Synchronization

Two distinct kinds of data transfer requirements arise from the MIMD/SIMD mode of operation. Data transfers between partitions are needed both with a computation structure and between the structures of different phases or stages. A synchronization mechanism is needed to control the data transfers. Additionally SIMD machines implemented as arrays of processor modules must deal with the problem of data alignment. Consider as an illustration row and column access of matrices. In a pipeline vector processor, e.g. [HIN72,WAT72,RUS78], it is possible to organize data in interleaved memories so that row and column access can be performed equally efficiently. The ILLIAC IV [BAR68] on the other hand is considerably harder to program for simultaneous row and column access for two reasons:

1. the physically distributed nature of the sources and targets of data.
2. a static and limited linkage network.

The Burroughs BSP [BUR77] avoids this problem by the brute force solution of using a cross point interconnection network between processor and memory modules.

We give here an outline of the pertinent features of the communication subsystem [PRE79,SEJ81] in TRAC, a representative reconfigurable computer and show how they provide capability for the movement of data both within and between SIMD partitions.

TRAC provides two kinds of physical channels for communication: packets and shared memories. Packets are continuous streams of bytes and are memory-to-memory transfers. A data vector and a realignment vector are specified. Packet movement through the network is used to create a realigned result vector.

The concept of shared memory in resource partitioned architectures such as TRAC is not the same as in multiple processors sharing access to a common physical memory address space such as C.MMP [WUL72]. In these latter architectures sharing is on a cycle by cycle basis with a possibility of interference when more than one processor endeavors to access a given memory module. Synchronization mechanisms for access to memory are commonly implemented in software or firmware. The execution of these sychronization mechanisms consumes memory bandwidth and themselves interfere with the performance of the sharing resources (e.g. the spin-lock mechanism as described in [OLE78]). In reconfigurable network architectures such as TRAC, sharing may be combined with synchronization by altering the interconnection network to move a physical memory module from one task address space to a different task address space. TRAC accomplishes this extended sharing concept by a hardware configuration with a shared module at the root of a tree whose leaves are processors (figure 2). To obtain a shared memory a processor must execute an ACQUIRE instruction. The processor blocks if the module has already been acquired by another processor. Retry effects only the acquisition circuitry, not access by other processors.

3.0 BLOCK TRIDIAGONAL SYSTEMS

Block tridiagonal systems of linear equations occur frequently in scientific computations, often forming the core of more complicated problems. Numerical methods for the solution of such systems are well understood and techniques tailored to the solution of such systems on pipelined supercomputers have been studied extensively [TRA76,MAD75,HEL77].

The linear system is represented as Ax=v with

$$A = \begin{bmatrix} b(1) & c(1) & & & & \\ a(2) & b(2) & c(2) & & & \\ & a(3) & b(3) & c(3) & & \\ & & \ldots\ldots & & & \\ & & & \ldots\ldots & & \\ & & & & \ldots\ldots & \\ & & & a(N-1) & b(N-1) & c(N-1) \\ & & & & a(N) & b(N) \end{bmatrix}$$

$$= (a(j),b(j),c(j))_{N}$$

where b(i) is a n_i x n_i matrix and a(1) = 0 and c(N) = 0.

The odd/even elimination method (and the odd/even reduction method which can be regarded as a compact version of the former) is widely regarded as an efficient direct method for the case where the $n_i \times n_i$ blocks are small enough to be stored explicitly [HEL77].

Consider odd/even elimination as described in Heller [HEL77], section 4: pick three consecutive block equations from Ax = v, A = (a(j),b(j),c(j))N

$$a(k-1)x(k-2) + b(k-1)x(k-1) + c(k-1)x(k) = v(k-1) \quad \ldots\ldots(k-1)$$

$$a(k)x(k-1) + b(k)x(k) + c(k)x(k+1) = v(k) \quad \ldots\ldots(k)$$

$$a(k+1)x(k) + b(k+1)x(k+1) + c(k+1)x(k+2) = v(k+1) \quad \ldots\ldots(k+1)$$

If we multiply equation k-1 by $-a(k)b^{-1}(k-1)$, equation k+1 by $-c(k)b^{-1}(k+1)$, and add, the result is:

$$\begin{aligned}&(-a(k)b^{-1}(k-1)a(k-1))x(k-2)\\ &+ (b^{-1}(k) - a(k)b^{-1}(k-1)c(k-1)\\ &- c(k)b^{-1}(k+1)a(k+1))x(k)\\ &+ (-c(k)b^{-1}(k+1)c(k+1))x(k+2)\\ \\ &= (v(k) - a(k)b^{-1}(k-1)v(k-1)\\ &-c(k)b^{-1}(k+1)v(k+1)).\end{aligned}$$

For k=1 or N there are only two equations involved and the modifications should be obvious. This operation eliminates the odd unknowns for k even and the even unknowns for k odd. By collecting the new equations into the block pentadiagonal system H.2x =v.2, (with A defined as H.1) it is seen that row elimination has preserved the fact that the matrix has only three non zero block diagonals, but they are further apart. A similar set of operations is applied combining equations k-2, k and k+2 in H.2 to produce H.3x=v.3 system. This process is repeated until only one block diagonal remains (or in the case of semi direct methods, some accuracy criteria are fulfilled). The initial coefficient matrix H.1 contains 3N-2 non zero blocks while the final matrix consists of N non zero blocks along the main diagonal.

Solving the N blocks independently gives the required solution.

Figure 3 shows the effect of 5 steps of elimination on a 16x16 block tridiagonal system.

4.0 DATAFLOW AND IMPLEMENTATION

In this section we will look at the dataflow characteristics of odd/even algorithms. The computational aspects, such as operation counts are well understood; the communication geometry is studied herein and found to be regular and simple.

Computationally, instead of determining $b^{-1}(i)$ explicitly, LU factorization of b(i) is generally resorted to:

compute LU factors of b(k), (1<=k<=N)

solve $b(k)\ [\ \underline{a}(k)\ \underline{c}(k)\ \underline{v}(k)] = [\ \bar{a}(k)\ \bar{c}(k)\ v(k)\ ]$, 1<=k<=N

$b(k).2 \longleftarrow b(k).1 - a(k).1\ \underline{c}(k-1) - c(k).1\ \underline{a}(k+1)$ ....1<=k<=N

$v(k).2 \longleftarrow v(k).1 - a(k).1\ \underline{v}(k-1) - c(k).1\ \underline{v}(k+1)$ ....1<=k<=N

$a(k).2 \longleftarrow a(k).1\ \underline{a}(k-1)$, 3<=k<=N

$c(k).2 \longleftarrow c(k).1\ \underline{c}(k+1)$, 1<=k<=N-2

4.1 Intertask Dataflow

The sequence of actions that results in the computation of H.i+1 from H.i is referred to as a stage: in this case each stage is shown to consist of three steps and the steps further consist of substeps.

Consider the input dataflow for computing H.2 and v.2. In the first step, the first substep results in the LU factorization of b(k) ; this is then used in the next substep for computing $\underline{a}(k)$, $\underline{c}(k)$ and $\underline{v}(k)$. N way parallelism is displayed in this step.

In the second step the computation of a(k).i+1and c(k).i+1 requires a(k).i, $\underline{a}(k-1)$.i and c(k).i, $\underline{c}(k+1)$.i respectively; $b(\bar{k})$.i+1 and v(k).i+1 require data from the (k-1), (k) and (k+1)th row equations. Binary operations are performed on the blocks - pairwise access to blocks is sufficent giving rise to upto (N/2) way parallelism.

Figure 4a shows the interconnection geometry needed for the second step for an 8x8 system. The blocks are stored in separately accessible shared memories one block row per shared memory. The diagram to the right of the 8x8 tridiagonal system is the inter connection pattern with circles representing processors and squares shared memories. The edges represent potential links that are activated as 4 separate patterns as shown further to the right. The new blocks computed at the end of substep 2 are shown in curly brackets between the patterns of substep 2 and 3 ; the remaining new blocks are completed at the end of substep 4.

The crucial observation here is that while the datasets are shared across processors, the sharing is conflict free within a substep. The connection pattern cycles through the states of a 2-pole 3 position switch.

H.2 is a pentadiagonal matrix - the application of an inverse perfect shuffle [STO71] partitions this matrix into two tridiagonal matrices, one consisting of the odd numbered coefficient blocks and the other of even numbered ones (Figure 4b).

If the matrix A contains N=2**m blocks then the dataflow geometry for the next step 2 is represented by a graph that is a proper subset of the graph used in the earlier step 2 (Figure 5). This inclusion property is seen in every step 2 until the block diagonal is computed.

Thus we use the precisely same dataflow template in the generation of every H.i+1 from H.i ; three steps with different connection geometries are needed - a direct connection, a 2-pole

3-position switch based connection followed by an inverse perfect shuffle.

A number of proposed reconfigurable computers can implement these and other communication behavior quite efficiently. Note that if we were tto hardwire the interconnect pattern we would be using a special purpose machine of limited applicability to other problems (Eg. the shuffle exchange network [STO71] in high performance FFT boxes).

An implementation based on the use of shared memory in TRAC is now sketched briefly. The processors are scheduled as SIMD partitions with width commensurate with the block size under consideration. The shared datasets are stored in shared memory modules - each circle of figure 4 is realized as an array of shared memories of width conformal with the processor width. The time to switch the 2-pole 3-position based switch is a critical parameter in the performance of the algorithm. On the basis of a 10 microsecond acquire time for an unacquired module this parameter is estimated at between 50 and 100 microseconds for TRAC.

4.2 Performance Estimation

The mode of operation described in the previous subsection consists of asynchronously executing processes which synchronize periodically to transfer data. Operations on different blocks may require different computation times. There may be, for example, different search times for the choice of pivot rows. Thus for a performance model to accurately represent this kind of behavior, it must be based on non deterministic time parameters.

We will now present a niave analysis based on average time parameters as a first step towards developing a performance model of reconfigurable computer operation.

The time for data movement depends upon the width of the processors and the width of the arithmetic. We choose a definite configuration to illustrate the magnitude of the communication costs. A 16 bit wide SISD partition will be assigned to each block. (If blocks are of uneven size a more powerful partition could be assigned to larger blocks.) Arithmetic will be on floating point numbers with 64-bit mantissas and 8-bit exponents.

Let the system be 16x16 blocks and each block be 8x8 (total matrix dimension 128x128). Each 8x8 block requires about 600 words of storage. A block row consisting of three 8x8 non zero blocks and a 8x1 vector requires about 2000 words of storage.

The following notation is used for representing operation times:

T.fpa: floating point addition
T.fpm: floating point multiply/divide
T.xfer: memory to memory transfer time for one word.
T.swi: acquisition and setup time to obtain shared memory.

The evaluation of H.5 from H.1 proceeds in 4 stages with each stage evaluating H.i+1 form H.i. The first step of a stage consists of the LU factorization of b which is used to evaluate $\underline{a}$,$\underline{c}$ and $\underline{v}$. The second step consists of three substeps that correspond to the three positions of the 2-pole, 3-position switch. The final step performs the inverse perfect shuffle to position data for the next stage. From the discussion of the previous subsection it is evident that the first and last step display 16 way parallelism and the second step 8 way parallelism.

H.5 is finally solved as 16 uncoupled linear systems to obtain the required solution.

We will use the notation Pi(k,l,m..) to represent the state where partition Pi is connected to datasets k,l,m.. and dataset k contains a(k),b(k),c(k) and v(k). This is the timing for the implementation with shared memory. The results of this analysis are discussed at the end of the section and can be directly skipped to without loss of continuity.

STAGE 1:Compute H.2 from H.1

Step 1:
Configuration: Pa(1), Pb(2),.... , Pp(16)
Setup time ~ T.swi.

Substep 1.1
Computation: Pa: b(1) (LU decomposition)
Pb: b(2)
.
.
Pp: b(16)
Compute time 200*T.fpm + 200*T.fpa
Substep 1.2
Computation: Pa: $\underline{a}$(1), $\underline{c}$(1), $\underline{v}$(1)
Pb: $\underline{a}$(2), $\underline{c}$(2), $\underline{v}$(2)
.
.
Pn: $\underline{a}$(16), $\underline{c}$(16), $\underline{v}$(16)
Compute time ~ 1200*T.fpm + 1200*T.fpa

Step 2
Substep 2.1
Configuration: Pa(1), Pc(2,3), Pe(4,5)
....., Po(14,15)
Setup Time ~ 2*T.swi

Substep 2.2
Configuration: Pa(1,2), Pc(3,4), Pe(5,6),
.......,Po(15,16)
Setup time ~ 2*T.swi.

Computation: Pa: b(1).2, a(1).2,
c(1).2, v(1).2
Pc: b(3).2, a(3).2,
c(3).2, v(3).2
.
.
.
Compute time ~ 2000*T.fpm + 2000*T.fpa

Substep 2.3
Configuration: Pa(1,2), Pc(3,4), Pe(5,6)
......., Po(15,16)
Setup time ~ 0*T.swi

```
Substep 2.4
Configuration:  Pa(2,3), Pc(4,5), Pe(6,7),
                ......., Po(16)
Setup Time: ~   2*T.swi

Computation:    Pa:     b(2).2, a(2).2,
                        c(2).2, v(2).2
                Pc:     b(4).2, a(4).2,
                        c(4).2, v(4).2
                .
                .
                .
Compute Time~  2000*T.fpm + 2000*T.fpa
Step 3

Substep 3.1
Configuration:  Pa(1),  Pb(2),  Pc(3),  Pd(4),
                Pe(5),  Pf(6),  Pg(7),  Ph(8),
                Pi(9),  Pj(10), Pk(11), Pl(12),
                Pm(13), Pn(14), Po(15), Pp(16)
Setup time ~ T.swi

Computation:    Transfer source dataset contents
                to local buffer.
Compute time ~ 2000*T.xfer

Substep 3.2
Configuration:  Pa(1),  Pb(9),  Pc(2),  Pd(10),
                Pe(3),  Pf(11), Pg(4),  Ph(12),
                Pi(5),  Pj(13), Pk(6),  Pp(16),
                Pl(14), Pm(7),  Pn(15), Po(8)
Setup time ~ T.swi

Computation:    Copy local buffer contents to
                target dataset.
Compute time~  2000*T.xfer
```

The important concern is the ratio of direct computation time to the sum of the total non computation time (this consists of the various T.swi, T.xfer, synchronization times etc.). Let us make reasonable assumption that the ratio of execution time for T.xfer¦T.fpa¦T.fpm¦T.swi are 1¦10¦100¦1000 and let T.xfer be one microsecond. Estimate the set-up time for T.fpa,T.fpm and T.xfer be equal to the arithmetic execution time.

For the shared memory implementation the total direct computation time is 1034 milliseconds (ms), total reconfiguration time is 9 ms and data transfer time is 8 ms. Thus, if synchronization time is zero, then the overhead associated with mapping the odd/even elimination to a parallel basis is about 17/1034 or about 2%. Synchronization delays result solely from the differences in processing time for each block. For uniform size blocks, processing time differences between blocks will result from differing effort for pivot selection. This should not exceed 1%. Reconfigurable architectures can assign processing partitions with power proportional to block size. (SISD partitions with a factor of 8 variation in power for 64-bit floating point numbers can be constructed on TRAC.) Thus synchronization delays should be not more than 10% of direct execution time. Using this as an upper bound the total overhead cost in this formulation is approximately 12%.

4.3 Intratask Dataflow

The use of an SISD partition for each block avoids the problem of alternate row/column addressing. Row and column accessing is necessary because the block matrices a(k) and c(k) are involved in both pre multiplication and post multiplication. The use of SIMD partitions would introduce efficiences in the computational part of the formulation. Data distribution, would however become more complex. Packet movement would be used to realign data between pre- and post-multiplication stages. This problem will be approached in a subsequent publication.

5.0 CONCLUSIONS

Reconfigurable array computers have been proposed as a candidate architecture for the VLSI implementation of supercomputers. A crucial motivation is that such machines provide a programmable rather than a fixed geometry communication subsystem. The ability to adapt communication geometry to the requirements of the algorithm is supposed to minimize non-computational execution costs on parallel architectures. A parallel formulation of odd/even elimination of block tridiagonal systems is used to illustrate the effectiveness of reconfigurability. The mechanisms of TRAC which are representative of such architectures are used in the formulation. Data movement, reconfiguration and synchronization costs are found to be small with respect to direct computation costs.

The development of parallel algorithms for reconfigurable architectures is shown to be tractable. This analysis of a parallel formulation of odd/even algorithms is intended to display a paradigm for the formulation of algorithms on reconfigurable array computers.

6.0 ACKNOWLEDGEMENT

We wish to thank Nancy Eatman for preparing the diagrams for this paper and Ashok Adiga for the text processing code that was used in the preparation of this manuscript.

This research was funded by NSF Grant MCS-77-15968.

7.0 REFERENCES

[BAR68] Barnes G. H., etal, 'The Illiac IV Computer', IEEE Trans on Comput, Vol C-17, 1968, pp. 746-757

[BUR77] Burroughs, 'BSP: Overview, Perspective, Architecture', 1977

[GEN78] Gentleman, W.M., 'Some Complexity Results for Matrix Computations on Parallel Processors', JACM 25, 112-115 (1978)

[GOK73] Goke R. L. and Lipovski G. J., 'A Banyan Network for Partitioning Multiprocessor Systems', 1st Symp on Comput Arch 1973, pp. 21-28

[HEL77] Heller D., 'Direct and Iterative methods for Block Tri Diagonal Linear Systems', PhD dissertation, CS Dept, CMU, Pittsburgh, 1977

[HIN72] Hintz R. G. and Tate D. P., 'Control Data STAR-100 Design', 6th Ann IEEE Comput Soc Conf, COMPCON 1972, pp. 1-4

[LIP77] Lipovski G. J. and Tripathi A. R., 'A Reconfigurable Varistructured Array Processor', Proc Intl Par Proc Conf, 1977,pp. 165-174.

[MAD75] Madsen, N. and Rodrigue, G., 'A Comparison of Direct Methods for Tridiagonal System Solution on the STAR-100', Lawrence Livermore Laboratory, 1975

[OLE78] Oleinick, P., 'The Implementation and Evaluation of Parallel Algorithms on the C.mmp', PhD Dissertation, Dept. of Comput. Sci., CMU, Nov. 1978

[ORG78] Organick E. and Hinds, J.A., 'Architecture and Programming of the B1700/B1800 Series', (North Holland, Amsterdam, 1978)

[PRE79] Premkumar U. V., etal, 'Interprocessor Communication in TRAC', 1st Intl Conf on Disti Comput and Systems, 1979, pp. 51-62

[RAU78] Raucher, T.M. and Aggarwala, A.K., 'Dynamic Problem Oriented Redefinition of Computer Architecture via Microprogramming', IEEE TC C-29, 1006-1014 (1978)

[RUS78] Russel R. B., 'The Cray-1 Computer System', CACM, Vol 21-1, Jan 1978, pp. 63-72

[SEJ80] Sejnowski M. C., etal, 'An Overview of the Texas Reconfigurable Array Computer', AFIPS NCC 1980, pp. 631-642

[SEJ81] Sejnowski, M.C., 'Packet Architecture of TRAC', MA Report, Comput. Sci. Dept., UT, Austin, May '81.

[SIE79] Siegel H. J., etal, 'PASM' TR.EE-79-40, School of Electrical Engineering, Purdue University, W.Lafayette, IN. 47907, Aug. 1979

[STO71] Stone H. S., 'Parallel Processing with the Perfect Shuffle', IEEE Trans on Comput, Vol C-20, 1971, pp. 153-161

[TRA76] Traub J. F., etal. 'Accelerated Iterative Methods for the Solution of Tridiagonal Systems on Parallel Computers', JACM, Vol 23, 1976, pp. 636-654

[WAT72] Watson W. J., 'The TI ASC- A Highly Modular and Flexible Supercomputer Architecture!', AFIPS FJCC, Vol 41, 1972, pp. 221-228

[WUL72] Wulf W. and Bell C. G., 'C.mmp- A Multi mini processor', AFIPS FJCC, Dec 1972, pp. 765-777

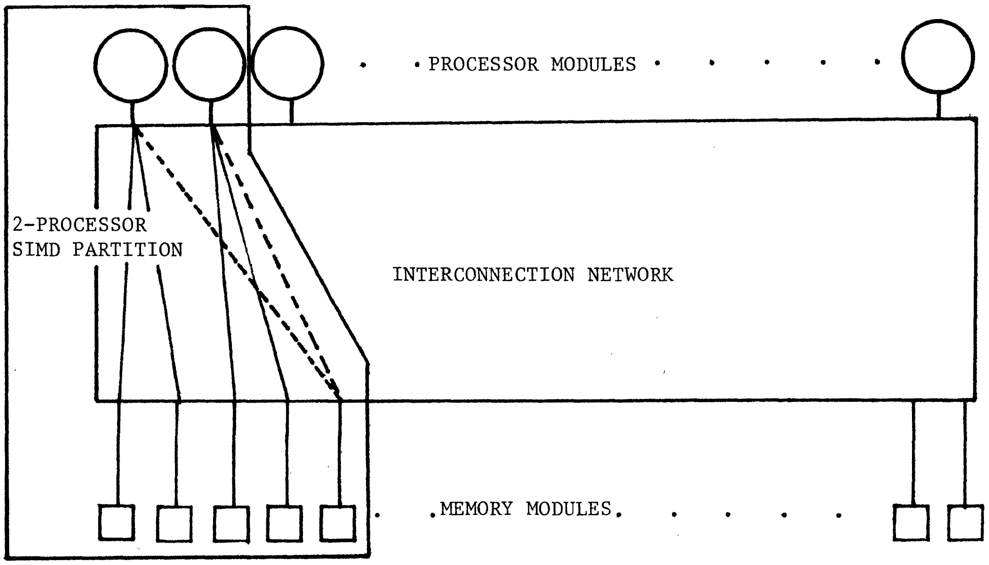
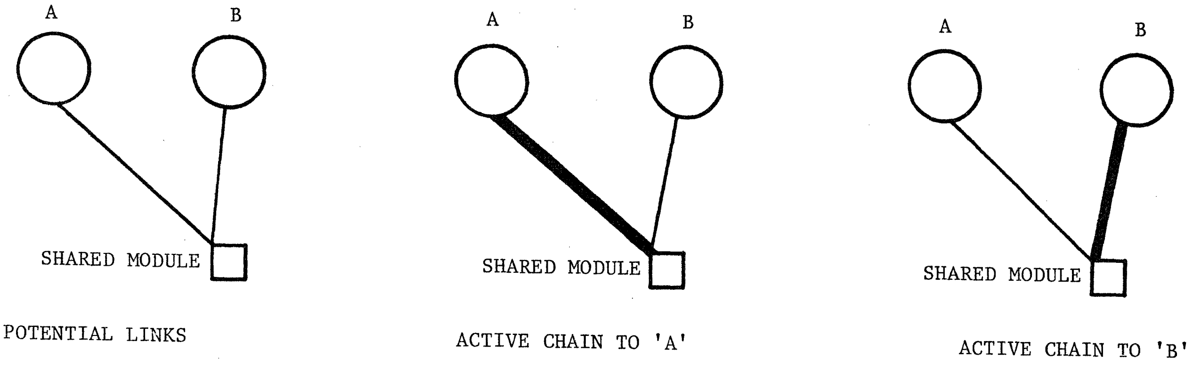


FIGURE 1: PARTITION ON A RECONFIGURABLE ARRAY COMPUTER



POTENTIAL LINKS

ACTIVE CHAIN TO 'A'

ACTIVE CHAIN TO 'B'

FIGURE 2: SHARED MEMORY

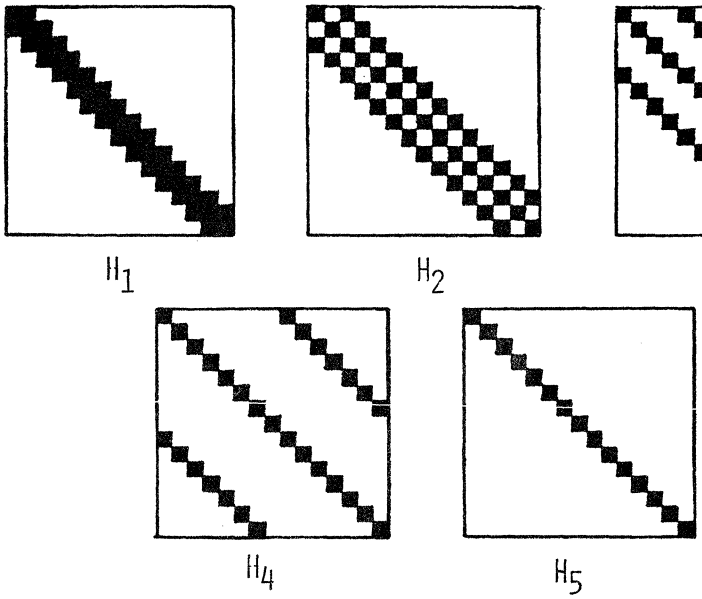


FIGURE 3: 5 STEPS IN THE ELIMINATION OF A 16x16 SYSTEM
(FROM HELLER [HEL77] pp. 39)

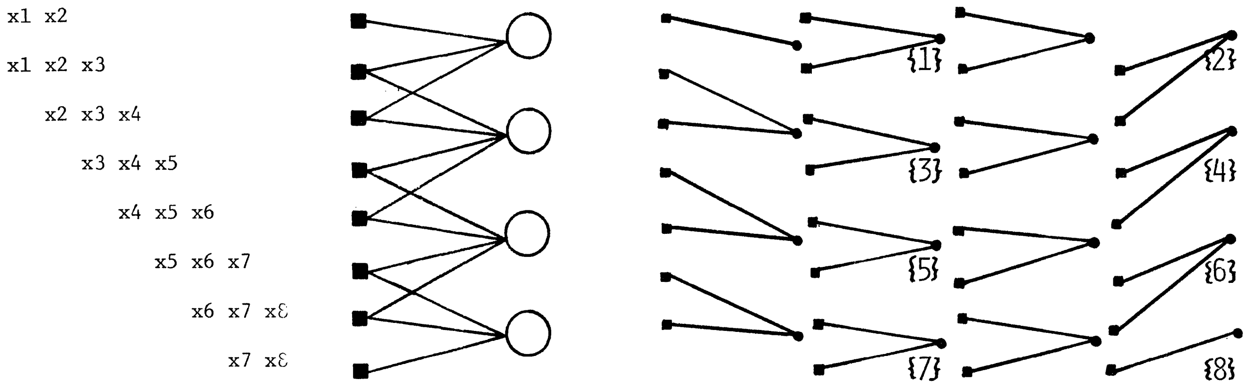


FIGURE 4A: INTERCONNECTION FOR STAGE 1 STEP 2 OF 8x8 TRIDIAGONAL SYSTEM

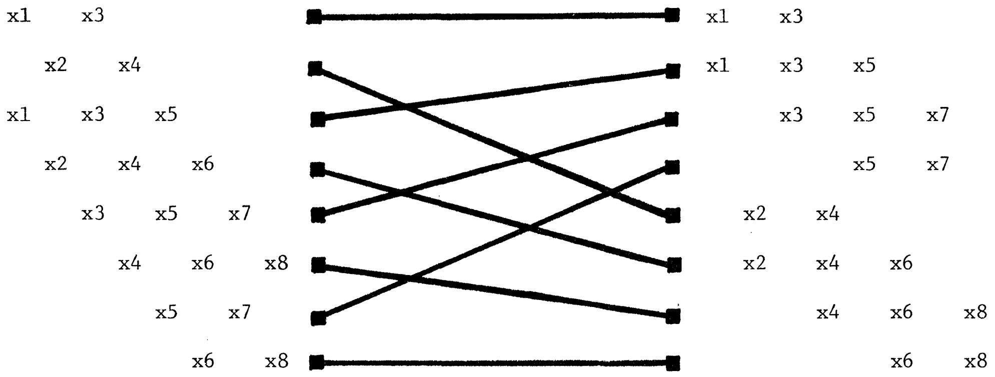


FIGURE 4B: INVERSE PERFECT SHUFFLE TO FORM 2 TRIDIAGONAL SYSTEMS

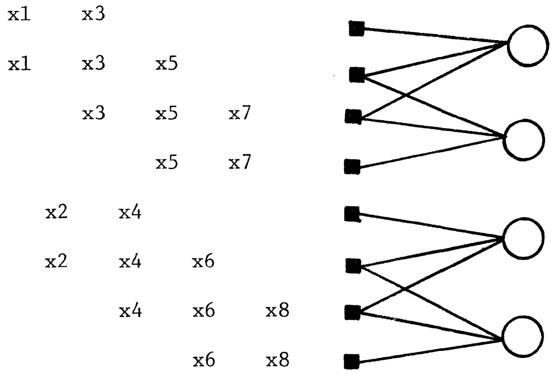


FIGURE 5: INTERCONNECTION FOR STAGE 2 STEP 2 OF 8x8 TRIDIAGONAL SYSTEM

ON MAPPING NON-UNIFORM P.D.E. STRUCTURES AND ALGORITMHS ONTO UNIFORM ARRAY ARCHITECTURES.

by Dennis Gannon[1]
Department of Computer Sciences, Purdue University
West Lafayette, Indiana.

ABSTRACT_ Adaptive algorithms for solving partial differential equation are studied as a means of providing improved speed-up when, in limited processor situations, traditional "uniform" grid parallel methods are inefficient. The difficulty with these methods is that the non-uniform data structures may not be well suited to parallel architectures designed for array and vector problems. In this paper the data-flow problems associated with a class of Multi-Grid algorithms are studied. It is shown that, in spite of non-uniform grid structures, a SIMD machine with an Ω network connection provides a good environment for adaptive computation. Time estimates that include interprocessor communications are derived.

1. INTRODUCTION

While most studies in parallel computation are based on algorithms designed around regular data arrangements like arrays and vectors, many important applications are more efficiently treated with some form of irregular or non-uniform adaptive structures. A simple example is the improvements in serial efficiency obtained for large sparse matrix problems by using linked lists and list algorithms rather than large two dimensional arrays. A second example is given by adaptive methods for solving partial differential equations. One of the fastest parallel algorithm (Sameh, Chen, and Kuck [9]) for solving the Poisson problem

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y)$$

requires a uniform n by n grid defined on a rectangular domain D as shown in figure 1.

With p^2 processors this method requires $O((n/p)^2 log(n))$ time to solve for the unknown u at each of the n^2 grid points given the "data" f at each point. When $p=n$ this algorithm is the best known, but when n is much larger than p one may wish to consider other methods. In many cases, the data f represents very localized activity that can be optimally approximated on a irregular grid as illustrated in figure 2.

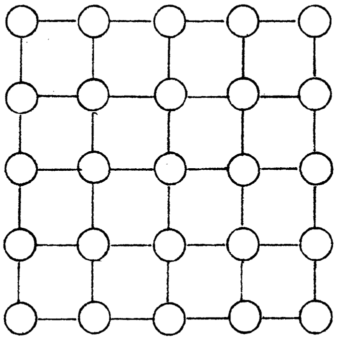


Figure 1. Uniform Grid

The advantage of this "adaptive" grid is that its granularity is fine only where needed and the overall number of node points is greatly reduced.

The savings generated by these techniques extend to parallel computation only if the architecture is rich enough to permit a programming of the algorithm so that the irregular processor communications do not add to theoretical complexity of the method. There

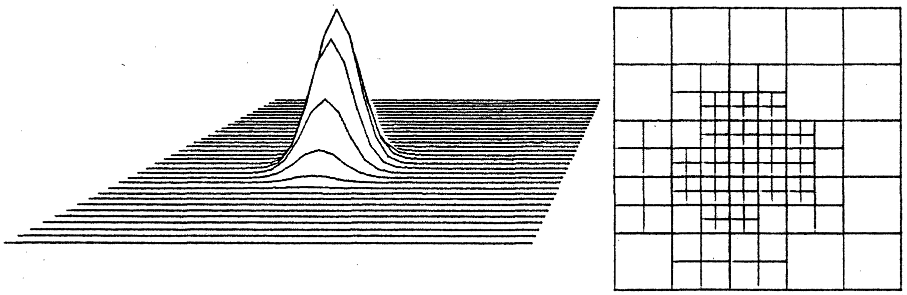


Figure 2. Non-uniform Grid Adaptive Solution.

[1] *Research supported by the National Science Foundation Grant MCS-8109512.*

0190-3918/81/0000/0100$00.75 © 1981 IEEE

are two approaches to solving this problem. For relatively small numbers of processing elements, one attractive solution is to use data flow machines where regularity of structure is of smaller significance than volume of computation. In the case of sparse matrix problems recent work includes the experiments by Lord, Kowalik, and Kumar [6] with the HEP architecture. For the P.D.E. problem described above Rheinboldt and Zave [11] have shown that the adaptive approach can be decomposed at the process level in a manner suitable for limited processor data flow computation.

For more structured, highly parallel computation, the solution is to endow a regular SIMD or MIMD processor array with a connection network capable of accomodating the irregular data requests generated by these adaptive algorithms. Indeed, most designs for architectures devoted to solving partial differential equations, such as the Flow Model Processor (FMP) proposed by Burroughs for NASA, are large multiprocessors equipped with a highly structured interconnection switch. The natural question is if adaptive computation can be shown to produce a real parallel speedup, then which interconnection method provides the most efficient implementation?

In the following paragraphs we illustrate that the problem of computing the solution to the partial differential equation on the adaptive grid can be "mapped" naturally onto a SIMD architecture consisting of an array of p^2 processors and p^2 memory modules interconnected by the well known Ω switch (a key component of the Burroughs FMP). Furthermore, when applicable, the method can run as fast as

$$O(k^3 log(p))$$

including inter processor communication where k is the number of levels of grid "refinement". In the optimal setting a uniform grid of size n by n corresponds to a value of $k = O(log(n/p)$ which, for the sake of comparison, yields an asymptotic estimate of

$$O(log^3(\frac{n}{p})log(p))$$

2. Grid Relaxations and the Mapping Problem.

The mapping problem can be formally defined as follows. Let x_i i=1,,N be the set of nodes at which we seek solution values u_i to the PDE given the data f_i. Let CG represent the the directed data flow graph of the algorithm. That is, the nodes of CG correspond to binary operations and the edges represent the flow of data between computations. A program of the algorithm for a P-processor parallel machine M is a decomposition of CG into disjoint sets of nodes $\{CG_i, i=1..T\}$ each of size $|CG_i| \leq P$ such that if $(x,y) \in CG$ is an edge and $x \in CG_i$ and $y \in CG_j$ then $i < j$. If we assume the machine is equipped with an interconnection switch capable of some set parallel data transfers between processors, then the mapping problem is that of choosing a set of processor assignments $\{f_i : CG_i \to M, i=1..T\}$ that minimize the the number of switch settings required between the T computation stages.

Frequently the algorithm dictates the appropriate switch to give an optimal result to the mapping problem. For example, Grosch [4] has observed that a large class of P.D.E. techniques can be mapped onto a p by p array of processors interconnected by the following three level network. At the first level let each processor be connected to its nearest neighbor in a square mesh lattice. The second level connects each column of p processors with a shuffle connection, and the third level connects each row with a shuffle. Because separable partial differential equations can be easily solved by combinations of Fourier transforms and odd-even reductions aligned along rows and along columns,this Perfect Shuffle, Nearest Neighbor (PSNN) network is well suited for most uniform grid algorithms like the Sameh-Kuck poisson solver described above.

By analyzing the data-flow of the basic components of an algorithm for self adaptive P.D.E. computation we shall see how to extend the PSNN network to an more complex interconnection switch appropriate for the irregular adaptive grids. The resulting network is then shown to be equivalent to the Ω switch of Lawrie [5].

The algorithm studied here is a parallel version of an adaptive Multi-grid method designed by Van Rosendale [10]. The method is based, in part, on the work of Brandt [1] who has studied both parallel and serial implementations of the Multi-Grid idea. The guiding principle of these algorithms is to use a sequence of grids, each finer than its predecessor, to accelerate the convergence of more standard iterative "relaxation" schemes. In the adaptive algorithm, the sequence of grid structures is defined by constructing a nested sequence of subdomains of the problem domain D.

$$D = D_0 \supset D_1 \supset D_2 \quad \ldots \supset D_k$$

A uniform subgrid G_i is placed over each domain D_i so that G_i refines G_{i-1} by quartering certain rectangles. Figure 3 illustrates a simple three level refinement. From the sequence of uniform grids $\{G_i, i=0..k\}$ the algorithm works on the sequence of composite grids $\bar{G}_i = \bigcup_{k=0}^{i} G_k$ for $i=0..k$ to obtain succesively better approximations to the solution starting from an exact solution on the coarsest grid $\bar{G}_0$. To simplify the description of the algorithm and its programming we shall assume we have $P = p^2$ processors with p a power of 2 and that each subgrid G_i is a rectangle of dimension k by l with both k and l being $\leq p$. (A more general algorithm is derivable without modifying the connection network constructed below, but the added detail provides little illumination of the basic result.)

The algorithm (described completely in the next section) is built on 3 basic operations, injection, projection, relaxation, which provide both the setting and a solution to the mapping problem.

2.1 Injection.

Given a piecewise linear solution $u^{(k)}$ on grid $\bar{G}_k$ there exists a natural interpolate $u^{(k+1)}$ on grid $\bar{G}_{k+1}$. The injection operation is this interpolation process denoted by

$$u^{(k+1)} := Injec(u^{(k)}, \bar{G}_k, \bar{G}_{k+1}).$$

The computation to be carried out is simple. The values of $u^{(k)}$ define $u^{(k+1)}$ at the nodes of all subgrids

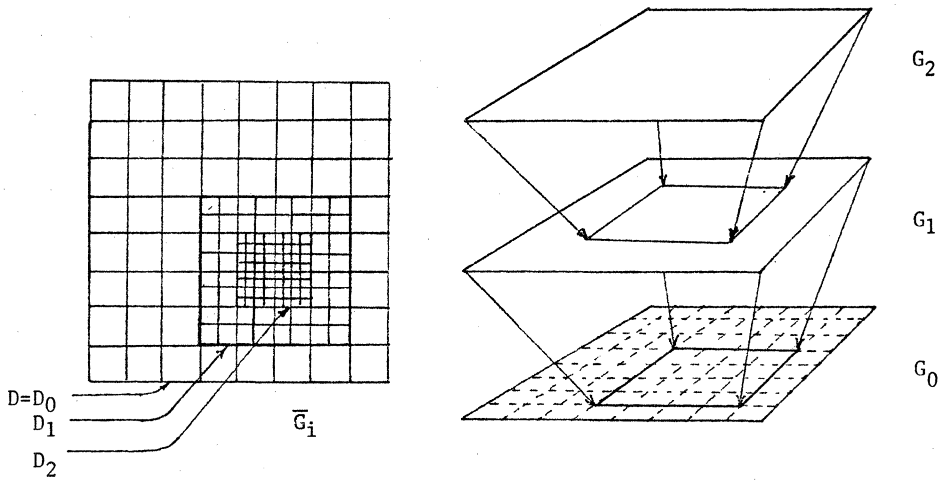


Figure 3. Level Decomposition of Adaptive Refinement.

of $\bar{G}_{k+1}$ except the last fine grid G_{k+1}. At the nodes of G_{k+1} that correspond to nodes of the coarser subgrid G_k, the value of $u^{(k+1)}$ is well defined by $u^{(k)}$. The remaining nodes of $G^{(k+1)}$ are created by the quartersection of rectangles of G^k and therefore the value of $u^{(k+1)}$ is determined by the average of the values at the corners of the quartersected square. The logical choice for mapping the grid structure is to assume for the moment that the p^2 processors are configured as a square array interconnected by a square mesh network. In this way each subgrid can be mapped into the processor array, and the various stages of the data flow graph can be viewed as processes interacting between the various subgrids. In this setting, the interpolation operator can be seen as taking values from subgrid G_k to values in G_{k+1}. The interconnection switch should map the nodes of G_{k+1} that lie in the embedding of G_k to the embedding of G_{k+1}. This is most readily understood by considering a one-column or one-row slice. In figure 4 the one dimensional view shows G_k along the bottom row of processors. The same set of processors are shown along the top row but this time representing the embedding of G_{k+1}. The lines connecting the two represent the identification of equivalent nodes in $\bar{G}_{k+1}$. Call the process of making this identification "subgrid expansion". In this one dimensional case, a connection network that provides the required mapping is a bidirectional omega switch on p processors Ω_p. This switch is constructed from $\log_2(p)$ shuffle-exchange operations. Figure 5 illustrates its performance on the case shown in Figure 4.

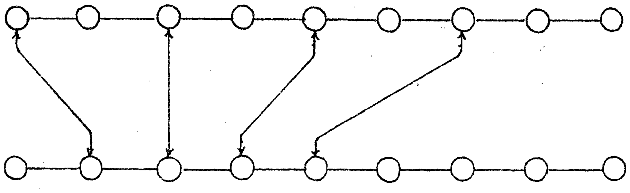


Figure 4. Subgrid Expansion.

LEMMA 2.1. Let $x_1,..., x_p$ denote the sequence of p processors and let $x_k, ..,x_{k+t}$, $t \leq p/2$ denote the indices of a subgrid, then the Ω_p network can map the subsequence to any even or odd subsequence of $x_1,..,x_p$.

PROOF. This result follows from the equivalence of Ω_p and the Batcher bitonic merge network (see for example the thesis of Parker [8]). To use this equivalence, one need only construct the appropriate bitonic sequence that, when merged, maps the subsequence to the appropriate place. Suppose x_k, must map to x_y. It follows that $x_{k+l} \rightarrow x_{y+2l}$ for $l \leq t$. Define the remainder of the bitonic sequence by mapping the remaining processors $x_{(k+s \ mod(p))}$ for $s = t+1,..p-1$ according to

$$x_{(k+s \ mod(p))} \rightarrow x_{y+2s} \text{ for } y+2s \leq p$$

and

$$x_{(k+s \ mod(p))} \rightarrow x_{2p-1-y-2s} \text{ for } y+2s > p.$$

The effect is to extend the increasing subsequence to a permutation consisting of one increasing set of indices and one decreasing set.

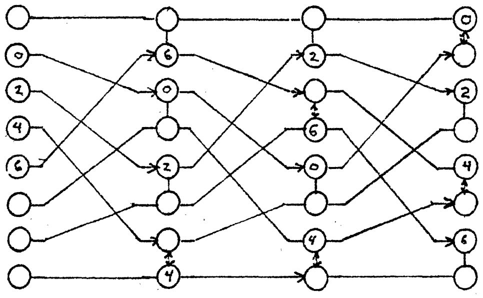


Figure 5. Subgrid Expansion via Ω_p Shuffle-Exchange.

To carry out the complete two dimensional subgrid expansion observe that it suffices to "expand" first along rows and then along columns. Furthermore, the averaging computation can be carried out by data transmissions along the square nearest neighbor network assumed to underlay the embedding of G_{k+1}. In other words, each node in G_{k+1} not lying on G_k is either the bisector of an edge or the center of a square in G_k. By first transmitting values along edges from those nodes in $G_k \cap G_{k+1}$ one can compute values along bisected edges. As illustrated in figure 6, a second broadcast pass is sufficient to propagate values to determine $u^{(k+1)}$ at those node in G_{k+1} forming the centers of squares in G_k. Summarizing we have

Proposition 2.1. Let $2\Omega_p$ be the network composed of p Ω_p switch networks aligned along rows and connected to an identical set of Ω_p networks along the columns of a p by p processor array. If we assume data transmission through an Ω network can be done in one major clock cycle the time for the subgrid expansion is two cycles. With the addition of a nearest neighbor connection capable of broadcasts this combined $2\Omega_p$-*NN* network permits the *injec* operator to be completed in a time of 4 clock cycles. (A more strict interpretation of the Ω_p switch as a $log(p)$ cycle device and the *NN* network capable of only parallel horizontal and parallel vertical data transmissions yields a time bound of $2log(p)+8$).

2.2 Projection.

Given a function $f^{(k+1)}$ corresponding to the data for an elliptic P.D.E. problem on $\bar{G}_{k+1}$ the projection operator determines a function $f^{(k)}$ corresponding to a reduced problem on $\bar{G}_k$. The operator, denoted by

$$f^{(k)} := proj(f^{(k+1)}, \bar{G}_{k+1}, \bar{G}_k)$$

has been interpreted by various authors to mean various things, but Nicolaides [7] has shown that the correct interpretation is that of the transpose of the injection operator. If we permit Ω_p to be bidirectional, the projection operator requires the same data flow patterns as does injection only reversed.

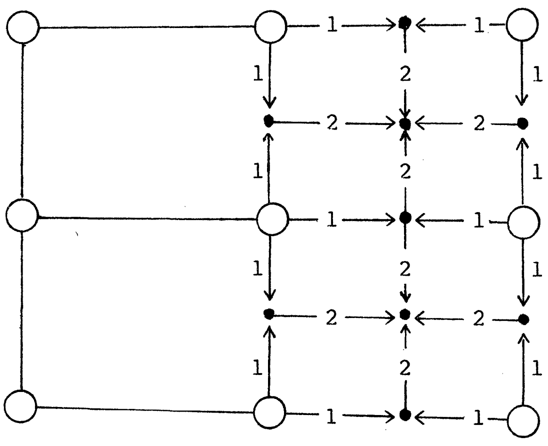


Figure 6. Local Data Flow for *Injec*.

2.3 Relaxation.

At each node x_s of a grid G the differential equation can be replaced by a difference equation

$$\sum_{j \in S_s} a_{sj} u_j = f_s$$

where u_j is the approximate solution value at x_j, S_s is the set of nodes of G serving as vertices of squares containing x_s as a vertex, and a_{sj} are the coefficients of the finite difference approximation (the exact form of which will be of no concern here). The classic parallel relaxation step starts with an approximate solution u_* to the difference equations above an computes an improved solution u'_* by a formula of the form

$$u'_s = u_s + \frac{\lambda}{a_{ss}}(f_s - \sum_{j \in S_s} a_{sj} u_j)$$

for some constant λ known as the relaxation factor. The parallel complexity of this computation is dependant on the structure of the set S_s. For the non-uniform grids described above S_s can take two forms as shown in figure 7.

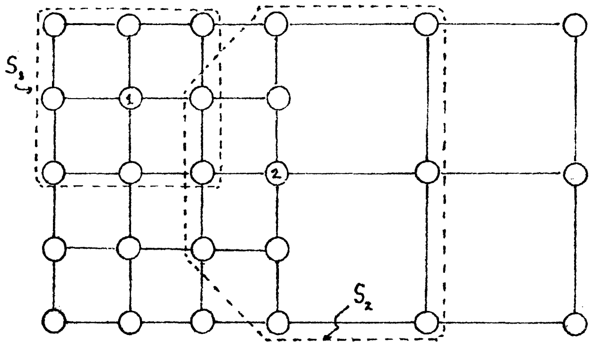


Figure 7. S_j Structures.

In the simplest form x_s is not on the boundary of two subgrids. In this case the nearest neighbor network is adequate to provide all data transmission. In the other case the node is on the boundary of subgrid G_k and the computation should be split with a partial result passed between G_k and G_{k-1} via the $2\Omega_p$ network. More formally, let

$$S_{s,k} = (\, x_j \in S_s \text{ such that } x_j \in G_k)$$

and let $intG_k$ denote the nodes in the interior of G_k and

$$extG_k = G_k - intG_{k+1}$$

The relaxation procedure becomes

```
Proc Relax($u'$, $u$, $\overline{G}_k$ );
  begin
    For $l$ := $k$ downto 0 do
      begin
        For each node $x_s \in ext\, G_l$ pardo
            $u'_s := u_s + \frac{\lambda}{a_{ss}}(f_s - \sum_{j \in S_{s,l}} a_{sj} u_j)$;
        For each $x_s \in G_l \cap G_{l-1}$ pardo
            $u'_s := proj(u', \overline{G}_l, \overline{G}_{l-1})$;

      end;
  end;
```

The exact amount of computation required depends on the form of the finite difference operator. Using the definition of S given above and the strict interpretation of Ω network timing the bound is approximately $(k+1)(log(p)+16)$

2.4 Network Equivalence.

By analyzing the data flow of the basic grid operators we have constructed a network $2\Omega_p-NN$ which can be viewed as an extension of the PS-NN connection introduced earlier. We now show

Proposition 2.2. By numbering the $P = p^2$ processors by columns, the networks $2\Omega_p$ and Ω_P are equivalent. Furthermore, The Nearest Neighbor connections (NN) can each be routed in one pass of the Ω_P network. In terms of the time required to route an item of data through the network and complete one multiply and add operation P processors interconnected by a bidirectional Ω_p have the upper bounds

1. *Injec* (.,.,.,.) **in 9 Ω_P-compute stages.**
2. *Projt* (.,.,.,.) **in 9 Ω_P-compute stages.**
3. *Relax* (u', u, G^k) **in 17k Ω_P-compute stages.**

Proof. Number the processor in row i and column j as $x_{i+p(j-1)}$. To prove the stated network equivalence we again exploit the equivalence of the Batcher Merge network $B(P)$ to Ω_P. The former can be described as $log(P)$ stages with the k^{th} stage defined by the processor connections

$$B_k(P)\colon x_l <---> x_{l+P/2^k},\quad k=1,..,P/2$$

used to execute $P/2$ compare-exchange steps in parallel. Observe that for $l=i+p(j-1)$ we have x_l is mapped to the processor with index

$$l + \frac{P}{2^k} = i+p(j-1) + \frac{p^2}{2^k}$$

$$= i + p(j-1+\frac{p}{2^k})$$

which for $k \leq log(p)$ represents the index of $B_k(p)$ in row i. For $k > log(p)$, we have

$$l + \frac{P}{2^k} = i + \frac{p}{2^{k-log(p)}} + p(j-1)$$

which represents $B_{k-log(p)}(p)$ in column j. Hence the $log(P)$ stages of $B(P)$ can be decomposed into the $log(p)$ stages of $B(p)$ organized along rows followed by the same number of stages of $B(p)$ organized along columns.

In other words,

$$\Omega_P = B(P) = 2B(p) = 2\Omega_p$$

To complete the mapping of $2\Omega_p-NN$ to Ω_P observe that the nearest neighbor network connection in column order has "horizontal shift" equivalent to "shift by p or $-p$; and "vertical shift" is the same as a shift by 1 or -1. But each of the latter uniform shifts are well known to be executable with the Ω connection (see [5]). The set of upper bounds follow from the bounds derived earlier by replacing $2log(p)$ and each unit time NN data move with one Ω_P transmission.

3. MULTI-GRID ANALYSIS.

A parallel version of the "locally refined" Multi-Grid algorithm of van Rosendale is given by an iterative application of the recursive procedure below.

```
  Proc MG(u, f, $k$);
    begin
1       for $i$ := 1 to $t$ do
            begin
              Relax($u'$, $u$, $\overline{G}_k$);    $u := u'$;
            end;
      if $k > 0$ then
        begin
2           for $i$ := 1 to $N_k$ pardo
                $f'_i := f - \sum_{j \in S_i} a_{ij} u_j$;

3           $f^{(k-1)} := proj(f' ,\overline{G}_k, \overline{G}_{k-1})$;

4           MG($u'$, $f^{(k-1)}$, $k-1$);
5           $u := u + Injec(u' ,\overline{G}_{k-1}, \overline{G}_k)$;
6           for $i := t+1$ to $M$ do
                begin
                  Relax($u'$, $u$, $\overline{G}_k$);   $u := u'$;
                end;
7       end else solve exactly on $\overline{G}_0$
    end;

  begin
    (* main *)
8    $u$ := exact solution on $\overline{G}_0$;
    k := index of finest grid;
9    for $i$ := 1 to k do
          begin
            $u := Injec(u, \overline{G}_i, \overline{G}_{i+1})$;
            MG($u$, $f$, $i$);
          end;
   end.
```

The number N_k is the total number of nodes in $\overline{G}_k$, and the loop bounds t and M in lines 1 and 6 are constants depending only on the partial differential equation in a manner discussed below. By starting with an exact solution of the finite difference equations on grid $\overline{G}_0$ (obtained for example, by the Sameh-Chen-Kuck fast Poisson solver) the method produces a sequence of approximate solutions for each grid $\overline{G}_l$ for $l=1..k$. The running time for this algorithm is derived as follows. Let T_k be the time for one call to MG(.,.,k) in terms of network communication-computation steps, where in this section we assume one pass through the Ω switch is $log(P)$ steps and the computation step as unity. From our previous analysis we obtain the recurrence relation

$$T_k := t((k+1)(log(P)+16)) + S + (log(P)+8)$$
$$+ T_{k-1} + (log(P)+8) + (M-t)((k+1)(log(P)+16))$$

where the summands on the right come from lines 1 through 6 of the program and the residual computation (time S) is similar to *Injec*. Solving the recurrence gives

$$T_k = \frac{Mk^2}{2} log(P) + O(klog(P))$$

The main block calls MG(*) as indicated above and resulting in a total time of $O(Mk^3 log(P))$

In order to arrive at an upper bound on the time required to compute a "final" solution, one must ask when an approximate solution to a set of finite difference equations that approximate a PDE is an adequate approximation to a true solution to the PDE. For a given grid structure $\bar{G}_k$ let ε_k denote the difference (in the mean square sense) between the true solution of the partial differential equation and the exact solution to the differential equation. Simply stated, the main result of the numerical analysis [10] is that there exists a constant C independent of G_k such that if $M >$ C then the difference between the approximate solution produced by the MG algorithm and the true solution of the PDE is less than 2ε. To compare two methods we must compare the computation time to produce solutions of comparable accuracy. While the comparison of this version of parallel Multi-Grid to the fast Poisson solver is the currently the subject of several numerical experiments that will be reported on later, it is possible to give a rudimentary analysis of expected performance.

Under optimal conditions on the initial data f and the PDE being solved, the truncation error ε_k is for a non-uniform grid $\bar{G}_k$ is comparable to a n by n uniform grid when $k = c(log(n/p))$ for a small constant $c \leq 2$. In such favorable circumstances we expect performance of

$$O(k^3 log(P)) = O(log^3(n/p)log(P))$$

which compares well with the bound of $O((n/p)^2 log(P))$ for a fast Poisson solver on the uniform grid.

REFERENCES

[1] A. Brandt, "Multigrid Solvers on Parallel Computers", ICASE NASA Langley Research Center, Hampton, Vi. Report No. 80-23, 1980.

[2] A. Brandt, N. Dinar, "Multigrid Solutions to Elliptic Flow Problems", Numerical Methods for Partial Differential Equations, S. V. Parter ed., Academic Press, 1979, pp.53-148.

[3] D. B. Gannon, "Self Adaptive Methods for Parabolic Partial Differential Equations", Department of Computer Science, University of Illinois, Urbana, UIUCDCS-R-80-1020, 1980.

[4] C. Grosch, "The Effect of the Data Transfer Pattern of an Array Computer on the Efficiency of Some Algorithms for the Tri-Diagonal and Poisson Problem", Array Architectures for Computing In the 80's and 90's, ICASE Workshop, April 1980, Hampton, Virginia.

[5] D. H. Lawrie, "Access and Alignment of Data in an Array Processor," IEEE Trans. on Computers, Vol. C-24, No. 12, pp. 1145-1155, Dec. 1975.

[6] R. E. Lord, J. S. Kowalik, and S. P. Kumar, "Solving Linear Algebraic Equations on a MIMD Computer," Proceedings of the 1980 International Conference on Parallel Processing, 205-210, IEEE 1980.

[7] R. A. Nicolaides, "On the L^2 convergence of an algorithm for solving finite element equations," Math. Comp. 31, 1977, 892-906.

[8] D. S. Parker, Jr. "Studies in Conjugation: Huffman Tree Construction, Nonlinear Recurrences, and Permutation Networks," Department of Computer Science, University of Illinois, Urbana, UIUCDCS-R-78-930, 1978.

[9] A. H. Sameh, S. C. Chen, and D. J. Kuck, "Parallel Poisson and Biharmonic Solvers", Computing 17 (1976), 219-230.

[10] J. R. Van Rosendale, "Rapid Solution of Finite Element Equations on Locally Refined Grids by Multi-Level Methods", Department of Computer Science, University of Illinois, UIUCDCS-R-80-1021, Urbana, Illinois, 1980.

[11] P. Zave, W. Rheinboldt, "Design of an Adaptive, Parallel Finite-Element System", ACM Trans. on Math. Software, vol. 5(1), 1979, pp.1-17.

A PRACTICAL ALGORITHM FOR THE SOLUTION OF LOWER TRIANGULAR SYSTEMS ON A PARALLEL PROCESSING SYSTEM

Robert K. Montoye and Duncan H. Lawrie

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

ABSTRACT

An algorithm is presented for a more efficient and implementable solution of lower triangular systems on a parallel (SIMD) computer. Additionally, this algorithm has been mapped to a hypothetical machine with as many memory units as processors, an Ω alignment network, and a control unit that can generate P-ordered memory addresses. Assuming that L is a unit lower triangular system of order N, the system can be solved in T arithmetic operations:

using $P = N^r$ processors,

if $r < \frac{3}{2}$, $T = O(N^{2-r})$

if $\frac{3}{2} \leqslant r < 3$, $T = O(N^{1-r/3} \times \log^{2/3}(N))$.

The data is directly accessible in the evaluation step and can be moved to a location where all required data for the inversion step can be accessed. The memory/processor connections are Ω passable and the processor/memory connections are Ω^{-1} passable. Preliminary error results of a FORTRAN simulation indicate correlation between this and the serial algorithm for both stable and unstable problems.

1. Introduction

This paper shall discuss the limited processor solution of unit lower triangular systems:

($L \times \underline{x} = \underline{f}$ with L of order N).

Time (T) is measured in terms of the number of operations that can be performed using up to P processors performing a single operation on different data and is proportional to the time required for a system to execute the algorithm considering both access and alignment penalties.

In previous papers on this topic:

Chen & Kuck's [ChKu74] "product form" proved:

$L \times \underline{x} = \underline{f}$ could be computed with

$$P = O(N^3) \qquad T = O(\log^2(N)). \qquad (1)$$

Hyfial & Kung [HyKu74] used this with problem partitioning to show that using $P = N^r$ processors,

$$\begin{aligned} &\text{with } \tfrac{3}{2} \leqslant r \leqslant 3 && T = O(N^{1-r/3} \times (\log^2(N))) \\ &\text{with } \quad r < \tfrac{3}{2} && T = O(N^{2-r} \times \log(N)). \end{aligned} \qquad (2)$$

Also of interest is the results of "column sweep" or direct forward substitution method [Kuck76]:

$$P = O(N) \qquad T = O(N) \qquad (3)$$

This research was supported in part by:
National Science Foundation grant MCS-81-00512
and Dept. Energy grant US DOE DE-AC02-81ER10822

To produce a limited processor algorithm, the system is partitioned into s=N/w blocks of width w:

$$\sum_{j=1}^{i} L_{i,j} \times \underline{x}_j = \underline{f}_i \qquad 1 \leqslant i \leqslant s$$

This has the form $L_{i,i} \times \underline{x}_i - \sum_{j=1}^{i-1} L_{i,j} \times \underline{x}_j \quad 1 \leqslant i \leqslant s$.

$\underline{x}_j$ and $\underline{f}_j$ are w element vectors and $L_{i,j}$ is w×w.

$$\begin{vmatrix} L_{1,1} & & & & \\ L_{2,1} & L_{2,2} & & & \\ \cdots & \cdots & \cdot\cdot & & \\ L_{i,1} & L_{i,2} & L_{i,i} & & \\ \cdots & \cdots & \cdots & \cdots & \\ L_{s,1} & L_{s,2} & L_{s,i} & \cdots & L_{s,s} \end{vmatrix} \times \begin{vmatrix} \underline{x}_1 \\ \underline{x}_2 \\ \cdot\cdot \\ \underline{x}_i \\ \cdot\cdot \\ \underline{x}_s \end{vmatrix} = \begin{vmatrix} \underline{f}_1 \\ \underline{f}_2 \\ \cdot\cdot \\ \underline{f}_i \\ \cdot\cdot \\ \underline{f}_s \end{vmatrix}$$

Figure 1 - Hyfial & Kung's Partitioning

The partitioned system can be solved by serially producing each new $\underline{f}_i = \underline{f}_i - L_{i,j} \times \underline{x}_j$, then solving the recurrence for $\underline{x}_i$ using $w = N^{r/3} = P^{1/3}$ and the "product form" [ChKu74] producing (2).

Considering that the product form is performed by explicit calculation of $\underline{x}_i = L^{-1}_{i,i} \times \underline{f}_i$, inversion (which adds an $O(\log^2(N))$ delay) can be performed for all $L_{i,i}$ in parallel, increasing the utilization by moving the $O(\log^2(N))$ term outside the loop on s. The evaluation phase is then a matrix multiplication, producing an O(log(N)) delay.

```
Solve for all the $L^{-1}_{i,i}$ in parallel ;
DO i = 1,s
        $\underline{x}_i = L^{-1}_{i,i} \times \underline{f}_i$ ;
        DO j = i+1,s (* in parallel *)
                $\underline{f}_j = \underline{f}_j - L_{j,i} \times \underline{x}_i$ ;
        END;
END;
```

a) The s $L^{-1}_{i,i}$ can be produced by products of their constituent elementary matrices in:

$$T \leqslant \left(\frac{s \times w^3}{2 \times P}\right) + O(\log^2(w)).$$

b) The resulting system can be solved in:

$$T \leqslant 2 \times s \times \log(w) + O(N^{2-r}).$$

c) Choosing $w_3 = (N^{r/3} \times \log^{1/3}(N))$ produces:

$$\begin{aligned} &\text{if } \quad r < \tfrac{3}{2}, && T = O(N^{2-r}) \\ &\text{if } \tfrac{3}{2} \leqslant r < 3, && T = O(N^{1-r/3} \times \log^{2/3}(N)) \end{aligned} \qquad (4)$$

which compares favorably with (2) above. Additionally, its limiting cases are:

the results of (1) at $P = \left(\frac{N^3}{\log^4(N)}\right)$ and

the results of (3) at $P = O(N)$.

0190-3918/81/0000/0106$00.75 © 1981 IEEE

2. Storage Scheme

The data for the problem is modeled as a two dimensional array stored across the P memories in column major order. Thus elements within a column are in consecutive memories and elements in adjacent rows are in memories that differ by the column length. The processor assignment will be described in this same light. Assuming A is dimensioned A(N,N), any two elements $A(R_a,C_a)$ and $A(R_b,C_b)$, are both accesible if and only if:

$$((C_a-C_b)\times N + (R_a-R_b)) \bmod P \neq 0.$$

A sufficient condition is that:

$$|(C_a-C_b)\times N + (R_a-R_b)| < P$$

as X and X + δ must be different mod P if $|\delta| < P$. Any set of elements whose linearized distance between all pairs is less than P is accessble.

3. Alignment Network

The networks used for data alignment (Ω for input and Ω^{-1} for output) have been extensively studied in [Lawr75], [Wen 77], [Yew 81]. The results needed in terms of the source-destination pairs that will pass a given network of P input and output ports will be listed here.

1) A mapping will pass an Ω network if for all i,j: $(s_i - s_j) \bmod P \leqslant (d_i - d_j) \bmod P$.
2) A mapping will pass an Ω^{-1} network if for all i,j: $(s_i - s_j) \bmod P \geqslant (d_i - d_j) \bmod P$.
3) Both networks are partitionable.
   If the source and destination locations are partitioned into blocks of size 2^J, the mapping is passable if both the mapping within each partition is passable and the mapping of the partitions in the system is passable.

4. Matrix Multiplication

The matrix-matrix multiplication operator performs all the arithmetic for the algorithm. This operator can be scheduled to maximize efficiency and minimize memory and alignment conflicts. The matrix multiplication of A (α by β) × B (β by γ) producing C (α by γ) requires the summation of a dot product of length β for each element of C. If α, β and γ are all powers of two and $\beta \leq \alpha$, the operation can be aligned using the Ω and Ω^{-1} network. Further details are in [Mont81].

5. Parallel Inverse Calculation

Inversion is accomplished using the identity from [Hous64] that the inverse of a unit lower triangular system is expressable as a product of the inverses of its constituent elementary matrices.

$$M_i = M_{i,i} = I - (L - I) \times e_i^t$$

$$L^{-1} = M_{n-1} \times M_{n-2} \times M_{n-3} \times \cdots \times \cdots \times M_3 \times M_2 \times M_1$$

There are s such systems of order w to be inverted. Each system will be fanned in with a binary tree similar to the technique used by Sameh & Brent [SaBr77] in which each level (ℓ) of the tree doubles the number of elementary matrices in one product (2^{ℓ}) and halves the number of products ($2^{\log(w)-\ell}$) being formed. Each elementary matrix product involves the multiplication of a matrix of size w by $2^{\ell-1}$ by a square matrix of size $2^{\ell-1}$ and the addition of a (w by $2^{\ell-1}$) matrix to the result. To efficiently implement this operation, the matrices to be inverted are extracted to a (w by N) array. This allows the data for the inversion to be accessed in at most two passes.

Step ℓ should take

$$T_{\ell} \leqslant 2\left\lceil \frac{(s\times(2^{\log w-\ell})\times(w\times 2^{\ell-1})}{P}\right\rceil + \log(2^{\ell}).$$

The time to solve for all inverses is

$$T \leqslant \sum_{\ell=1}^{\log(w)} T_{\ell} \leqslant \frac{s\times w^3}{2\times P} + 0(\log^2(w))$$

6. Solving the Partitioned System

After the inverses have been computed, the next w elements of the x-vector can be solved and the rest of the f-vector can be updated in two matrix multiplications:

$$\underline{x}_i = L_{i,i}^{-1} \times \underline{f}_i$$

is a matrix (w by w) vector (w by 1) product. Assuming that $P \geqslant \frac{w^2}{2}$, $T = 2 + \log(w)$. The most processor intensive step of:

$$\underline{f}_j = \underline{f}_j - L_{j,i} \times \underline{x}_i$$

is a matrix (N by w) vector (w by 1) product: This should take: $T \leqslant 2\left\lceil\frac{w\times N}{P}\right\rceil + \log(w)$. The total over all s,

$$T \leqslant 2s \times (2 + \log(w)) + 2\times N^{2-r}.$$

Since both the inverted submatrices and the matrix for updating are stored in column major order in partitions of their own dimension, they are accessible and alignable.

7. Total Solution Time

By choosing $w = P^{1/3}\times \log^{1/3}(N)$, the unit lower triangular system with L of order N can be solved using $P = N^r$ processors in time:

if $r < \frac{3}{2}$, $T = 0(N^{2-r})$
if $\frac{3}{2} \leqslant r < 3$, $T = 0(N^{1-r/3}\times\log^{2/3}N)$

using the algorithm previously discussed. The following graph compares Hyfial & Kung's algorithm, product form with folding [Kuck76], and the current algorithm for N = 500 with processing time = 2, alignment time = 1, and memory time = 1.

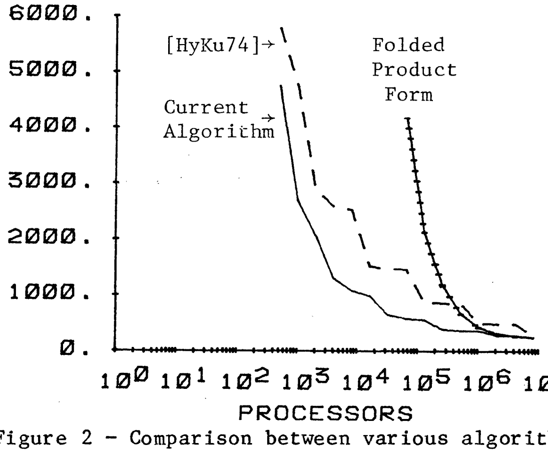


Figure 2 - Comparison between various algorithms.

8. Numerical Experiments

A program that simulated the arithmetic involved in this algorithm was written in FORTRAN. The array indices on the most parallel steps were linear combinations of 5 do-loop indices in parallel (indicating that a control unit that could generate P-ordered vectors of depth 5 could produce the results shown here). The results are obtained by comparing double precision serial solutions with the results derived with this algorithm. The partition width was reduced to w=1 at P=N to force "column sweep" [Kuck76], known to be computationally equivalent to the serial algorithm, to allow comparison with serial algorithms.

The graphs that follow are generated in the following manner. Using a specific lower triangular matrix, an x-vector is generated using a uniform distribution (±1). The matrix is multiplied by this vector to produce an f vector. The difference between the solution generated using the previous algorithm and double precision column sweep is the error. This value is plotted for N = 64, comparing single precision column sweep (using w = 1, P = 64), inverting matrices of size 8 with P = 256, inverting matrices of size 16 with P = 1024, and inverting matrices of size 64 with P = 16384. The number of digits of accuracy is then $-\log_{10}$(relative error). The number of digits of accuracy is then averaged for each element for 25 cases of random x-values with the same matrix. The first matrix represents the 3-term recurrence of the Chebyshev polynomial $\tau(x)$ for x=2.

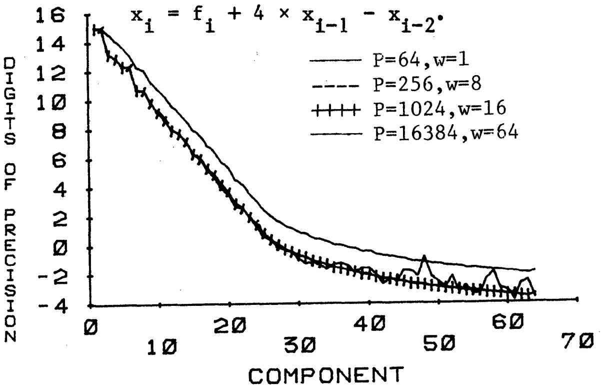


Figure 3 - Comparison of errors for unstable matrix.

The second example is the solution to:

$$x_i = f_i - 1.9 \times x_{i-1} + 0.9 \times x_{i-2}.$$

It is a well-conditioned problem as its characteristic roots lie within the unit circle.

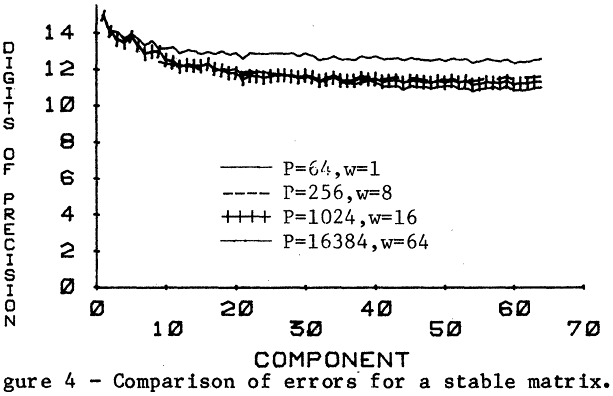


Figure 4 - Comparison of errors for a stable matrix.

9. Conclusion

The method of problem partitioning (first appearing in [HyKu74]) has substantial potential when the additional parallelism that such a method allows is exploited. The parallel inverse calculation places the $O(\log^2(N))$ delay associated with matrix inversion outside of the seriality of solving the partitioned system. This allows:

a) The dominance of the $O(N^{2-r})$ term for $P<N^{3/2}$.

b) The selection of wider partitions
$$w = P^{1/3} \times \log^{2/3}(N) \text{ and as a result:}$$
$$T = O(N^{1-r/3} \times \log^{2/3}(N)).$$

This technique should allow more practical solution of larger general unit lower triangular systems with limited processors. However, of greater concern is the fact that the alorithm can be implemented on a parallel processing system with as many memories as processors and an efficient alignment network connecting them. Additionally, the control requires only generation of P-ordered vectors for memory accesses and lockstep (SIMD) processing. A data storage scheme that allows the algorithm to be executed and data accessed in the same order of time as the theoretical result, by using a scratch data area to store the matrices to be inverted, has been shown. Finally, a small set of experimental error results have been shown to be close to the serial results. In conclusion, a practical, limited processor algorithm for the solution of unit lower triangular matrices has been demonstrated.

10. References

[ChKu75] Chen, S. C. and Kuck, D. J., "Time and Processor Bounds for Linear Recurrence Systems," IEEE Transactions on Computers, Vol. C-24 (1975), pp.701-717.

[Hous64] Householder, A. S., The Theory of Matrices in Numerical Analysis, Blaisdell,New York,1964.

[HyKu74] Hyfial, L. and H. T. Kung, "Parallel Algorithms for Solving Triangular Linear Systems with Small Parallelism," CDS Report, Carnegie-Mellon University, Pittsburgh, December, 1974.

[Kuck76] Kuck, D. J., "Parallel Processing of Ordinary Programs," in Advances in Computers, M. Rubinoff and M. C. Youvits eds., Academic Press, New York, 119-179, 1976. November, 1975.

[Lawr75] Lawrie, D. H., "Access and Alignment of Data in an Array Processor," IEEE Transactions on Computers, Vol. C-24, No. 12, 1975, pp. 1145-1155.

[Mont81] Montoye, R. K., "Simulation of the Solution of Recurrence on a Parallel Processing System," M.S. Thesis, May, 1981.

[SaBr77] Sameh, A. H. and R. P. Brent, "Solving Triangular Systems on a Parallel Computer," SIAM Journal of Numerical Analysis, Vol. 14 #6 December 1977, pp.1101-1113.

[Wen 76] Wen, K. Y., "Interprocessor Connections--Capabilities, Exploitation, and Effectiveness," PhD Thesis, University of Illinois, October, 1976.

[Yew 81] Yew, P. C., "On the Design of Interconnection Networks for Parallel and Multiprocessor Systems," PhD Thesis, University of Illinois, May 1981.

A PIPELINED DIGITAL ARCHITECTURE FOR COMPUTING A MULTI-DIMENSIONAL CONVOLUTION*

K. Y. Liu
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA. 91109

Summary

Two-dimensional (2-D) cyclic convolutions have found many applications such as image processing [1] and synthetic aperture radar (SAR) processing [2], etc. The major problem when one uses the conventional FFT technique to compute the 2-D convolutions is that complicated matrix transpose operation must be performed. To alleviate this problem, several authors [3], [4] have suggested that efficient algorithms using polynomial transforms can be used to compute a 2-D convolution. Recently Reed et al. [5] extended the results given in [3], [4] and developed an efficient algorithm using the radix-2 fast polynomial transform (FPT), the fast Fourier transform (FFT), and the Chinese Remainder Theorem (CRT) to compute a 2-D cyclic convolution. This FPT-FFT-CRT algorithm requires fewer multiplications and about the same number of additions as the conventional FFT approach for computing a 2-D convolution. In [6], the author and Reed et al. proposed a parallel, pipeline architecture to implement this new algorithm for real-time SAR processing application.

In this paper, the work in [5], [6] is further extended to derive a pipelined digital architecture composed of modular FPT, FFT, and CRT computational units for efficiently computing a 2-D convolution. The extension of this machine concept to efficiently compute a multi-dimensional cyclic convolution is also presented in this paper.

Let a_{t_1,t_2} and b_{t_1,t_2} be two $d_1 \times d_2$ arrays, where $0 \leq t_i \leq d_i - 1$ for $i = 1,2$. Then the 2-D cyclic convolution of a_{t_1,t_2} and b_{t_1,t_2} can be expressed as a one-dimensional polynomial convolution [3]

$$C_{n_1}(Z) = \sum_{t_1=0}^{d_1-1} A_{t_1}(Z)\, B_{(n_1-t_1)}(Z) \bmod \left(Z^{d_2} - 1\right) \quad (1)$$

for $0 \leq n_1 \leq d_1 - 1$, where $(n_1 - t_1)$ denotes the residue of $(n_1 - t_1)$ modulo d_1.

In (1) the polynomial $C_{n_1}(t)$, $A_{t_1}(Z)$ and $B_{(n_1-t_1)}(Z)$ are of the form

$$A_{t_1}(Z) = \sum_{t_2=0}^{d_2-1} a_{t_1,t_2} Z^{t_2} \quad (2)$$

If $d_2 = 2^m$, then one can express $Z^{d_2} - 1$ as a product of $(Z^{d_2/2} + 1)$ and$(Z^{d_2/2} - 1)$.

Since these two factors are relatively prime, by the Chinese Remainder Theorem (CRT) for polynomial [7], the polynomial congruences

$$C_1^{(1)}(Z) \equiv C_{n_1}(Z) \bmod \left(Z^{d_2/2} + 1\right) \quad (3a)$$

and

$$C_2^{(1)}(Z) \equiv C_{n_1}(Z) \bmod \left(Z^{d_2/2} - 1\right) \quad (3b)$$

have a unique solution

$$C_{n_1}(Z) = C_1^{(1)}(Z)\left(-\frac{1}{2}\right)\left(Z^{d_2/2} - 1\right) + C_2^{(1)}(Z)\left(\frac{1}{2}\right)\left(Z^{d_2/2} + 1\right) \bmod \left(Z^{d_2} - 1\right) \quad (4)$$

Thus we have decomposed a d_2-point 1-D polynomial convolution into two $d_2/2$-point 1-D polynomial convolutions. Note that in (4) the arithmetic required to compute $C_{n_1}(Z)$ from $C_1(1)(Z)$ and $C_2(1)(Z)$ requires only cyclic shifts and additions.

Applying the same technique to the factor $(Z^{d_2/2} - 1)$ yields the following congruences

$$C_{21}^{(2)}(Z) \equiv C_2^{(1)}(Z) \bmod \left(Z^{d_2/4} + 1\right) \quad (5a)$$

$$C_{22}^{(2)}(Z) \equiv C_2^{(1)}(Z) \bmod \left(Z^{d_2/4} - 1\right) \quad (5b)$$

which can be solved by an equation similar to (4).

*This paper presents one phase of research conducted at the Jet Propulsion Laboratory, California Institute of Technology under Contract No. NAS7-100 sponsored by the National Aeronautics and Space Administration.

0190-3918/81/0000/0109$00.75 © 1981 IEEE

If one uses the transformation $Z = w_1u_1$ given in [4], where w_1 is a $d_2/2$th root of -1, on $C_1{}^{(1)}(Z)$, then (3a) can be expressed as

$$\tilde{C}_1^{(1)}(u_1) = C_1^{(1)}(w_1u_1) = C_{n_1}(w_1u_1) \bmod \left(u_1^{d_2/2} - 1\right)$$

Thus $\tilde{C}_1^{(1)}(u_1)$ can be computed similar to the case given for $C_2{}^{(1)}(Z)$. $C_1{}^{(1)}(Z)$ can be obtained by the inverse transformation $u_1 = w_1{}^{-1}Z$ on $\tilde{C}_1{}^{(1)}(u_1)$. Thus we have decomposed a d_2-point 1-D polynomial convolution into four $d_2/4$-point 1-D polynomial convolutions.

If one repeats the above procedures, then one can decompose a d_2-point 1-D polynomial convolution into 2^i, $d_2/2^i$-point 1-D polynomial convolutions, where i is the level of decomposition. Thus in the computation of a 2-D convolution, the input polynomial $A_{t_1}(Z)$ is decomposed into 2^i polynomials $A_{t_i}{}^{(1)}(Z)$ by moduloing the appropriate polynomials. Each of these polynomials is then convolved with the corresponding polymial $B_{(n_1-t_1)}{}^{(i)}(Z)$ obtained likewise from $B_{(n_1-t_1)}(Z)$. The results of these polynomial convolutions are then combined using the Chinese Remainder Theorem to form the final result $C_{n_1}(Z)$. Since the above technique uses 1-D polynomial convolutions of identical size, modular polynomial convolution and Chinese Remainder Theorem computational circuits can be used as basic building blocks to implement a 2-D convolution system. Moreover, since the computation of these 1-D polynomial convolutions are independent, these convolutions can be done in parallel.

Theoretically, one can decompose a long 2-D convolution into many small and identical polynomial convolutions. However, fast algorithms may not exist when computing a small polynomial convolution of arbitrary size. It was shown in [4] - [6] that when $d_2 = 2^m$ and $d_1 = 2^{m-r+1}$ for some r, $1 \leq r \leq m$, a fast polynomial transform can be used to compute the 1-D polynomial convolutions. Thus when the decomposition level is equal to k, where $1 < k \leq r$, one can use the fast algorithm presented above involving FPT and FFT to compute the 2^k, $d_2/_2 k$-point polynomial convolutions. Of course, when k = r, one can use 1-D polynomial convolutions of the smallest size to compute a $d_1 \times d_2$-point 2-D convolution.

As an example the computational flow diagram of a $d_1 \times d_2$-point 2-D convolution, where $d_1 = 2^{m-r+1}$ and $d_2 = 2^m$ with r = 2, is shown in Fig. 1. Note that the maximum possible decomposition level r = 2 is used. Hence this 2-D convolution is decomposed into 4, $d_2/_4$-point polynomial convolutions, where each of the polynomial convolutions is computed using the fast algorithm discussed above. A pipelined architecture to implement this example is shown in Fig. 2. A detailed description of this architecture is given as follows.

In Fig. 2 the input data is coming in serial word-by-word along the d_2 direction, i.e., consecutive d_2 words are considered as one line along the d_2 dimension in a d_1xd_2 array. The input is controlled by a switch. During the first half of the d_2 points, the switch is in position 1. During the second half of the d_2 points, the switch is switched to position 2. Thus the second half of the d_2-point data is added and subtracted with the first half of the d_2-point data to perform the polynomial modulo $(Z^{d_2/2} - 1)$ and $(Z^{d_2/2} + 1)$ operations required by the first level of the convolution decomposition. The same technique is applied to the two branches of the second level of the convolution decomposition except now a delay of $d_2/4$ is needed to perform the modulo $(Z^{d_2/4} - 1)$ and $(Z^{d_2/4} + 1)$ operations. Also at proper branch of the second level of the convolution decomposition, multiplication by $w_1{}^{\ell_1}$, where $\ell_1 = 1,2, \ldots, d_2/2$, is performed on the input data to perform the transformation $Z = w_1xu_1$. The output of the second level decomposition is fed into the 1-D polynomial convolution which consists of a pipelined FPT [6], a pipelined FFT [1], a multiplier, an inverse FFT, and a pipelined inverse FPT. The constant filter coefficients $B_{t_1}{}^{(k)}(Z)$ is read out from a table and multiplied with the FFT outputs.

The Chinese Remainder Theorem (CRT) computational units shown in Fig. 2 to compute an equation of the form given in (4) can easily be implemented by delay lines and adders. From Fig. 2 one can see that a FFT butterfly type of circuit [1] and serial memories can be used as the basic building blocks to implement the system. With the advent of VLSI technology, such building blocks can easily be implemented on VLSI chips.

The about technique and architecture for computing a 2-D convolution can easily be generalized to compute a multi-dimensional convolution of dimension greater than 2. Let the input data be $d_1 \times d2 \times -- \times d_n$ arrays. Then it can be shown that a fast algorithm similar to the FPT-FFT-CRT algorithm discussed above exists if d_1, d_2, ---, d_n satisfy the following condition:

$$d_n = 2^m,\ d_{n-1} = 2^{m_1-r_1+1} = 2^{m_2},\ d_{n-2} = 2^{m_2-r_2+1}$$

$$---,\ d_2 = 2^{m_{n-2}-r_{n-2}+1} = 2^{m_{n-1}},\ d_1$$

$$= 2^{m_{n-1}-r_{n-1}+1}$$

where $1 \leq r_i \leq m_i$ for i = 1,2,...,n-1.

References

[1] L.R. Rabiner and B. Gold, Theory and Application of Digital Signal Processing, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1975.

[2] C. Wu, "A Digital System to Produce Imagery from SAR Data," AIAA Systems Design Driven by Sensors Conf., Pasadena, California, Oct. 18-20, 1976.

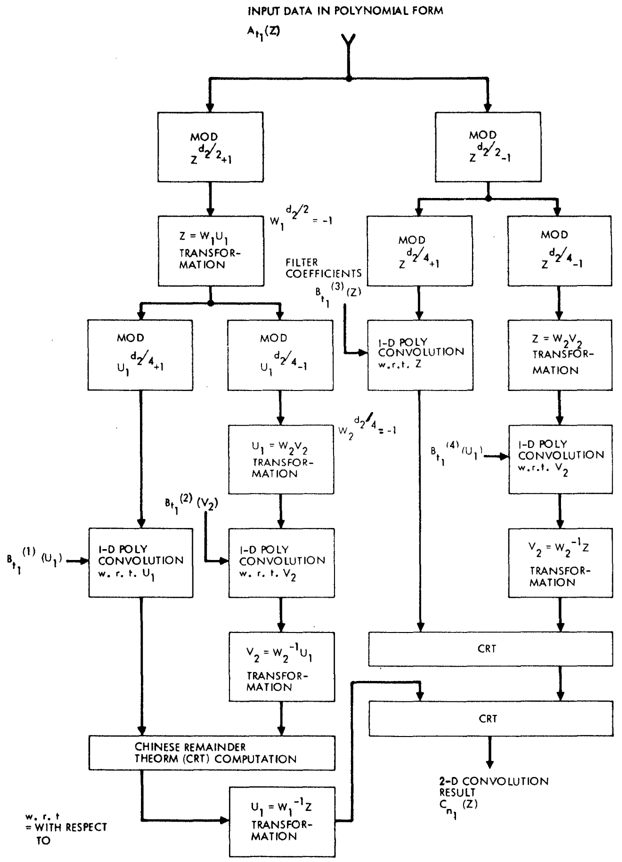
[3] H.J. Nussbaumer, and P. Quandalle, "Computation of Convolutions and Discrete Fourier Transforms by Polynomial Transforms." IBM J. Res. Develop., Vol. 22, No. 2, Mar. 1978.

[4] B. Arambepola, and P.J.W. Rayner, "Efficient Transforms for Multi-dimensional Convolutions." Electronic Letters, 15 March 1979, Vol. 15, No. 6, pp. 189-190.

[5] T.K. Truong, I.S. Reed, R. Lipes, and C. Wu, "On the Application of a Fast Polynomial Transform and the Chinese Remainder Theorem to Compute a Two-Dimensional Convolution." IEEE Trans. Acoust., Speech, Signal Processing, Feb., 1980.

[6] I.S. Reed, T.K. Truong, and K.Y. Liu, "A Parallel, Pipeline Architecture of the Fast-Polynomial Transform for a Real-Time Synthetic Aperture Radar Processor," Proc. of International Computer Symposium, Dec. 18, 1980, Taipei, Taiwan, Republic of China, pp. 1028-1042.

[7] E.R. Berlekamp, Algebraic Coding Theory, McGraw-Hill Book Company, New York, 1968.



INPUT DATA IN POLYNOMIAL FORM
$A_{t_1}(Z)$
MOD $Z^{d_2/2}+1$
MOD $Z^{d_2/2}-1$
$Z = W_1U_1$ TRANSFORMATION
$W_1^{d_2/2} = -1$
FILTER COEFFICIENTS $B_{t_1}^{(3)}(Z)$
MOD $Z^{d_2/4}+1$
MOD $Z^{d_2/4}-1$
MOD $U_1^{d_2/4}+1$
MOD $U_1^{d_2/4}-1$
1-D POLY CONVOLUTION w.r.t. Z
$Z = W_2V_2$ TRANSFORMATION
$U_1 = W_2V_2$ TRANSFORMATION
$W_2^{d_2/4} = -1$
$B_{t_1}^{(4)}(U_1)$
1-D POLY CONVOLUTION w.r.t. V_2
$B_{t_1}^{(2)}(V_2)$
$B_{t_1}^{(1)}(U_1)$
1-D POLY CONVOLUTION w. r. t. U_1
1-D POLY CONVOLUTION w. r. t. V_2
$V_2 = W_2^{-1}Z$ TRANSFORMATION
$V_2 = W_2^{-1}U_1$ TRANSFORMATION
CRT
CRT
CHINESE REMAINDER THEORM (CRT) COMPUTATION
2-D CONVOLUTION RESULT $C_{n_1}(Z)$
w. r. t = WITH RESPECT TO
$U_1 = W_1^{-1}Z$ TRANSFORMATION

Figure 1. Flow Diagram of a d_1 x d_2, 2-D Convolution with $d_2 = 2^m$ and $d_1 = 2^{m-r+1}$, where r=2

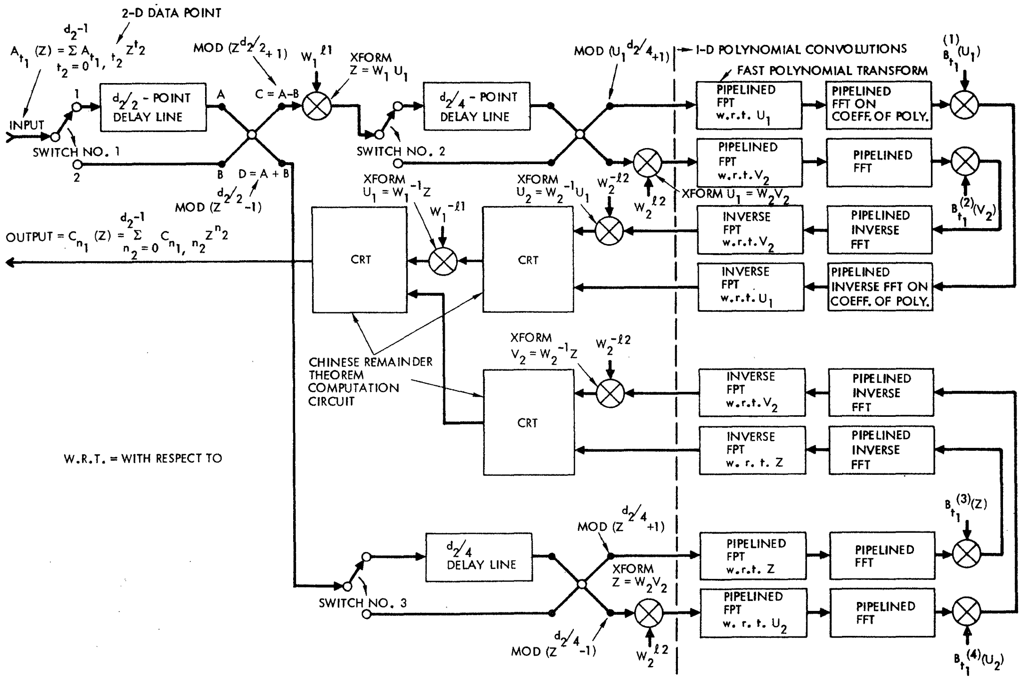


Figure 2. Detailed Implementation of a Pipelined Fast Polynomial Tree for Computing a $d_1 \times d_2$, 2-D Convolution with $d_1 = 2^{m-r+1}$ and $d_2 = 2^m$, where r=2

REAL-TIME LISP USING CONTENT ADDRESSABLE MEMORY *

Jeffrey G. Bonar and Steven P. Levitan
Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

Abstract -- The dynamic data structures of LISP require periodic garbage collection, prohibiting the use of most LISP implementations for real-time applications. We propose a scheme for implementing a real-time LISP system which uses Content Addressable Memory (CAM) to allow incremental garbage collection. In our scheme, all basic LISP operations, notably including retrieving a free cell for CONS, the list building function, and retrieving a current name-value binding, can be implemented with four or fewer CAM searches and very little other computation. Furthermore, CAMs are well suited for sufficiently inexpensive implementation with VLSI technology. Our system is *not* suitable if a virtual memory environment is needed, and becomes considerably more complex with CDR-coding. We are currently implementing a version of our scheme on a microcomputer.

Introduction

There are many *real-time* tasks which lend themselves to *Artificial Intelligence* (AI) solutions. Examples include assembly line robots, rapid transit system controllers, many complex scheduling tasks, and intelligent assistants for interactive devices. Such systems will most likely be designed and tested in LISP. The flexibility and expressibility of LISP have made it the "work-horse" language of the AI community. Can the prototype systems, still written in LISP, then be transferred to the final "production model"? We feel they can, but not with a standard LISP implementation.

The dynamic data structures of LISP require the use of "garbage collection" to reclaim memory as the data structures of the program grow and shrink. Garbage collection is typically done in a two phase process of first tracing and marking all active data, and then collecting all unmarked data. Depending on the size of the memory this operation can cause serious delays in processing. These delays can occur any time the program needs a new free cell. In particular they could occur during time-critical applications. An alternative space management scheme, reference counting, is unacceptable because it allows unbounded delays whenever a cell is released to the free list. This is because *all* successors of the released cell could become garbage and would have to be put on the free list at the same time. For these reasons a standard LISP implementation is not considered acceptable for real-time environments.

In this paper we discuss a real-time LISP implementation. Various LISP machines (e.g. Greenblatt [7] and Deutsch [4]) -- although usually presented as personal computing tools -- have shown that special purpose processors can vastly increase the speed and utility of LISP programs. Our paper shows how special purpose *associative memory* can be used to gain additional benefits.

Following Baker [2] we define a *real-time* list processing system as having "the property that the time required by each of the elementary operations is bounded by a constant independent of the number of cells in use". Baker's real-time LISP system involves incrementally compactifying and linearizing active cells by moving them between two memory partitions while leaving the garbage behind. Wadler [11] analyzes and summarizes a real-time scheme involving two processes running in parallel: the *mutator* is the application program while the *collector* keeps the free-list from becoming empty.

Our scheme uses specialized hardware, Content Addressable Memory (CAM), to create a very fast real-time LISP system, using a very simple set of algorithms. This speed and simplicity, which are the advantages of our scheme, are due directly to our use of CAM to examine all cells in memory *in parallel*.

We begin with a discussion of CAM. After presenting our real-time LISP scheme, its limitations are discussed. Finally, we discuss our implementation of this scheme.

Content Addressable Memory

General Description

Content Addressable Memory (CAM) is memory organized such that each word can compare its *contents*, rather than its address as in random access memory (RAM), with a value broadcast by the central processor [5]. This comparison process is done by all CAM words simultaneously. The processor can then interrogate the CAM to discover which words, if any, match the broadcast value.

Each word of a CAM memory has an associated responder bit (see figure 1). This single bit is *reset* if the contents of the word do not match the broadcast value, held in a register called the

* Support for this work was partially provided by the Army Research Office under grant DAAG29-79-G-0046.

0190-3918/81/0000/0112$00.75 © 1981 IEEE

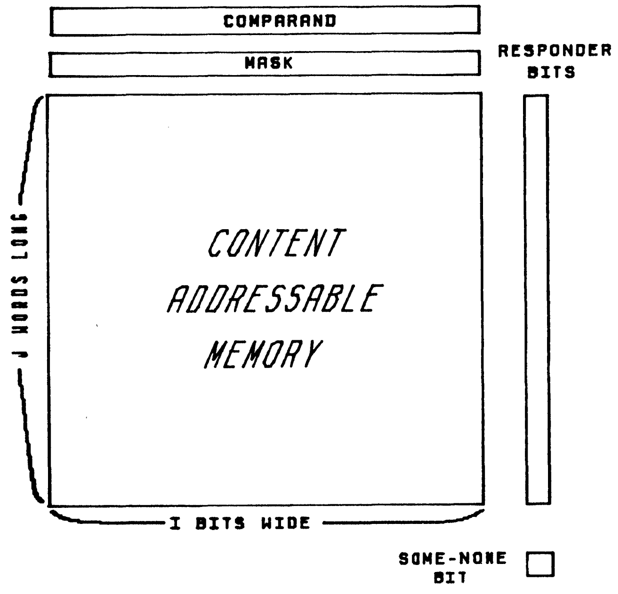


Figure 1. CAM Organization

comparand. All responder bits are typically OR'ed together and their disjunction is available to the processor as the signal SOME-NONE. Using SOME-NONE the processor can determine if there are any words that match the comparand. Additionally, a function to count the number of responders is often provided.

Another function the responder bits provide to the processor is to allow it to select a single responder if more than one exists. This is done by daisy-chaining the responder bits such that when the signal SELECT-FIRST is generated by the processor only the first responder in the chain remains set and all the others are reset. The processor can also perform the function SET-ALL which sets all the responder bits true. This is usually done before the comparand is broadcast to the memory.

Along with the comparand the processor also broadcasts a mask value. This is used by the words of the CAM to determine which bits of the word are to participate. For bits in the word where the mask bit is not set, no comparison takes place. The full operation is:

```
for all Words J
    for all Bits I in Word J
        Responder_bit[J] <-
            Responder_bit[J]
         and
            ( ( Mask_bit[I]
                and
                 CAM_bit[I,J] = Comparand_bit[I]
              )
             or not Mask_bit[I]
            )
```

Note that this operation takes place in all words in parallel.

The processor can also perform the operations READ-RESPONDERS and WRITE-RESPONDERS. These allow the processor to read the contents of and change the contents of all words whose responder bits are set. This operation is often implemented to be under the control of the mask. Finally it is often convenient to allow the processor to access the CAM as a regular RAM and allow reading and writing of single words.

Suitability For Very Large Scale Integration (VLSI)

CAM is well suited to VLSI implementation. Foster [6] and Mead and Conway [10] both discuss the practical design of a VLSI CAM circuit. Two of the most important criteria for determining if a circuit can be implemented efficiently in VLSI are the regularity of circuit components and the number of input/output pins necessary [10]. CAM, like RAM, has an inherently regular sub-structure: the word.

To minimize the pinout (the number of input/output pins needed) several techniques can be used. First both the comparand and the mask values can be broadcast to the CAM in a bit serial protocol. This would mean that comparisons are done one bit at a time across all words in parallel. Bit serial operation would slow down the comparisons somewhat, but only on the order of the number of bits in a word. (a)

To minimize pinout further, the data in, data out, and address lines of the circuit can be multiplexed onto the same pins of the package. This technique has been used successfully for other types of VLSI circuits, for example, the Zilog Z8000 microprocessor. Minimizing the number of pins (and output drivers) would significantly reduce the cost of the circuit and increase the area available for storage.

The cost of CAM has been estimated to be 1.5 to 3 times the cost of an equivalent size RAM [6]. Memory sizes up to 64k of 32 bit words per circuit are not inconceivable [10]. Printed circuit cards containing 4k bytes of CAM have been on the market since 1978 [8].

Finally, CAM architectures lend themselves to a solution of the yield problem for VLSI. The problem is that a single flaw in one place of a VLSI circuit will cause the whole circuit to be unusable. As the physical area of VLSI circuits increases, so does the the probability of a flaw ruining a given circuit [10]. Since CAM operations, unlike RAM operations, do not depend upon where in memory a particular value is stored, it would be possible to disable flawed words of a CAM circuit, after testing, and still use the resulting (smaller) memory.

(a) The time per bit would be on the order of 10 nano-seconds. Therefore, even with bit serial operation, with reasonable word lengths, the time for a CAM operation would be on the order of the time for a machine instruction.

The Ideal CAM for LISP

For most applications CAM words are quite long. The Semionics CAM, for example, has 256 bytes (2048 bits) per word [8]. This allows entire records of data to fit in one word. A record might contain an employee's name, address, telephone number, pay rate, regular hours, overtime hours, etc. This would allow searching on any field of the record to retrieve it. Although there are standard techniques for spreading records across two or more CAM words, this slows the search considerably [6].

An ideal CAM for LISP has much shorter words since it is desirable to have only one LISP cell per CAM word. We discuss several types of LISP cells below. Here we concentrate our discussion on list cells which have seven fields: Flags, Garbage, Cell_type, Left, Left_type, Right and Right_type.

The Flags field is used for complex CAM searches involving logical disjunction and conjunction of different match criteria [6]. The bits in the Flags field are used as "temporary storage" for the responder bit of each word. The Flags field could be replaced by several auxiliary responder bits for each word and CAM operations to logically combine them [6] [8].

The Garbage field need be only one bit, indicating if the cell were "free". Using this bit we completely dispense with the Free list found in most LISP implementations.

The Cell_type field indicates if the cell is a list cell, a string cell, or any one of a number of other types. We discuss this in detail later. The Cell_type will facilitate any desired strength of typing and also allow cells of different types to share the same memory space (without partitioning) and the same garbage collecting scheme.

The Left_type and Right_type fields will also enforce typing. They allow us to pack short integers, bit strings, and pointers to machine language code into the cell. In addition they simplify the garbage collect process by allowing us to test whether a given Left or Right is a pointer.

The Left and Right fields would, as usual, be large enough to point to any other cell in memory. That is, a memory with 2**n CAM words (cells) would require n-bit Left and Right fields.

The CAM operations that need to be supported are SET-ALL, MATCH, SELECT-FIRST, SOME-NONE, READ-RESPONDERS, WRITE-RESPONDERS, READ, and WRITE as outlined above. The COUNT-RESPONDERS is not necessary. Additionally, for the name-value binding scheme outlined below, a FIND-GREATEST function would be helpful.

The Algorithm on a Simplified LISP CAM

We begin the description of our algorithm using a CAM in which each word contains one simplified LISP cell with only three fields: Left (CAR) and Right (CDR), which both point to another LISP cell, and a Garbage bit (see figure 2).

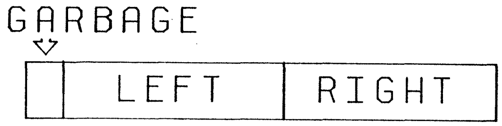


Figure 2. Simplified CAM LISP Cell

The key observation about garbage collection with such a cell is that we can find if there are any pointers to a given cell with two CAM operations: a CAM search of the Left fields and a CAM search of the Right fields, of all cells in memory.

Any practical implementation would use CAM words to hold several different kinds of cells. In particular, our implementation uses special cell types to allow garbage collection of strings, name-value bindings, and the primitives of the GRASPER graph processing language [9]. We discuss how these special cells are handled after presenting the simplified one cell type algorithm.

When a free cell is needed, a CAM search is done for a cell whose Garbage bit is set. This is done by the Supply_free_cell routine in figure 3 (which appears at the end of the paper). One of these cells is selected with the SELECT-FIRST operation. This cell, call it C, is returned as the needed free cell. It is still necessary, however, to propagate "garbageness" to the sub-structures of this cell. This is done by the Potentially_make_garbage routine in figure 3. We do this by first CAM searching the Left and Right fields of all other cells for equality to C.Left. If there are no responders to this search (SOME-NONE has value NONE), then the cell pointed to by C.Left is garbage and we set its garbage bit. If C.Left = NIL, then the search need not be done. We handle C.Right in an identical way. The algorithm requires that all cells be initialized with their Garbage bits set and their Left and Right fields set to NIL.

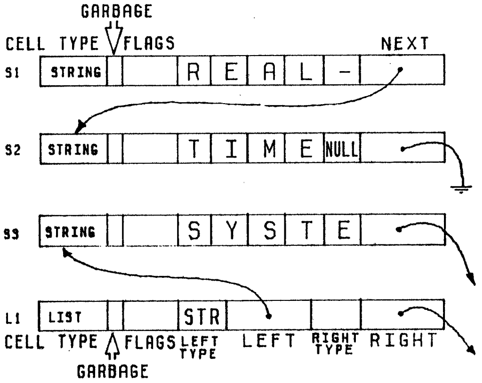
A piece of list structure potentially becomes garbage when one of possibly many pointers to it is deleted. This can occur in several ways during the execution of a program. The functions REPLACA and REPLACD explicitly delete pointers from the left (CAR) and right (CDR) fields of list cells. The function SET (assignment) also deletes the pointer to a variable's old value. These functions all call the routine Potentially_make_garbage on the the pointer they are deleting. This routine determines whether to set the Garbage bit of the head cell of the structure pointed to. All sub-structure will be handled if that head cell is made garbage and when it is actually reused.

Circular lists cannot be garbage collected in our regular scheme because there is always a pointer to any cell in the circle. They can be accommodated, however, either by requiring the user to release them explicitly, or by simulating them with a "lazy evaluation" scheme (see Allen [1] for details on lazy evaluation).

Extensions For Other Cell Types

Our scheme is easily adapted to other kinds of dynamic data structures. Here we will discuss an implementation for strings. Remember that, as discussed earlier, our LISP list cell actually has seven fields. The simplified cell is augmented with a Type field for the cell and for the Left and Right fields. These fields are necessary for the algorithm, but also allow us to enforce typing. Typically, typing is done by putting all of one kind of data together so that address alone can be used to determine type. In our scheme, if a field is of type T, it may only point to a cell of type T.

Strings are made up of linked lists of cells (see figure 4). String cells, like any other cell type, must be fit into the existing size CAM word and must have Type, Garbage, and Flags fields. They also have several bytes of character data and also Next, a Cell_ptr implicitly of type string. The implicit typing saves space in the cell and it does not cause a problem, since string cells can point only to string cells.



The string "REAL-TIME" (S1) and a list cell.(L1) whose CAR points to a string beginning "SYSTE".

Figure 4. Example of CAM LISP Strings

Unlike a list, when the head of a string becomes garbage, the entire string is known to be garbage. Potential "garbageness" need only be propagated down the Next field link and the Other_ptrs_to operation need not be done.

For example, in figure 4, assume that cell L1 is made garbage. When the cell is chosen to be reused, we attempt to propagate "garbageness" to L1.Left. If there are no other pointers to cell S3 the string "SYSTE..." becomes garbage. S3 is marked garbage and when it is reused no other CAM searches need be done.

Atoms are also implemented as special cells. In addition to the Flag, Cell_type, and Garbage fields, atoms have a Value field and Value_type field, pointing to the atom's static binding, and a Print_name field implicitly of type string (that is, pointing to a cell of type string).

A Truly Associative "A-List"

In LISP each function call creates a set of name-value bindings which exist during the execution of the function and disappear at its completion. This is roughly equivalent to the formal to actual parameter bindings in other programming languages. Traditional binding schemes use one or more lists to associate names with values. A list used this way is called an A-List for Association-List (see Allen [1] for more details).

In our scheme the A-list, like the Free list, does not exist. Instead the bindings are held in a set of distinguished cells, existing anywhere in CAM. When entering a new environment, we increment an environment counter and create a set of CAM cells to hold the names bound in that environment, their values, and the new environment number. Now we can ask the question above as a single compound CAM search for a name-value binding within an environment, and retrieve the current binding directly. Since the current value of a name might not be in the most current environment, we need to search for the greatest environment number for that name.

When an environment is exited, a pair of CAM operations is executed. First a search for all environment cells with the current environment number, followed by a WRITE-RESPONDERS operation to make all these cells garbage. Since no other cell will point to these binding cells, even if some do point to their descendants, they can all be turned into garbage in one operation.

Figure 5 summarizes all the cell types discussed in this section.

Garbage, Flags, and Cell_type fields occur in each cell.

List	Left, Left_type, Right, Right_type
Atom	Print_name (implicitly of type string) Value, Value_type
String	Character_1,..., Character_n, Next (implicitly of type string)
Environment	Environment_number, Name (implicitly of type atom), Value, Value_type

Figure 5. Summary of Cell Types

Other Issues

CDR-Coding

Many recent LISP implementations use CDR-coding, compact encodings of list representations which take advantage of statistical regularity in list structures (see Bobrow and Clark [3] for a summary and discussion of these schemes). A CAM augmented LISP with CDR-coded cells is easy to imagine, though it would require considerable extra time and complexity in the implementation of the basic LISP operations. Finding all pointers to a given cell would, in general, require a CAM search for each possible interpretation of a cell pointer field.

Given decreasing hardware costs, we did not feel it necessary to compromise the simplicity and speed of our algorithms. In particular, CDR-coding offers no solutions to our primary goal of real-time operation since it reduces space rather than time needs.

Virtual Memory

Our scheme does not support virtual memory. In general, it would be impossible to perform the test Other_ptrs_to on a given cell without paging every active page of the virtual memory into CAM. The application programs we envision for our system can always be tested in advance to determine their space needs. More CAM cells can always be added without a time penalty.

Our Implementation

We are currently implementing the LISP system discussed above using a Z80-based microcomputer and 80K bytes of CAM. The CAM, Semionics Recognition Memory (REM) [8], is organized as 320 256-byte words (called "super words" in the company literature). We do not need such long words and have cut the memory into vertical slices, yielding 32 LISP cells per word. Although this means that many of our CAM operations will have to be repeated 32 times in the worst case (once for each vertical slice), the system runs at an acceptable speed. The real-time properties of our system remain intact.

The project is a pilot study to examine two issues. First we wish to show that even with relatively slow CAM (bit serial searches on the order of 1 micro-second per bit) which is not organized to our needs, we can build a real-time, self-contained LISP system.

Second, the graph processing language GRASPER uses many associative operations which can be supported by CAM. (b) GRASPER objects have the same dynamic allocation needs as other LISP objects. We will embed a subset of the GRASPER language into our LISP system using the cell typing conventions already discussed. We expect to show the advantages of a CAM based GRASPER system as part of a feasibility study for the design and implementation of a state of the art CAM on our VAX 11/780.

Conclusions

We have presented a scheme for implementing a real-time LISP system by using Content Addressable Memories for storage of the basic LISP cells. Not only does our scheme perform all elementary operations in real-time, it also has the following other advantages:

1. All cells are available for use, in contrast to other real-time schemes.

2. Retrieving the correct value for a name can be be done truly associatively, always requiring only two CAM operations.

3. Strings and other dynamic data types can be elegantly and efficiently integrated into the basic scheme without partitioning memory.

4. CAM is eminently suited to modern VLSI implementation techniques.

Our scheme does have limitations, however:

1. Circular lists cannot easily be garbage collected.

2. Our scheme does not lend itself to a virtual memory environment.

We believe that even given the above limitations, our scheme is an attractive alternative for self-contained, dedicated systems. It is usable in a real-time environment and all basic LISP operations perform extremely quickly. We believe that tested AI systems written in LISP could be transferred to a CAM-augmented LISP machine without costly redesign and without recoding in a standard systems programming language (e.g. assembly language or Ada). In this way we hope our scheme will aid in the creation of simpler yet more powerful computer-controlled systems.

(b) GRASPER is used to represent and operate on semantic nets, augmented transition networks (ATNs), HEARSAY-II style blackboards, and other associative data structures used by AI projects at the University of Massachusetts.

Acknowledgements

We would like to thank Caxton Foster for his wealth of knowledge about CAMs and "hardware solutions". Raj Wall is working with us to implement the system. Our interactions with him have been very valuable. John Lowrance provided us with much information about the basic implementation and operation of LISP and GRASPER. We would also like to thank Jeff Conklin, Dan Corkill, Elliot Soloway, Bev Woolf, and the referees for their careful reading of earlier drafts.

References

[1] John Allen, Anatomy of LISP, McGraw-Hill Book Company, (1978), pp. 149-153.

[2] Henry G. Baker, "List Processing in Real Time on a Serial Computer," Communications of the ACM, (April, 1978), pp. 280-294.

[3] Daniel G. Bobrow, and Douglas W. Clark, "Compact Encodings of List Structure," ACM Transactions on Programming Languages and Systems, (October, 1979), pp.266-286.

[4] L.P. Deutsch, "A LISP Machine With Very Compact Programs," Proceedings 3rd IJCAI, Stanford, California, (1973), pp. 697-703.

[5] Caxton C. Foster, Computer Architecture, second edition, Van Nostrand Reinhold Co., (1976).

[6] Caxton C. Foster, Content Addressable Parallel Processors, Van Nostrand Reinhold Co., (1976).

[7] R. Greenblatt, LISP Machine Progress Report, AI Lab. M.I.T., Cambridge, Massachusetts, memo 444, (August,1977).

[8] Sydney Lamb, "An Add-In Recognition Memory For S-100 Bus Microcomputers-Parts 1,2, and 3," Computer Design,(August-October, 1978).

[9] John D. Lowrance, GRASPER 1.0 Reference Manual, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts, Report 78-20, (December,1978).

[10] Carver Mead, and Lynn Conway, Introduction to VLSI Systems, Addison-Wesley Publishing Co., (1980).

[11] Philip L. Wadler, "An Analysis of an Algorithm for Real Time Garbage Collection," Communications of the ACM, (September, 1976), pp. 491-500.

Figure 3. Algorithm for CAM Augmented LISP Garbage Collection

```
function Supply_free_cell : Cell_ptr;
    (* called by CONS to find a cell it can use
       to build a list structure with. In addition
       this function does the incremental garbage collect *)
    var Free_cell, Temp : Cell_ptr;
    begin
    search for first Free_cell from Cell
        where Cell[Free_cell].Garbage
        do begin
           if Cell[Free_cell].Left <> Nil_ptr
               then begin
                    Temp := Cell[Free_cell].Left;
                    Cell[Free_cell].Left := Nil_ptr;
                        (* These two make sure a check for other
                           pointers = Cell[Free_cell].Left will
                           not respond to that field itself *)
                    Potentially_make_garbage (Temp)
                        (* propagate "garbageness" *)
                    end;

           if Cell[Free_cell].Right <> Nil_ptr
              then begin
                   Temp := Cell[Free_cell].Right;
                   Cell[Free_cell].Right := Nil_ptr;
                       (* These two make sure a check for other
                          pointers = Cell[Free_cell].Right will
                          not respond to that field itself *)
                   Potentially_make_garbage (Temp)
                       (* propagate "garbageness" *)
                   end;
           return Free_cell
           end

        else System_error ("Cell space full")
    end;



procedure Potentially_make_garbage (C : Cell_ptr);
    begin
    Cell[C].Garbage := not Other_ptrs_to (C)
    end;



function Other_ptrs_to (C : Cell_ptr) : boolean;
    var Responder : Cellptr;

    begin
    search for Responder from Cell
        where         not Cell[Responder].Garbage
                 and  Cell[Responder].Left = C
        do return true
        else search for Responder from Cell
                    where       not Cell[Responder].Garbage
                            and Cell[Responder].Right = C
                    do return true
                    else return false
    end;
```

(continues on the next page...)

(...figure 3 continued)

```
procedure Init_CAM;

   var Responder : Cell_ptr;
   begin
   search for Responder from Cell
       where true
       do begin
          Cell[Responder].Garbage := true;
          Cell[Responder].Left := Nil_ptr;
          Cell[Responder].Right := Nil-ptr
          end
   end;
```

Notational Conventions

The CAM is seen as an "associative" array of records, where each record represents the data in one CAM cell. Standard indexing into the array allows us to treat the CAM as RAM. From the above we have two data types:

```
Cell_ptr = 1..Num_cells;


Cell = associative array [Cell_ptr]
          of record
              Garbage : boolean;
              Left, Right : Cell_ptr
             end
```

The basic CAM operation is:

```
search for [ first ] <index variable into CAM>
    from <CAM array name>
    where <boolean expression>
    do <statements>
    else <statements>
```

The <index variable> is available within the do <statements> to syntactically represent all cells that meet the search criteria. This <index variable> is a free variable ranging over all possible values, that is, indexing all cells in the CAM array. For each CAM cell where the <boolean expression> is satisfied, the do <statements> are executed. The do <statements> are performed in parallel for these cells. In the case of "search for first", the index variable gets set to the value of the first responder. In the case that no cells satisfy the <boolean expression>, the else <statements> are executed. Typical CAMS do not support the generality implied by this construct. In particular, arbitrarily complex <boolean expressions> will take N CAM searches, where N is the number of disjuncts in a disjunctive-normal-form version of the <boolean expression>, and do <statements> are limited to assignments to the cells indexed by the <index variable>. Other operations can be supported either by more intelligent CAM cells or by a micro-coded CAM controller. Our algorithms use the construct in ways easily implemented in CAM.

THE M.A.P. PROJECT
AN ASSOCIATIVE PROCESSOR
FOR SPEECH PROCESSING

*

V. CORDONNIER - L. MOUSSU

University of Lille
(FRANCE)

*

ABSTRACT

MAP is an associative multiprocessor of medium size. It has been designed for experimentation in pattern recognition area - especially speech recognition. The machine is composed of sixteen microprogrammable processors. At the microprogram level, every processor is autonomous and can perform its task without receiving any external command. At the collective level, control is assumed by an extra master processor. This processor is concerned with Input-Output and common orders distribution. The architecture presents special accomodations for synchronization between processors. Some of them are driven by an associative arrangement. The total instruction rate is 68 MIPS, allowing a real time processing of the speech.

INTRODUCTION

The architecture of a multiprocessor machine must optimize, both data and instruction flows. Often, these two goals appear to contradict each other. However some facilities may occur when these flows are driven with a good regularity or repetition of simple pattern. Particularly, when a unique model of control distribution and data manipulation may be taken as a general representation of the behaviour of the processors, the architecture may be designed according to it (for example - vector computing with SIMD architecture).

In the most general case it is quite impossible to find out such a model and, accordingly, to obtain a satisfying balance between two constraints :

- availability of a flexible control scheme for parallelism able to support distributed algorithms

- realization of a fast and simple communication tool between processors.

The first goal implies the design of independant and autonomous processors but, conversely, represents a difficulty for getting an easy solution for the second one. Communications have to be localy controled by each processor according to a communication protocol. Then, data transfers, are complicated and slow.

So, when studying a special purpose architecture the first step is to point out the regular properties of the application involving facilities in control distribution and data.

Pattern recognition applied to voice analysis has two typical characteristics :

- Input data flow is strictly sequential and periodic
- The amount of data to be held at a time is not very large and may be easily ordered.

THE SPEECH RECOGNITION CONTEXT

The most usual way to drive a speech recognition process is to use a mathematical representation derivated from signal processing models [11].

The aim of the project is to use an associative model related with a data base organization [4].

Speech processing may use as an input unit, a channel analyser. It is composed of sixteen input filters distributed along the voice spectrum. At every sampling period, a filter issues a digital value in proportion to the quantity of energy received in the channel.

According to the noise and the limits of precision, one value may be represented by a binary positive number of height bits. Then a sample is a 16 bytes vector or a 128 bits word. Period may be taken between 10 and 50 ms.

The input data flow may be looked as a two dimensions array in a timefrequency diagram (fig. 1). [9][10].

At the phonetic level, the element to be identified is named "phonem" and represents a typical sound produced by the speaker. [12][13].

A phonem stretches itself in the two directions time and frequency -as a fuzzy pattern- it seems to be possible to recognize such a pattern by comparison with models which have been stored in an associative memory [2][8]. Unfortunatly the direct comparison is impossible and it is necessary to extract from the input flow some characteristic informations such as :

- mean value
- peak location
- ratios in upper and lower frequencies
- measurement of relief
  etc ...

These informations come out from an horizontal (time) or vertical (frequency) or mixed analysis [3].

Using these informations, a process must follow various tracks among the stored patterns used as references. It has to compute a dynamic score for each of them and to decide :

- rejection of a bad candidate
- acceptance of one or several good candidates (a choice will be done at the upper lever referring to syntax or semantics)

0190-3918/81/0000/0120$00.75 © 1981 IEEE

- pursuing the operation with the following samples
- activate new candidates.

Although MAP is designed for experimentation rather than for exploitation, the previous considerations seem to be general and lead the organization of storage. The informations used as references are sets of samples. These sets are organized in files. Some files are time indexed and represent phonems. Some files are type indexed and gather all the samples which have similar properties or measurements (fig. 2).

A reference sample located in the date memory may belong to several files and the associative process will have to follow various links before idntifying a phonem. Consequently this memory must present the following characteristics :

- basic items are samples (128 bits)
- there is a need for a fast (parallel) access to one sample.
- facilities must be provided for multifiles description (linkage).

It is obvious that a multiprocessor is adequate for such a processing [5], [6] ,[7] including :

- parallel computing in order to extract significant informations from unknown samples.
- parallel access to models of patterns.
- comparisons between vectors and measurement of distances.

GENERAL DESCRIPTION OF MAP

The processing model derived from this application may be described by the flow of fig. 3.

MAP has been designed from this model with two control levels. It is composed of sixteen 8 bit microprocessors. The Low Level Control (LLC) is local to the PE and brings facilities for autonomous processing. The High Level Control (HLC) is unique and has to drive, organize and synchronize collective activities.

At the High Level a single control unit issues general commands that are identified at the same time by the PEs. A general command is initiated when the former one has been achieved by all the processors. From this point of view, the machine seems to have an SIMD architecture with a sequential running of the program.

At the Low Level, a specific program is located in the control memory of each processor. So every processor is able to perform its own and particular part of the task. A processor must take into account :

- its own location
- its status (resulting from previous operations)
- informations produced by neighbours.

Two buses allow communications between HLC and LLC. The command bus is provided for distributing general commands or common data in parallel. The control bus is organized in a polling-selecting manner and driven by the HLC processor. By this means the HLC processor may observe closely the activity of the PEs and pick out final results.

Every processor may access two routing registers. The first one -128 bits- may be shifted to the right along all the sixteen PEs. The other one -136 bits- may be shifted to the left via the HLC processor. Routing operations are controled by the processors themselves. Two neighbour processors or a consecutive set of processors may request a partial use of these buses for local communications.

The storage is divided in two parts : one processor possesses its own control memory. According to the characteristics of the chip - 8 × 300- this control memory is a 4K - 16bits RAM- . During processing, this memory cannot be altered and is used as a ROM.

Data memory is organized in a 128 bits wide - 16 K words store. Every slice of 8 bits is dedicated to one processor. There are 9 adresses producers : every pair of processors PLUS the HLC processor may access the data memory through a priority encoder. This unit is provided for conflicts management but, most of the time, these conflicts may be avoided by synchronization at the LLC.

They are mainly two types of informations to be stored in the data memory :

- voice samples represented by sixteen ordered bytes
- linking informations, that is to say, addresses represented by 8 double bytes. One word of memory contains 8 links, thus a sample described by this word may belong to 8 different files.

As these linking informations are used to construct complex data structures between reference samples, the data memory is seen as a special purpose, read only, data base with a fastened access and a limited capacity.

THE PROCESSING ELEMENT

In spite of an appearance of choice, they were not a great amount of possibilities for the microprocessor of a processing element :

- a custom designed processor was rejected because of the delays
- rapidity is a major argument
- ability for microprogramming is important
- data manipulations are considered to be more interesting than computing possibilities

The typical architecture of 8 × 300 from Signetics seemed to present the best characteristics for these criterions. [1]

A processing element is composed with :

- CPU : 8 × 300 - 250 ns for one instruction
- three registers for memory control
- four registers for routing
- two registers for exchanges with the control and command buses
- two registers for sorting
- two registers for synchronization
- four 16 K bits static RAM_S arranged in 4 K - 16 bits store.

All the program is loaded into the RAM before starting, though a special loading bus. This program is composed of :

- a general command analyser
- various sequences corresponding to the commands. The maximum number of sequences is 255. They are initiated by HLC
- synchronization and communication procedures.

During a sequence a processor is able to access the data memory, to exchange informations with its neighbours, to receive and send informations from or to the HLC processor, to present and accept synchronization demands and, of course, to perform local computations. A macro-assembler bring facilities for writing the sequences in parallel.

Fig. 4 show the architecture of a PE and fig. 5 is a simplified representation of the program organization.

In order to increase the performances of collective operation a wired sorting unit has been added to the processors. This unit gives at any time the maximal value among those presented in parallel by all the processors. This SORTER is a tree and returns to all the processors the number of the winner.

It takes three instructions (750 ns) before getting the results of a sorting operation :

MORE VALUE TO RSORT
COMPARE WINNER'S CODE TO LOCAL CODE
JUMP IF NOT EQUAL

Many general purpose sequences have already been written, let us give some examples (I is one instruction or 250 ns) :

- compute the mean value rounded in one byte (12 I)
- compute the location of gravity center (16 I)
- compute the moment of inertia with regard to a processor (28 I)
- find the best ressemblance between a given sample and a file of ℓ references with distance ;

$$d = \sum_{1}^{16} |x_i - r_i| \rightarrow (5 + 10\,\ell).I$$

- find the best ressemblance with distance

$$d = \sum_{1}^{16} \sqrt{x_2^2 - r_2^2} \rightarrow (5 + 18\,\ell).I$$

The general control processor is also a 8 × 300 module. Over and above the communications with the PE_S it has a private memory used as a general control store. This memory is shared in a multi-access arrangement with a conventional processor. Because of the low rate of the inputs and outputs, a microprocessor is sufficient, then the HLC processor has only to search and distribute general commands.

This host processor also has two extra roles

- load the PE_S programs before executing a program
- compile new programs to be loaded from macro-assembler to 8 × 300 machine language.

Fig. 6 shows the architecture of the whole system.

THE SYNCHRONIZATION UNIT.

Because it is the most important part of the distributed control, the synchronization unit will be described in detail.

There are two occasions where processors must execute in a synchronous way :

- at the end of a LLC sequence in order to obtain a new general command from the HLC processor
- before communications, sorting operations or memory accesses.

All the processors must have exactly the same behaviour during the operations because all of them are working at the same level. For this matter the synchronization is designed according to an associative model.

The first family of synchronization tools is applied to well delimited sets of processors. For one given set each processor K has two flags :

- a D_K Flag used as an output device (demand of synchronization in the set)
- a C_K Flag used as an input device (command of synchronization for the whole set).

The logical relationship between these flags is easily realized with a unique AND circuit :

$$C_K = D_1 \cap D_2 \dots \cap D_K \cap \dots \quad D_n \quad K = 1,n$$

Synchronization occurs when each of the PE_S of the set execute the same sequence of instructions :

```
                SET $D_K$ = TRUE ;
        WAIT : WHILE $C_K$ = FALSE GOTO WAIT ;
        NEXT : ...................... ;
```

As a common clock drives all of them, the processors are going to execute the NEXT labeled instruction at the same time. This operation is possible because the 8 × 300 processor is able to perform in one instruction the test of C_K and the corresponding jump.

Tools have been wired for the following sets :

Sets of two PE_S : $(P_0P_1)(P_2P_3)...(P_{14}P_{15})$: Flag D_1
$(P_1P_2)(P_3P_4)...(P_{15}P_0)$: Flag D_2

Set of four ` $(P_0P_1P_2P_3)..(P_{12}P_{13}P_{14}P_{15})$Flag D_3

Set of sixteen $(P_0P_1..........(P_{14}P_{15})$: Flag D_4

Set of seventeen $(P_0P_1..........$ $P_{14}P_{15}$ plus the HLC processor) : Flag D5
this latter set is particularly used before getting a new general command.

Another manner to obtain synchronization between groups of processors consists of a dynamic construction of the group. The interest of such a tool is to allow synchronization by observing the results of processing rather than the location of processors. This is necessary within an associative process when processors may issue some specific result, the value of which is significant for driving cooperation between them. Namely a subset of the network may request a synchronization because every processor of that subset holds a typical result while all the others do not.

For that purpose, a processor may display one of the heigt names (0 to 7).

Names 0 and 1 have special meaning :
0 : the choice of a synchronization name has not yet been done
1 : no synchronization required
2,7 : effective synchronization names.

The management of these names is realized according to the following rules :

(a) if there exist, at least, one synchronization name equal to 0 no synchronization is possible

(b) in order to allow other groups to synchronize, a processor must display its choice (≠ 0) as soon as it is in position to do

(c) displaying a name is not realy a request for synchronization ; the request is represented by an extra D flag. (D6)

(d) among the processors that have displayed the same name, and AND circuit is dynamicaly provided and delivers the C command when all the D flags have been switched on.

Realization is quite simple and entirely static. First a gate is provided in order to take in charge the (a) rule.

As there are six names, each of them is controled by one AND gate. Every gate is controled by all processors through a network driven by names. This network must decide whether one name, for one processor, is active or not.

Fig. 7 presents an illustration of some usual cases of synchronization.

CONCLUSION

It is easier to design a special purpose processor than a general purpose one. The behaviour of programs is more closely identified and a specific model of instructions and data flow may be established. Accordingly, the architecture is more sophisticated and the performance increased.

In the MAP project these considerations gave the possibility to take advantage of two points :

- the main data structure is a fixed vector.
- the control may be separated in a high (general) and a low (local) level.

The former point imposed to realize a very flexible synchronization system between processors. Such a system brings a great facility for writing parallel programs.

This study was supported by CNET, the French administration for Research in Telephone and Telecommunication area. The machine in now under test and must be operationnal in a few weeks.

BIBLIOGRAPHY

[1] SIGNETICS - *8 X 300 reference manual - 8T32 reference manual.*

[2] S.S. YAU, H.S. FUNG - ***Associative processor architecture : A survey.*** Computing surveys Vol. 9, n° 1, March 77, pp 3-27.

[3] C.A. FINNILA, H.H. LOVE - ***The associative linear array processor.*** IEEE Transactions on computers, Vol. 26, Feb 77, pp 112-125.

[4] D.C.P. SMITH, J.M. SMITH - ***Relational data base machines.*** Computer, vol. 12, march 79, pp 28-38.

[5] P.J. SADOWSKI - ***Exploiting parallelism in a relational associative processor.*** 4th Workshop on Computer Architecture for non numeric processing, ACM Syracuse University, Aug. 78, pp 99-109.

[6] J.R.CARLBERG - ***A paged hardware associative memory.*** David W. Taylor Naval Ship

Research and Development Center
BETHESDA MARYLAND, Aug. 77.

[7] A.D. FALKOFF - ***Algorithms for parallel search memories.*** Journal of the ACM 9, 4, Oct. 62, pp 488-511.

[8] S.S. YAU, C.C. YANG - ***Pattern recognition by using an associative memory.*** IEEE Transactions on computers 15, n° 6, Dec. 66, pp 944-947.

[[9] H.F. SILVERMAN, N.R. DIXON - ***The Modular Acoustic processor.*** IEEE Transactions on Acoustics speech and signal processing 25, n° 5, Oct. 77, p 367.

[10]L. MOUSSU - ***Modèle fonctionnel de mémoire associative ; application au traitement de la parole.*** Thèse 3ème cycle Feb. 81.

[11]J.L. FLANAGAN - ***Speech analysis, synthesis and perception.*** Springer Verlag, NY. 1972.

[12]R. de MORI - ***Recent advances in automatic speech recognition.*** Int. Journal Conference on pattern recognition KYOTO, Apr. 78.

[13]P. QUINTON - ***Contribution à la reconnaissance automatique de la parole. Utilisation de méthodes heuristiques pour la reconnaissance des phrases.*** Thèse d'Etat, Rennes 1980.

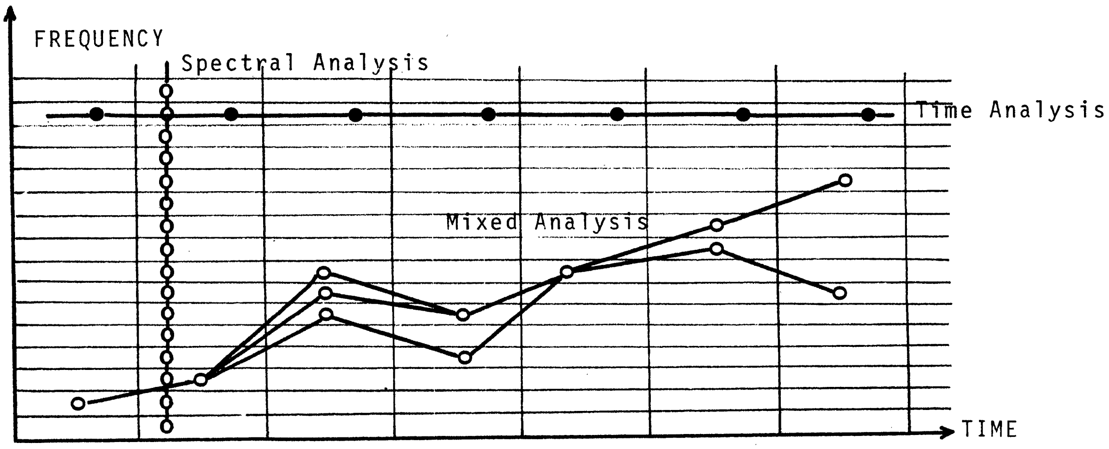


FIGURE 1 : THE TIME-FREQUENCY ARRAY.

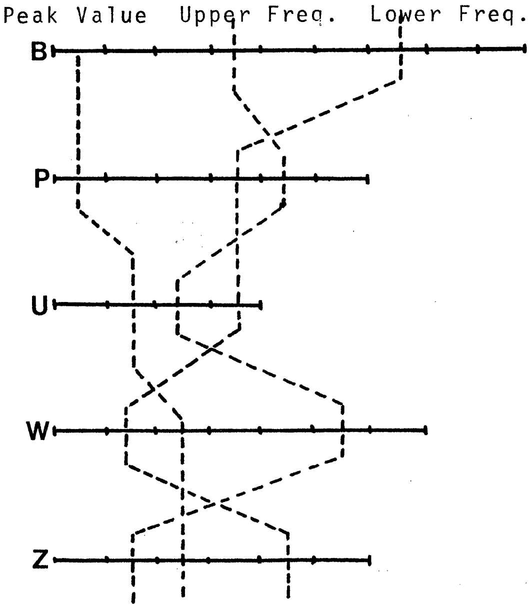


FIGURE 2 : DATA ORGANIZATION

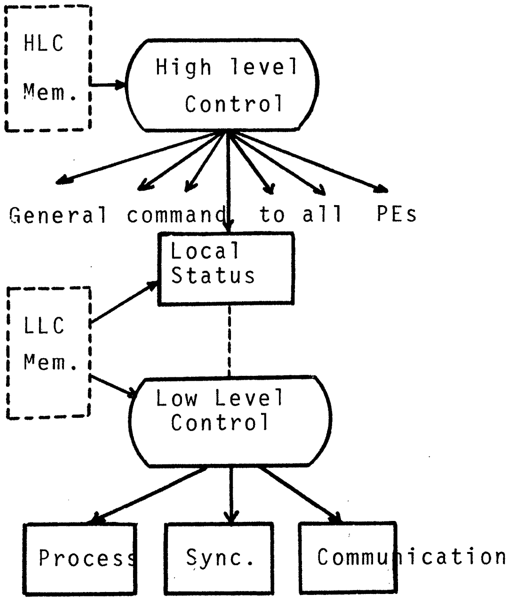


FIGURE 3 : CONTROL DISTRIBUTION

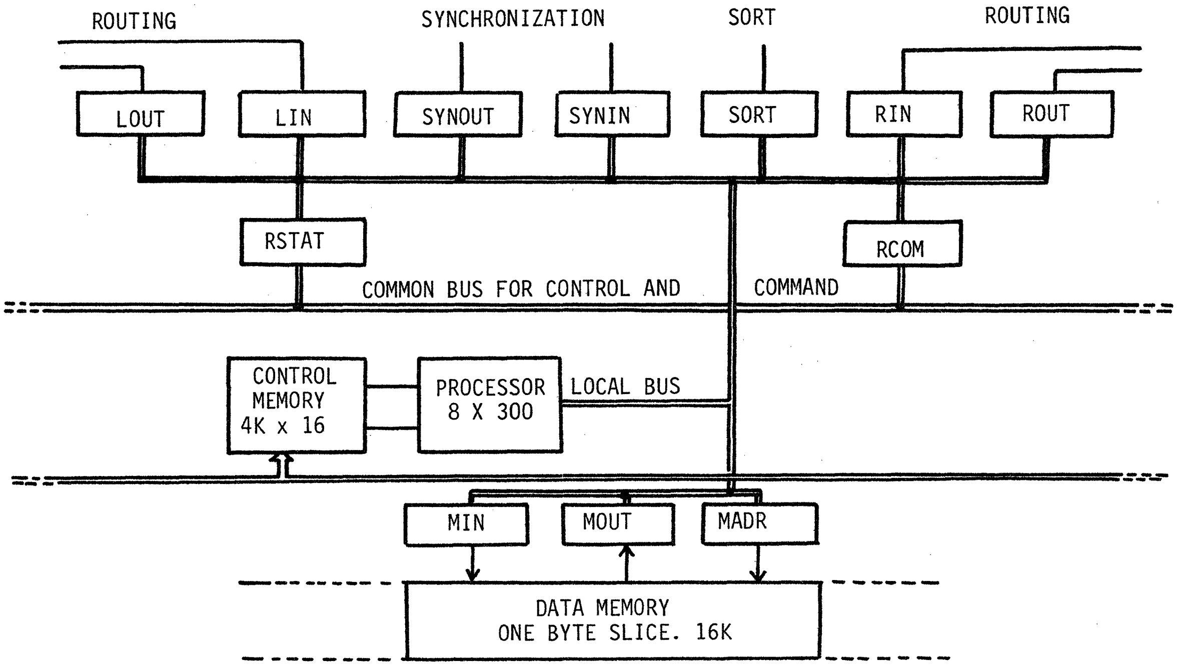


FIGURE 4 : THE ARCHITECTURE OF A PROCÈSSING ELEMENT.

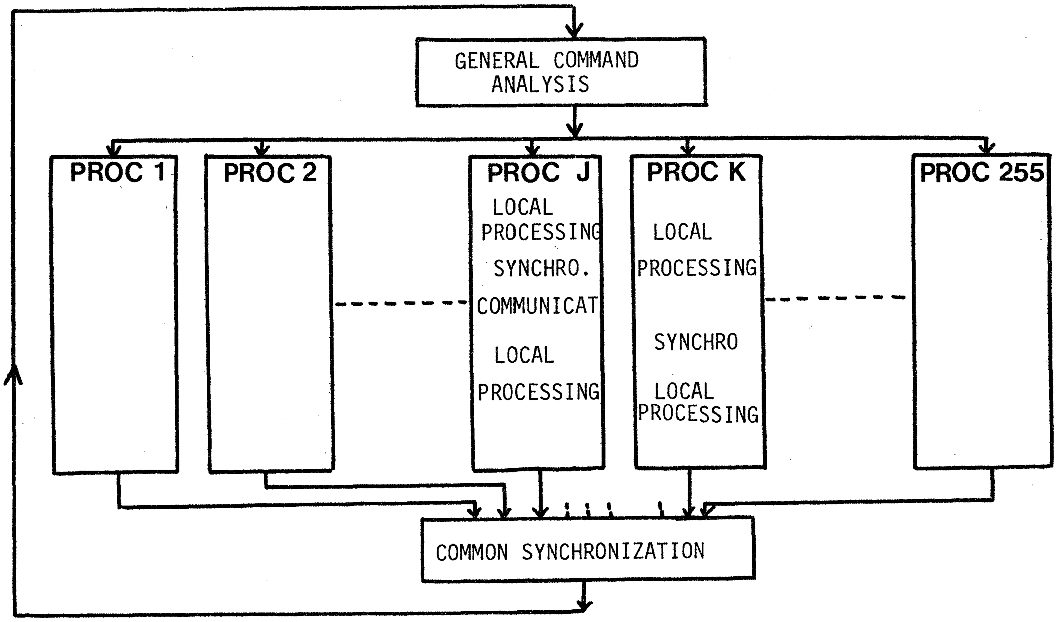


FIGURE 5 : THE CONTROL FLOW CHART FOR ONE PROCESSING ELEMENT.

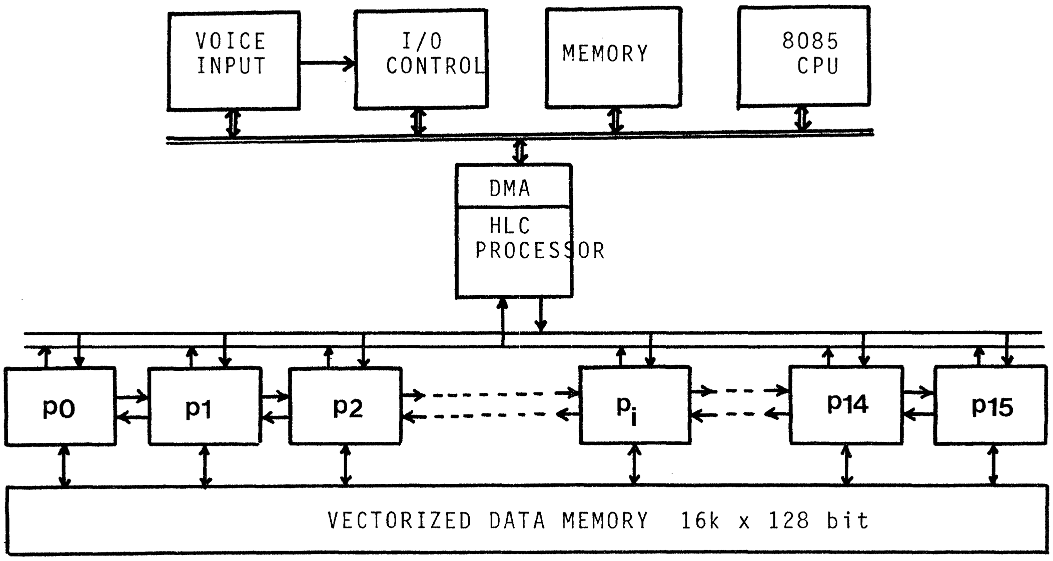


FIGURE 6 : GENERAL ARCHITECTURE OF M.A.P.

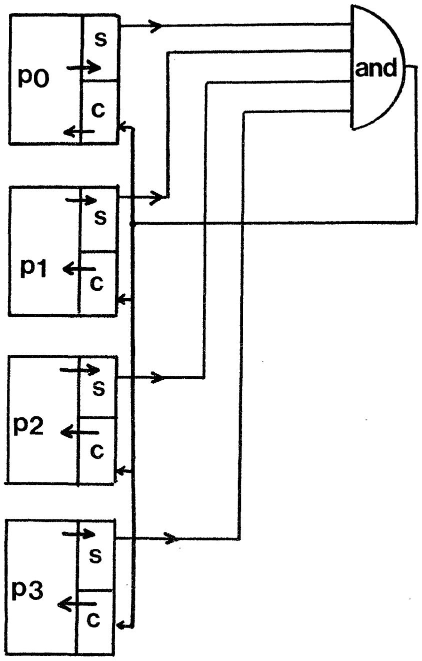


FIGURE 7A : SYNCHRONIZATION BY LOCATION. SYNC(P0,P1,P2,P3)

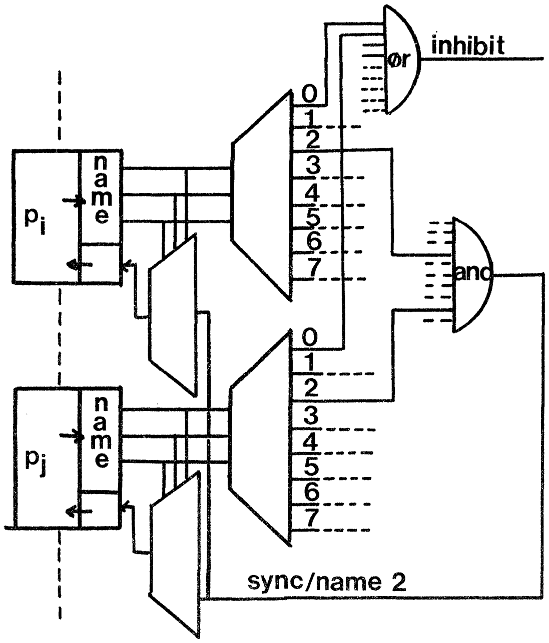


FIGURE 7B : SYNCHRONIZATION BY NAME. SYNC(NAME 2) FOR Pi AND Pj.

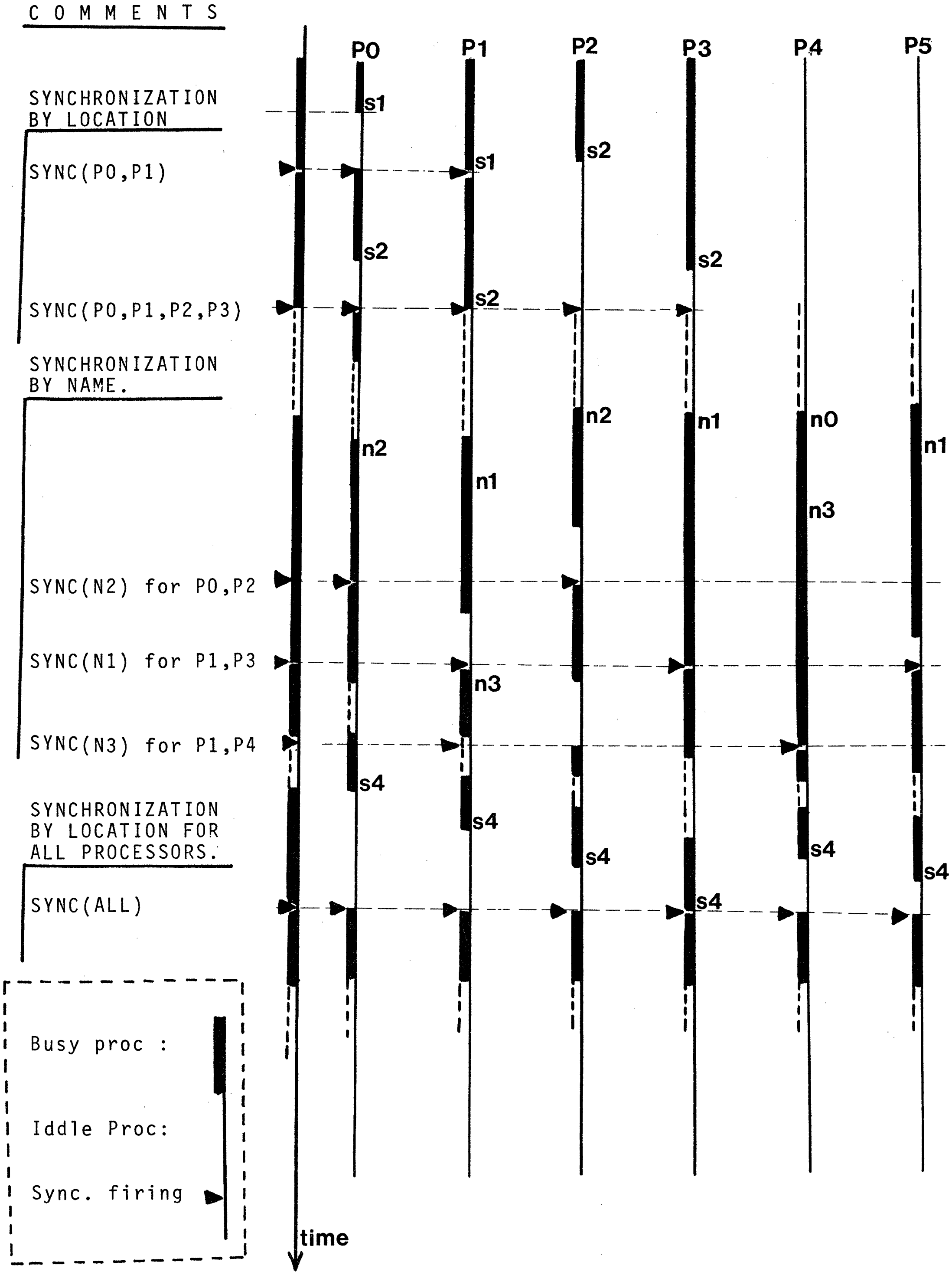


FIGURE 8: EXAMPLE OF THE SYNCHRONIZATION PROCESS.

AIRBORNE ASSOCIATIVE PROCESSOR (ASPRO)

Jon M. Surprise
Program Manager, Digital Technology Department
Goodyear Aerospace Corporation
1210 Massillon Road
Akron, Ohio 44315

Introduction

Under company sponsored Research and Development programs and subsequently under Navy Contracts 00019-78-C-0598 and 00019-79-C-0563, Goodyear Aerospace Corporation performed extensive tradeoff studies based on experience with STARAN[TM], to demonstrate the advantages of associative processing for airborne surveillance. Two advantages are: the simplicity of the software for managing the surveillance data base, and the high inherent processing speed of ASPRO. The necessary small size (0.35 ft^3) and low power (330W.) are realized using custom CMOS VLSI and multichip CMOS random access memory. The ASPRO processor, now in final development, will augment the existing data processor aboard the Navy's Grumman E-2C aircraft. Its combination of content-addressability, multi-dimensional access (MDA) memory, and parallel processing provide a powerful architecture for real-time processing applications.

Architecture

The basic architecture of ASPRO is shown in Figure 1.

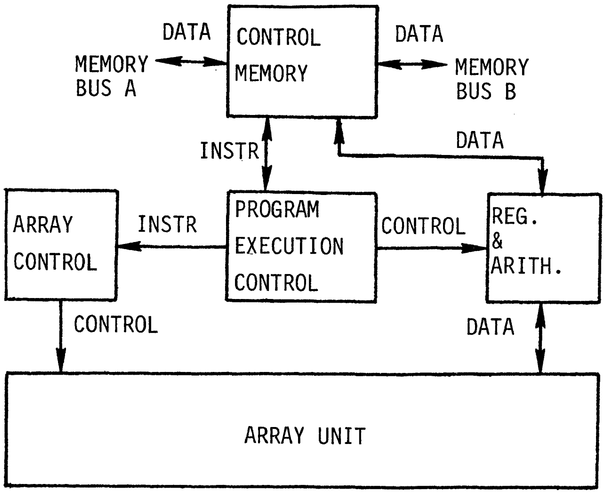


Figure 1. Block Diagram of ASPRO

ASPRO is divided into five functional subsystems:

Control Memory. This subsystem is made up of three types of storage: (1) buffer memory, (2) program memory, and (3) read-only memory (ROM).

The buffer memory provides storage for input and output data for the ASPRO. It consists of two identical modules, each capable of storing 8192 words of 32 bits each. Each buffer memory module has three access ports. Two of the access ports are connected to the two memory buses of an external computer. The third port is connected to the ASPRO's internal bus system.

The program memory, which is loaded through the buffer, provides storage for the ASPRO machine instructions. Its one access port is connected to the ASPRO bus system.

The ROM provides nonvolatile program storage for certain essential operations including program load and basic built-in test routines.

Program Execution Control. This unit controls execution of instructions stored in program memory. Four index registers are provided, as is a subroutine stack capable of accommodating 15 levels of subroutines. Conditional branches to any location in program memory can be executed. To maximize performance, fetch of the next instruction is initiated at the earliest possible stage of the current instruction such that it overlaps the current instruction execution.

Register and Arithmetic. This section contains the working registers, sequential arithmetic unit, and buses required for data transfer and control exclusive of the array. The logic in this unit consists of twenty-four 16-bit registers, two 32-bit registers, and a 16-bit arithmetic logic unit (ALU) interconnected by a bus system. The 32-bit memory bus is connected to one port of the common register and to the 32-bit instruction register.

Data to and from the array unit flows through the register and arithmetic logic. The 32-bit bidirectional array data bus is split into an array input and an array output bus by the interface logic. All data transferred to and from the array are buffered by the common register.

Sixteen general 16-bit registers and eight specific 16-bit registers are accessed via a 16-bit data bus. The general registers are loaded from the ALU output and can be used for either ALU input argument. The specific registers are dedicated to array operations to hold array addresses and loop counts.

The arithmetic logic unit (ALU) permits conventional arithmetic and logic operations to be performed upon data presented to the working registers. It can perform seven arithmetic and nine logic operations on two 16-bit operands. Multiplexers at the ALU inputs provide the capability to select various pairs of source operands.

Array Control. This unit provides the timing and control to execute the specified array operation. Basic array operations include Read Array, Write Array Masked/Unmasked from Common or Array Register, and Output to Common Register. When reading from the array, array control sets

[TM] Goodyear Aerospace Corporation

0190-3918/81/0000/0129$00.75 © 1981 IEEE

up control lines to perform one of 16 possible Boolean operations between array data and the Processing Element registers. Logical sequences of these operations permit a wide variety of associative functions to be performed on the array data.

Array Unit. The multidimensional access (MDA) array unit consists of four basic components: array memory, flip (permutation) network, processing elements, and response-store resolver. It is partitioned into 17 array modules. Sixteen modules of 128 words each make up the 2048-word array. Each word is 4096 bits in length. The seventeenth 128 word module is a spare which may be switched in if one of the basic modules fails. Each module comprises a 128-word by 4096-bit array of solid-state MDA storage and 128 processing elements (PE's).

The 2048 words of 4096 bits each provide a total of 8 megabits of data storage. Format of the 4096 bits is under total software control. Operand lengths can be 1 to 256 bits. The MDA storage organization provides access in either the bit or word direction, a technique proven in the STARAN™ associative processor from which ASPRO has evolved. The flip network and associated address logic permits MDA using conventional RAM. This, in conjunction with the response store and resolver permits parallel processing and content-addressability without sacrificing normal word-mode input-output.

The array includes 2048 PE's. Each PE contains 3 single-bit registers, and can: buffer data from or to array memory, execute all logical operations on two single-bit operands, conditionally inhibit a write instruction, and provide the response store function for search operations. Associative array input and output is 32 bits via the common data bus. The array is partitioned to provide for reduction in required volume and power by the efficient use of custom VLSI circuitry. CMOS/SOS technology has been used for the PE VLSI integrated circuit design because of its low power and high speed.

Software

A significant amount of system software is being provided to allow users to develop application programs. Software tools include: assembler, linker, loader, librarian, subroutine library, a debug package and diagnostics. Most of the system software is written in a high order language for portability, enabling program development on a variety of general purpose computers.

The assembler is a conventional two-pass assembler which supports structured modular programming. The mnemonics are separable into two sets. One set is for the sequential control portion of ASPRO and is much like the instruction set for conventional sequential computers. The second set is for the associative memory and consists of double and triple address arithmetic and logical operations.

The output of the assembler is an object module which can be combined with other object modules via the linker and librarian into a load module. The load module, when loaded into the ASPRO can be interactively debugged with the debug package. The debugger allows the user to stop the program at any program location, dump registers or memory contents, change those contents and then continue the program. In the trace mode, selected registers and memory can be dumped automatically after every instruction is executed. These and other features of the ASPRO debugger provide the user with a powerful debugging tool.

Two types of diagnostics are being developed for ASPRO: an on-line self-test program which is executed periodically to assure operational integrity and off-line diagnostics to isolate faults to a specific section of hardware.

Performance

The relative processing time for ASPRO in a radar tracking application is significantly less than a conventional processor when the number of tracks increases from several hundred to several thousand.

Table I is a simplified comparison of processing time for some typical operations on a data base of 2000 items.

Table I. Performance Comparison

OPERATION ON 2000 ITEMS	ASPRO ASSOCIATIVE PROCESSOR	CONVENTIONAL COMPUTER
SINGLE-BIT SEARCH	0.5 µSEC	1000 µSEC
16-BIT ARITHMETIC OPERATION	32 µSEC	3000 µSEC

Conclusions

The ASPRO processor is a dense, low-power, high performance processor. This parallel processing system is designed to replace or augment existing, conventional airborne data processing systems. ASPRO's simple software and high-speed search and processing capabilities provide a unique, cost-effective solution to real-time signal processing.

References

[1] K. E. Batcher, The Multidimensional-Access Memory in STARAN, IEEE Transactions on Computers, February, 1977, Vol. C-26, pp. 174-177.

[2] B. W. Prentice, Implementation of the AWACS Passive Tracking Algorithms on a Goodyear STARAN, Proceedings of the Sagamore Computer Conference, August, 1974, pp. 250-269.

[3] E. E. Eddey and W. C. Meilander, Application of an Associative Processor to Aircraft Tracking; Ibid pp. 417-430.

MODELLING OF LARGE-SCALE MARKOV CHAINS WITH ASSOCIATIVE PIPELINING

Simon Ya. Berkovich

The George Washington University
Department of Electrical Engineering
and Computer Science
Washington, D.C. 20052, USA

Summary

The scope of various applications of the associative or content-addressable processors (see, e.g.[1]) is extended to random walk modelling. The main problem in this modelling is to provide a random choice among a set of alternatives. Let us consider n alternatives with probabilities P_i. The choice of an alternative with corresponding probability can be presented as a hit by a random number R in the range (0-1) of one of the intervals $(S_0 - S_1)$, $(S_1 - S_2)$, $(S_2 - S_3)$, ..., $(S_{n-1} - S_n)$, where

$$S_0 = 0,\ S_1 = P_1,\ S_2 = P_1 + P_2,\ \ldots\ S_n = 1$$

This procedure can be organized as a search among the numbers S_0, S_1, S_2, . . . S_{n-1} for that which is the largest smaller than R. Using an associative memory of ternary elements the intervals $(S_{i-1} - S_i)$ can be presented in such a way that this search will be performed with one memory call [2]. (The third state of the ternary associative element (-M) provides matching signals for both "0" and "1" interrogations, and it can be implemented either with special hardware or with software using two bit combinations in binary associative memory.)

We will illustrate this method by an example (Fig. 1). Suppose we have four alternatives with the probabilities 2/16, 5/16, 6/16 and 3/16. The total random number range $0000 \le R \le 1111$ can be covered, for example, by the following ternary combinations:

1	-	000M	-	2/16
2	-	001M 010M 0110	-	5/16
3	-	0111 10MM 1100	-	6/16
4	-	1101 111M	-	3/16

Such a representation is a subject of minimization (cf. Fig. 1,a). The alternatives 1, 2, 3 and 4 will be accessed with the probabilities 2/16, 5/16, 6/16 and 3/16, respectively, because the chances for a random number R to match to one of these intervals is proportional to its length, i.e., 2, 5, 6 and 3.

The transition matrix of a discrete Markov chain, p_{ij}, can be stored in a format: (i, ∇, j), where i is a starting state, ∇ - a ternary combination corresponding to the choice of the j state. The interrogation of the associative memory by (i, R) will result in a random choice of "j" with the probability P_{ij}, i.e., "a transition i→j". The process of Markov chain modelling is a succession of such transitions.

The associative pipelines as suggested in[3] have actually the same algorithmic capabilities as associative processors, but the pipelines are more efficient in implementation and suitable for processing of large volumes of information. The uniform cells of the associative pipeline realize in succession the transformations isomorphic to that realized by the associative processor in parallel. The random choices are made in the pipeline cells by picking-up the numbers of the alternatives from the passing word-stream when the ternary combinations corresponding to their probabilities match the provided random numbers. All cells operate on the word-stream concurrently with the shift in time according to the propagation delay. The associative pipeline can easily perform the first-match selection for the multiple responses, so the necessary intervals for random choice can be constructed simpler using overlapping ternary combinations with partial screening of the successors. This is illustrated in Fig. 1,b. There are two possibilities for selection alternatives # 1, three possibilities for # 4, six possibilities for # 3, and five for # 2. It does not matter that the alternatives are not presented by contiguous segments; if R is uniformly distributed, the chances of the selection of the alternatives will correspond to their probabilities, i.e., # 1 - 2/16, # 2 - 5/16, # 3 - 6/16 and # 4 - 3/16. It should be emphasized that the choice of the alternative is performed in each cell independently and is determined by its own random number R only.

The basic unit of the computer system for modelling Markov chains is presented in Fig. 2.

The output of the pipeline cell is connected to its command register to extract information from the word-stream. The possibility of the control of the computing process through the word-stream is an attractive property of the associative pipelining, which can be efficiently used in different problems as, for example, considered in [4]. The word-stream is a mixture of transition matrix elements in one format and control computer messages in another format. The mode of operation is specified by tag bits, which also serve as a lock

0190-3918/81/0000/0131$00.75 © 1981 IEEE

to permit some operations on a given word and to prevent further access to this word in other cells.

The basic operations are the following:

1. Initialization - a random walking point should be set into a certain initial position.

2. Transition - moving from a given state i to one of the states j according to the probabilities (P_{ij}).

3. Random number supply - after each transition the random number R must be changed to determine the choice of the next alternative.

4. Sensing - the results of random walking should be returned to the word-stream and processed by the control computer; two types of Markov chains are usually considered: with and without absorbing states, the results of the modelling are some characteristics of the random walks to absorbing states in the first case, or of the equilibrium distribution in the second case.

The suggested technique is a typical example of the organization of the computing processes with associative pipelining. Not bound by storage limitations, it can be efficiently applied to the investigation of very large stochastic models in system analysis and computational physics.

References

1. C. C. Foster, Content Addressable Parallel Processors. Van Nostrand Reinhold Co., 1976.

2. S. Ya. Berkovich and Yu. Ya. Kochin, "Search for Numbers that are Nearest to a Given Number," Automation and Remote Control, V. 36, No. 1, pp. 343-345 (1975).

3. S. Ya. Berkovich, "An Outline of the Computer System with Associative Pipelining," Proceedings of the 1980 International Conference on Parallel Processing, pp. 47-48.

4. J. M. Pullen, An Architecture for a Database Computer Using Associative Pipelining, D. Sc. Dissertation, The George Washington University, 1981.

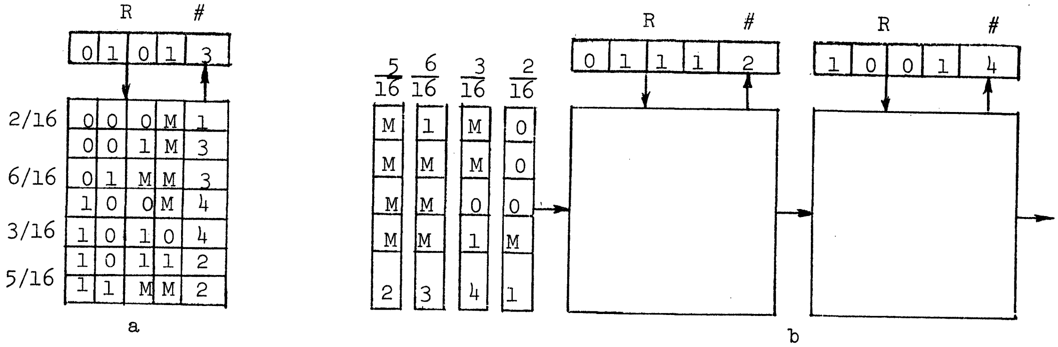


Fig. 1 Random choice with the associative processor (a) and pipeline (b).

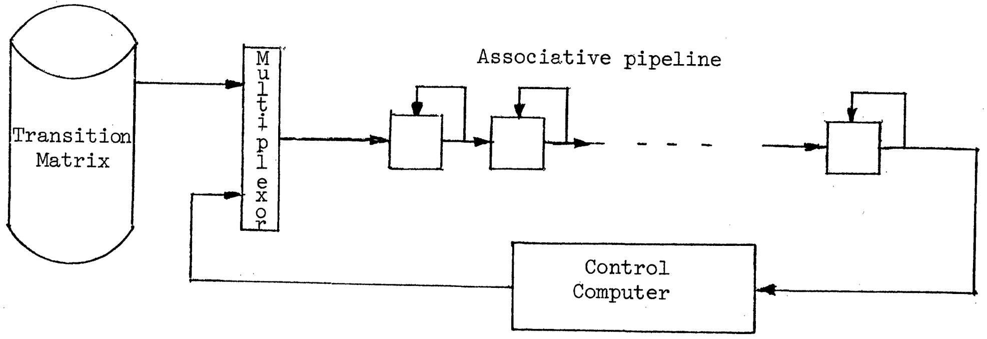


Fig. 2 Basic computing unit

RECONFIGURATION OF DYNAMIC ARCHITECTURE INTO MULTICOMPUTER NETWORKS

Svetlana P. Kartashev
University of Nebraska-Lincoln,

and

Steven I. Kartashev
Dynamic Computer Architecture, Inc.

ABSTRACT -- This paper considers reconfiguration of dynamic architectures into multicomputer networks that can assume rings, trees, and stars configurations.

Reconfiguration algorithms introduced are one-step algorithms performed concurrently by all network nodes requested for reconfiguration. The time of this step is the time to execute a one-bit shift and mod 2 addition. These reconfigurations can be accomplished with special shift-registers called shift-registers with variable bias (SRVB) introduced into each network node, N, that store the position code of this node. Upon receipt of the reconfiguration instruction, each such register generates the position code of the network node, N, with which node N must establish a data path consistent with the overall network configuration (tree, star, or ring).*

1. INTRODUCTION

As was shown in the literature [1-3], a dynamic architecture may increase a system throughput using the following adaptations to algorithms:

1) Adaptation of the resources to instruction and data parallelism, and

2) Reconfiguration of the resources into multicomputer, multiprocessor, array, and pipeline architectures.

A multicomputer adaptation to algorithms is generally understood as:

a) The architectural capability to partition resources into a variable number of dynamic computers with changeable word sizes, and

b) The capability of multicomputer architecture to function as a multicomputer network characterized by different topological configurations among its computers.

Since various techniques that implement the a) property of dynamic architectures were studied in [2-5], this paper concerns itself with reconfiguration of dynamic architecture into a multicomputer network.

2. APPLICATION OF MULTICOMPUTER NETWORKS

A multicomputer network is characterized by different *network structures* formed by network computers, otherwise called *network nodes*. As was shown in the literature [6-12], the most convenient network structures are rings, trees, stars, binary cubes, closely connected graphs, and mixed structures that involve various combinations of the above mentioned structures.

Rings, cubes, and strongly connected graphs are useful for computational algorithms in which each computer node performs computations only and assigns no computations to other nodes.

Trees and stars are useful for both computational and control algorithms. Trees and stars are described by two types of nodes--*leaves* and *non-leaves*--where a leaf is generally understood as a node of the lowest level, i = 0, and a non-leaf node has level i > 0. A node(s) of the highest level, i = L, is called a *root*.

A tree or a star may have one or several roots describing a corporate structure with one or several directors, respectively. The difference between a star and a tree is in the number of nodes of a lower level that are adjacent with a node of level i. In a tree, this number does not exceed one for a root, and two for each node that is neither root nor leaf. For a star, each non-leaf node of level i may have more than two adjacent nodes of a lower level.

If a tree or star has one root it is called a *one-root tree* or a *one-root star*; if it has several roots, it is called a *multiple root tree* or a *multiple root star*.

3. REQUIREMENTS FOR A MULTICOMPUTER NETWORK

In order to provide a multicomputer network with very high flexibility and reduce the amount of data to be transferred among its nodes, any network must be provided with the following characteristics.

C1. *Minimal Reconfiguration Time*. This is understood as the minimal time required by the network to reconfigure itself into any of the network structures indicated above.

C2. *Multifunctional Node*. This is understood as the capability of each node, N, to be connected into any network structure (ring, strongly connected graph, cube, tree, or star). Within a tree or a star a multifunctional node should be capable of functioning as a root (single or multiple), leaf, or a non-leaf node. As a result, a programmer will be able to minimize idle resources not involved in a particular computation and to eliminate traffic bottlenecks created in particular portions of a network due to overcentralization of information flow from the root(s) to other nodes.

C3. *Variable Word Sizes of a Network Node*. To increase the network flexibility each network node must be provided with the capability to change its word size. This will minimize the amount of resource interconnected into a particular network configuration, and allows computation of additional programs using the same resources. The advantages of such computations are coincident with those performed by dynamic architec-

0190-3918/81/0000/0133$00.75 © 1981 IEEE

tures in general. These are treated extensively in [2-5].

A multicomputer network that is provided with properties C1, C2, and C3, and performs reconfigurations into the network structures described above, can be organized using the DC group described in [1-3].

4. CONTRIBUTION TO THE STATE-OF-THE-ART

This paper studies reconfiguration of dynamic architecture into rings, trees, and stars. Reconfiguration algorithms developed are based on the theory of shift-register sequences as follows.

A network node N activates a data path with its immediate successor, N^*, in the given network structure when N generates the position code of N^* [2, 3]. This activation may be done with the use of shift-register and special constant B brought with the reconfiguration instruction to all network nodes that are requested for reconfiguration.

4.1. Rule of Succession during Reconfiguration

To activate each data exchange that can be either PE-ME*, PE-PE*, ME-PE*, or ME-ME*, it is sufficient for the processor element, PE, belonging to computer element, CE, identified with network node N to generate the position code of the CE* identified with the network node N^* that contains a second element of the exchanging pair (ME* or PE*). This will be denoted as transition $N \rightarrow N^*$ meaning that: a) N will generate position code of N^*, b) N will establish a given data path between N and N^*, and c) the data path activated by N can be made bidirectional--either from N to N^* or from N^* to N--and it can be one of the four types considered above (PE-ME*, PE-PE*, ME-ME*, or ME-PE*).

(Here for simplicity it is assumed that each node N is equivalent to one CE. An extension of the results accomplished to a dynamic computer, $C(k)$, assembled of k CE can be done very easily by assigning the same position code to all its CE's.)

To minimize the time of reconfiguration, it is reasonable to assume that for each network structure, such rule of succession, $N \rightarrow N^*$ should be maintained during reconfiguration for which each node N has a minimal number of successors N^* in this structure. Then it will take the minimal reconfiguration time to establish all the data paths between N and each of its successors, N^*.

While for rings, the rule of minimal number of successors is trivial, for trees and stars it requires that the succession be maintained in the direction from leaves to root(s). Namely for each $N \rightarrow N^*$, the level of N is lower than N^*.

If all these paths are established concurrently, the entire network reconfiguration takes time T of activating only one network transition, $N \rightarrow N^*$, and can be performed with a one-step reconfiguration algorithm performed concurrently by all network nodes.

4.2. Application of Shift-Register Theory

In this paper, trees, stars, and rings will be generated with the use of shift-register theory. Its application proceeds along the following lines.

Assume that each network node N is provided with a special shift-register of length n which stores its position code N, where n is the size of the code (Fig. 1). Suppose that in the given network structure to be assumed, node N should be succeeded by node N^* via PE-PE*, PE-ME*, ME-ME*, or ME-PE* data path. Then for each type of communication between N and N^*, node N generates position code N^* using a left-shifted shift-register that generates N^* as follows:

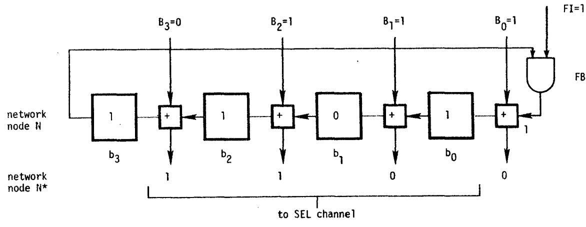


FIGURE 1

Shift-Register with Variable Bias B = 0111

$$N^* = 1[N] \oplus B \qquad (1)$$

where $1[N]$ is one-bit shift of N to the left and B is an n-bit reconfiguration constant brought with the reconfiguration instruction to all network nodes that are requested for reconfiguration. Reconfiguration constant B will be called *bias* and the shift-register of Fig. 1 is called a *shift-register with variable bias* (SRVB).

In Fig. 1, $N = 1101 = 13$, $B = 0111 = 7$. This gives $N^* = 1[1101] \oplus 0111 = 1011 \oplus 0111 = 1100$. Therefore network node N_{13} generates position code $N^* = 1100 = 12$ of its successor in the given network structure. This code will then activate a given data path between nodes N_{13} and N^*_{12}.

The gate FBG in Fig. 1 is called a *feedback gate*. Introduction of the FBG gate allows a shift-register, SRVB, to perform two types of shifts: a) *circular* $1[N]_1$, when *feedback input* FI = 1; and b) *non-circular* $1[N]_0$, when FI = 0.

As will be shown later, if FI = 1, concurrent shift-registers of network nodes generate rings; if FI = 0, they generate trees, where the meaning of FI is brought to each node with the reconfiguration instruction.

However, different network structures depend not only on the value of bias B, and feedback input FI, but also on the type of the SRVB activated in each node.

To this end SRVB can be *single* and *composite*. A *single SRVB* has a unique feedback gate FBG, which connects its MSB with LSB. A *composite SRVB* is formed from k ($k > 1$) single shift-registers each having a unique feedback gate, FBG_i.

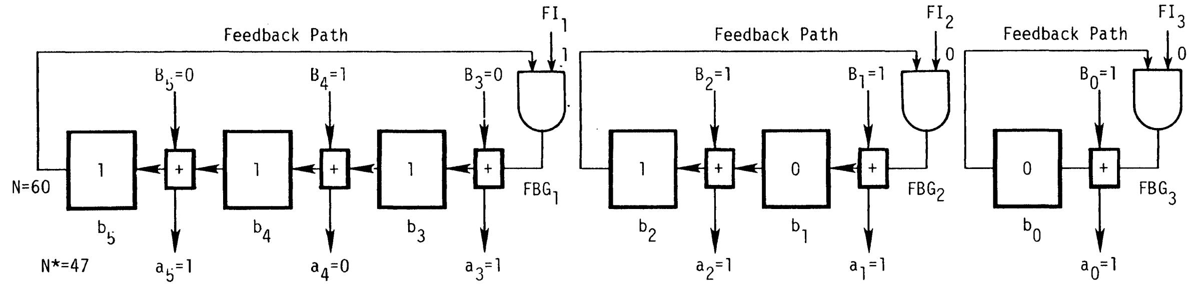


FIGURE 2

Composite SRVB

For instance, Fig. 2 shows a composite shift-register with three feedback gates, FBG_1, FBG_2, FBG_3.

Generally, in a shift-register with variable bias, each bit can broadcast its value via one of two alternative paths: a) a unique *shift-path* when it is shifted left to the next more significant bit, and b) a unique *feedback path*; when it is sent right to some less significant bit.

Activation of either a shift or a feedback path for each bit can be made by a special reconfiguration code RC stored in the reconfiguration instruction that performs reconfiguration into a given network structure. This instruction also brings to each node the same bias B that forms position code of the CE* identified with node N* that succeeds node N in a given network structure. The same bias B received by PE of node N is conceived of as an address of the instruction stored in local ME that initiates a subroutine of communication between node N and N*.

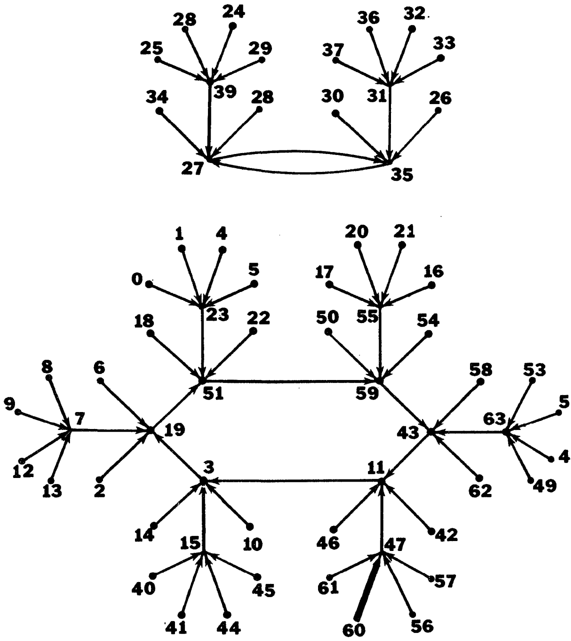


FIGURE 3

Network Structure Generated by Composite SRVB of Fig. 2

For instance, if the reconfiguration instruction stores bias B = 010111 and reconfigures the shift-register, SRVB, of each network node N into a composite one shown in Fig. 2, then the network structure formed is shown in Fig. 3. As seen, it consists of a 6-root star and a 2-root star.

For instance, composite shift register partitions N = 60 = 111100, into $b_5b_4b_3$ = 111; b_2b_1 = 10; b_0 = 0. Bias B = 010111 is also partitioned into $B_5B_4B_3$ = 010; B_2B_1 = 11; and B_0 = 1. Therefore the composite shift register generates the following successor N* of node N_{60}: $a_5a_4a_3 = 1[111]_1 \oplus 010 = 111 \oplus 010 = 101$; $a_2a_1 = 1[10]_0 \oplus 11 = 00 \oplus 11 = 11$; $a_0 = 1[0]_0 \oplus 1 = 0 \oplus 1 = 1$, giving N* = 101111 = 47.

Similarly, one can obtain any other single successor N* of the given node N. As follows, reconfiguration into the structure of Fig. 3 is performed during the time of one 1-bit shift and mod 2 addition executed concurrently by all the network nodes that receive the same bias B = 010111 and the same reconfiguration code RC that reconfigures each SRVB into the composite register shown in Fig. 2.

4.3. Contribution to the Ongoing Research

The contribution of this paper to current state-of-the-art on network reconfiguration is two-fold: 1) It devises original, simple, and elegant techniques on network reconfiguration into the structures that proved to be convenient for a large class of computational and control algorithms. The time for such reconfigurations approaches the theoretically minimal boundary. 2) It further expands a shift-register theory described in [13-17] as follows.

In the literature the shift-register studied is shown in Fig. 4. Here each circle marked with B_i means connection if $B_i = 1$ and disconnection if $B_i = 0$. Thus $B = (B_{n-1}, \ldots, B_0)$ is conceived of as the same bias as was introduced above for the shift-register SRVB. The difference between these two registers is: Fig. 4 shows a *linear*

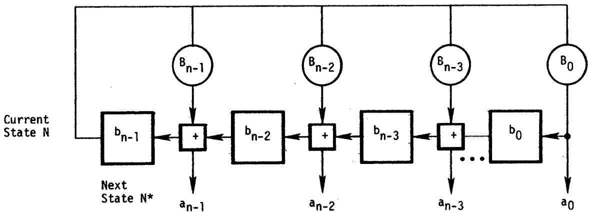


FIGURE 4

Linear Shift-Register

shift-register which broadcasts to each mod 2 adder the meaning of its MSB provided $B_i = 1$. In the linear shift-register each next state N* generated can be obtained via matrix multiplication $N^* = N \cdot A$ where N is a current state stored in bits $b_{n-1}, b_{n-2}, \ldots, b_0$, and A is the *canonical shift-register matrix* given below:

$$A = \left|\begin{array}{ccccccc} B_{n-1} & B_{n-2} & B_{n-3} & \cdots & B_2 & B_1 & B_0 \\ 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 & 0 \\ & \cdot & & \cdot & & \cdot & \\ & \cdot & & \cdot & & \cdot & \\ & \cdot & & \cdot & & \cdot & \\ 0 & 0 & 0 & \cdots & 1 & 0 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 1 & 0 \end{array}\right|$$

For instance, if bias B = 1011 and the current state N = 1100, then the next state N* to be generated by a linear shift-register is $N^* = 1100 \cdot \left|\begin{array}{c} 1011 \\ 1000 \\ 0100 \\ 0010 \end{array}\right| = 0011$. Find the next state N* generated by the SRVB storing the same N and B (assume that gate FBG is set since for linear shift-register its MSB is also fed back to the LSB): $N^* = 1[1100]_1 \oplus 1011 = 1001 \oplus 1011 = 0010$. As follows linear shift-register and SRVB generate different next states for the same current state N and bias B inasmuch as in SRVB each mod 2 adder receives bit B_i rather than MSB of the register if $B_i = 1$ as is the case for the linear shift-register. As a result, different structures of trees, stars, and rings are generated by these two types of shift-registers. In particular, a fundamental property of a linear shift register is that it always generates next state N* = 0000 if a current state N = 0000, i.e., 0 always generates a cycle of period 1 since $N^* = 0 \cdot A = 0$. On the other hand, SRVB maps 0 onto bias B, i.e., if N = 0, it is succeeded by N* = B. This means that if bias $B \neq 0$, then 0 is a node of a network structure other than cycle of period 1.

For the linear shift-register this is fundamentally impossible. For instance, for the single circular SRVB receiving B = 0111, 0 belongs to the following ring of period 8: {0,7,9,4,15,8,6,11}. If this shift-register stores N = 0101, it generates another ring of period 8: {5,13,12,14,10,2,3,1}. Indeed, if N = 5, it is succeeded by $N^* = 1[0101] \oplus 0111 = 1010 \oplus 0111 = 1101 = 13$, etc. This network structure cannot be obtained with 4-bit linear shift registers no matter what bias B is selected, since linear shift-registers always map 0 onto 0. Thus, the remaining 15 nodes cannot be formed into 2 rings of period 8 each since this will require 16 nodes.

Hence, the network structures generated by SRVB and linear shift-registers are not equivalent. Furthermore, a fundamental drawback of a linear shift-register is that the techniques for finding the network structures that can be generated are very laborious and complex, since they are based on finding the periods of polynomials over Galois field [13-16]. The complexity of these techniques grow exponentially with an increase in n, the number of bits in a shift-register. However, for complex multicomputer networks having a large number of nodes the size n of a code that identifies each node may become significant (n = 10 and more). Thus it becomes prohibitively difficult to utilize elegant results of linear shift-register theory in order to tabulate different cycles and trees that may be generated in an n-dimensional binary space with the use of linear shift registers. As for stars, linear shift-registers can generate no stars by definition.

On the other hand, all the network structures generated by SRVB (single and composite) can be described with very simple formulas that can be used by the programmer performing various reconfigurations in the multicomputer networks.

As will be shown in this paper, complexity of the techniques remain constant and does not depend on n, the size of the position code N. Thus, these techniques are applicable to complex multicomputer networks, inasmuch as they allow obtaining simple and fast reconfiguration algorithms and simple descriptions of various network structures that can be generated in the network.

The only area of equivalence among linear shift-register and shift-register with variable bias is when bias B = 0. If B = 0, both registers generate either the same binary tree with the root R = 0, or both are transformed into a *circulating shift-register* whose structure has been extensively studied in the literature [16].

5. NETWORK RECONFIGURATION

If an application program needs a new network structure for execution of its tasks, it contains global or local modification of the reconfiguration instruction, RIN, where a global modification establishes the same type of data exchange for all network transitions, N → N*, whereas a local modification of RIN allows different data exchanges for various network transitions. RIN can be executed in an array or even in a single CE. It stores the following codes: 1) Code RR of requested resource which determines whether or not a requested resource is ready for reconfiguration. 2) Reconfiguration code, RC, that reconfigures the shift-register, SRVB, of each requested network node N, into the type that generates the required network structure. 3) The bias B, which allows each shift-register, SRVB, reconfigured by the RC code to generate position code N* that succeeds N in the given network structure. 4) Program user code, NP, that is used in the priority analysis, aimed at determining the priority of the program to perform network reconfiguration. 5) For global modification of the RIN instruction it stores the code of exchange, COE, provided all requested nodes will maintain the same type of exchange (PE-ME*, PE-PE*, ME-ME*, or ME-PE*). Each CE that receives reconfiguration code RC, bias B, and code of exchange, COE, performs the following steps.

Step 1. It sends RC to its shift-register, SRVB, to reconfigure it into the type (single or composite), that generates the required network structure. The bias B is sent to this SRVB to generate the position code of the network node N* that succeeds N in this network structure.

Step 2. The bias B is used as the base address of the task that begins execution in the network structure. As a rule, bias B stores a jump instruction which performs jump to another location of the local memory, ME.

Step 3. For the global modification of the RIN instruction, the code of exchange, COE activates a needed data path between network nodes N and N*, where N* was formed during Step 1.

Such an organization of RIN allows very fast reconfigurations into the network structures which proved to be very efficient for computation. The time of these reconfigurations approaches the absolute minimum due to the following reasons: a) Concurrent 1-step reconfiguration algorithms in which the entire network reconfiguration is made in one step by all network nodes during the time of 1-bit shift and mod 2 addition. b) Minimal time required to establish each data path between two network nodes, N and N*.

6. TYPES OF SHIFT-REGISTERS

This section will introduce the techniques for describing various types of shift-registers, SRVB.

6.1. Arithmetic Formats

Each composite SRVB will be described with an *arithmetic format*, AF = $[k_1, k_2, \ldots, k_p]$, where k_i is the size of each single shift-register contained in SRVB. Obviously $n = k_1 + k_2 + \ldots + k_p$ where n is the size of the SRVB.

Since each single shift-register of the arithmetic format AF may perform either circular shift provided that the feedback input FI = 1 or non-circular shift provided FI = 0, the arithmetic format AF may be divided into the following categories: a) *Circular* AF_1, when all its single shift-registers perform circular shifts; b) *Non-circular* AF_0, when all its single shift-registers perform non-circular shifts; c) *Mixed* AF_{10}, when single shift-registers described by it perform circular and non-circular shifts.

It will be convenient to represent mixed AF_{10} as a combination of circular and non-circular AF, i.e., $AF_{10} = AF_1 \times AF_0$, where AF_1 includes all circular single shift-registers and AF_0 includes all non-circular ones.

For instance if $A_{10} = [3_0, 4_1, 5_1, 2_0]$, then $A_1 = [4, 5]$ and $A_0 = [3, 2]$, i.e., $A_{10} = A_1 \times A_0$.

6.2. Reconfiguration Code

Reconfiguration of the SRVB into any given arithmetic format will be performed with the reconfiguration code, RC. RC is stored in the reconfiguration instruction and described as follows:

It is (2n-1)-bit code where n is the size of each SRVB. It consists of (n-1) 2-bit zones, Z_i, each including two bits, S_i and F_i, and one 1-bit zones, Z_0 including only one bit, F_0. Thus RC = $(Z_{n-1}, Z_{n-2}, \ldots, Z_1, Z_0)$.

Each zone Z_i encodes, respectively, feedback and shifting paths for the two bits b_i and b_{i-1} of the SRVB, where b_i is more significant than b_{i-1} (Fig. 5).

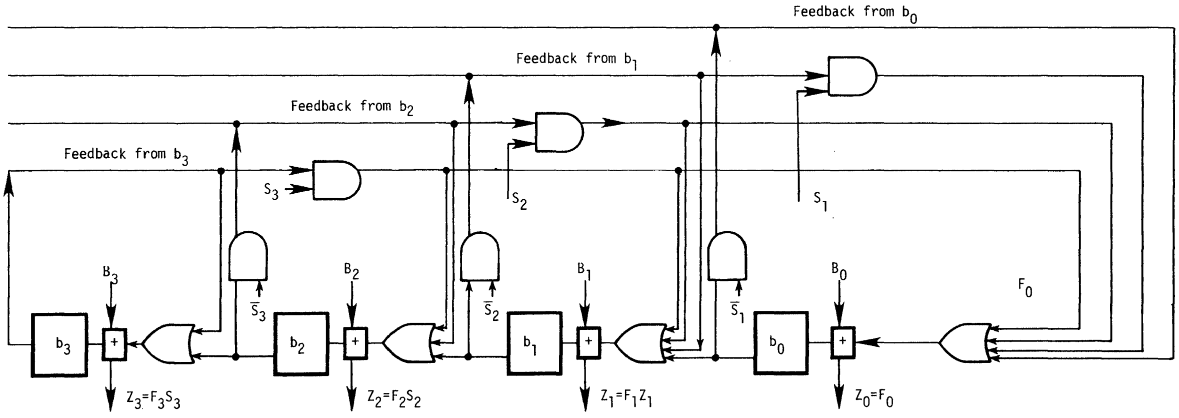


FIGURE 5

Shift and Feedback Paths in 4-bit SRVB

For each zone, $Z_i = (F_iS_i)$, the values of F_i and S_i show what type of path is activated for every pair of consecutive bits, b_i and b_{i-1}. If $F_i = 1$, bit b_i receives circular feedback information; and it receives no shift information from the next less significant bit, b_{i-1}. If $F_i = 0$, bit b_i receives either no feedback, or it is non-circular feedback (for trees and stars).

Bit $S_i = 1$ of zone Z_i stands for left shift from b_{i-1} to b_i and $S_i = 0$ stands for no shift from b_{i-1}. Therefore, together F_i, S_i show what type of path is activated between b_i and b_{i-1}; shift path ($S_i = 1$, $F_i = 0$) or feedback path to b_i and no shift from b_{i-1}, $S_i = 0$, $F_i = 0 \vee 1$.

Since $S_i = 1$ means that bits b_i and b_{i-1} belong to the same shift register and $S_i = 0$ means that they belong to two different shift registers, each $\bar{S}_i$ is sent to activate a new feedback path initiated in b_{i-1}. Likewise, each S_i is sent to the feedback path initiated in b_i either to maintain it if $S_i = 1$ or block unwanted transfer of b_i to less significant bits of shift register if $S_i = 0$.

Example. In Fig. 6, if $S_3 = 1$, bits b_3 and b_2 belong to the same shift register and $S_3 = 1$ maintains the feedback path initiated in b_3. At the same time $\bar{S}_3 = 0$ blocks b_2 from initiating its own feedback path.

If $S_3 = 0$, b_3 and b_2 belong to two shift registers. Thus $\bar{S}_3 = 1$ initiates a new feedback path from b_2 and $S_3 = 0$ blocks unwanted transfers of b_3 to other less significant bits, etc. As follows, selection of the RC code can be formalized and described with a very simple algorithm that is not introduced in this paper.

7. SINGLE NETWORK STRUCTURES

The objective of this section is to outline the ways for solving the following problem:

Given bias B and an arithmetic format AF = $[k_1, k_2, \ldots, k_p]$. Find the network structure that is generated.

The solution of this problem will allow a programmer to select bias B and reconfiguration code RC and obtain all the network structures that are needed.

Before attacking a general case of arbitrary AF consider the so-called *single network structures* produced by single shift-registers, i.e., those identified by AF = [n].

These can be of two types: rings and trees, specified by circular and non-circular arithmetic formats, respectively. Rings will be described first.

7.1. Single Ring Structures

A *single ring structure*, SRS, is a set of rings that is generated by single shift-registers available in network nodes. To define SRS means to define the following:

a) A set of periods, SP = {T}, where T is the period of a ring generated in the SRS, and

b) The number D(T) of rings having the same period, T.

Therefore, we define SRS as: SRS = $\{D(T) : T \varepsilon SP\}$.

7.1.1. *Set of Periods for Single Ring Structure*

The set of periods, SP, is completely specified by the bias B: namely, how many ones are in B--odd or even. Before introducing this result we will make some definitions.

By the *weight*, W, of the bias B we mean the number of ones it has. We say that bias B is *even* if its weight is an even number and B is *odd* if its weight is an odd number.

Let SD_n be a set of divisors for number n and $\overline{SD}_{2n} = SD_{2n} - SD_n$ where (-) is understood as a set subtraction.

For instance, for n = 6, $SD_6 = \{6,3,2,1\}$ and $\overline{SD}_6 = SD_6 - SD_3 = \{6,3,2,1\} - \{3,1\} = \{6,2\}$. As follows $\overline{SD}_p$ may be specified only if p is even. With this in mind let us introduce one theorem that specifies the set of ring periods, SP, for single ring structures.

Given: Circular single arithmetic format $AF_1 = [n]$ and Bias B fed to each shift-register.

Theorem 1. If bias B is even, then SP = SD_n; if bias B is odd, then SP = $\overline{SD}_{2n} = SD_{2n} - SD_n$. (2)

Example. For the SRVB in Fig. 6, specified with arithmetic format $AF_1 = [3^1]$, bias B = 001 is odd, since its weight W = 1 is an odd number. Thus a set of periods SP = $\overline{SD}_{2 \cdot 3} = \overline{SD}_6 = SD_6 - SD_3 = \{6,2\}$. If the same shift-register is fed with an even bias (B = 000, 011, 110, 101) then the set of ring periods, SP = $SD_3 = \{3,1\}$ (Fig. 7).

7.1.2. *Number of Rings with the Same Period*

As was seen, the set of ring periods, SP, can be found very easily. It is either SD_n or $\overline{SD}_{2n}$ for $AF_1 = [n]$. Similar simplicity is associated with the formulas that find D(T), the number of rings having period T, where T is a member of SP.

Theorem 2. In a single circular shift-register, $AF_1 = [n]$, fed with an even bias B, for any period, $T \varepsilon SP$,

$$2^T = \sum_{T' \varepsilon SD_T} T' \cdot D(T') \qquad (3)$$

As follows from (3), this formula is recursive since for any ring period, T, one can find the number, D(T), of rings with period T, only after finding D(T') for all divisors T' of T.

Example. Given $AF_1 = [6]$ fed with an even bias, B.

Using Theorem 1, one obtains that the set of its ring periods is SP = $SD_6 = \{6,3,2,1\}$. Using Theorem 2, apply the recursive procedure for finding D(T) for any $T \varepsilon SP$. Start with T = 1, 2^1 =

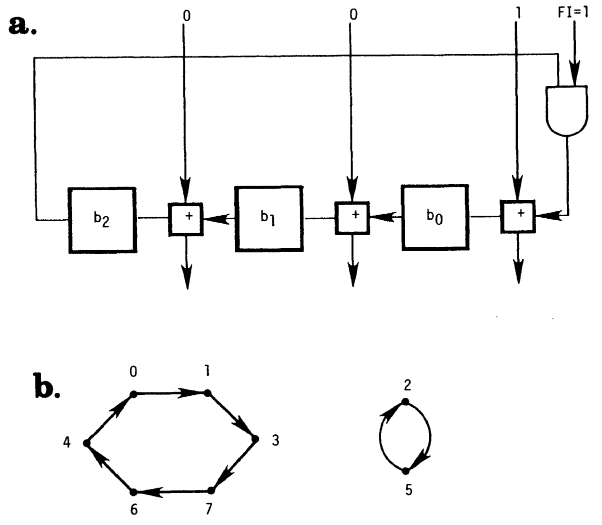


FIGURE 6

a. Single SRVB with $AF_1 = [3^1]$ and B = 001

b. Single Ring Structure Generated by This SRVB

1·D(1); D(1) = 2; for T = 2, $2^2 = 1\cdot D(1) + 2\cdot D(2)$ and D(1) is a known value. $2^2 = 2 + 2\cdot D(2)$; D(2) = (4-2)/2 = 1; for T = 3, $2^3 = 1\cdot D(1) + 3\cdot D(3)$ and D(3) = (8-2)/3 = 2; for T = 6, $2^6 = 1\cdot D(1) + 2\cdot D(2) + 3D(3) + 6\cdot D(6)$ and $D(6) = (2^6-1\cdot D(1)-2\cdot D(2)-3\cdot D(3))/6 = (2^6-2-2\cdot 1-3\cdot 2)/6 = (64-2-2-6)/6 = 9$. Thus we found that the single ring structure, SRS, generated by this shift-register is: SRS = {2(1),1(2),2(3),9(6)}.

Similar simplicity is associated with finding the numbers of rings with period T generated by shift-registers receiving an odd bias B. Since the set of ring periods, SP = $\overline{SD}_{2n} = SD_{2n} - SD_n$, where n is the size of shift-register, then we may establish the following Theorem 3.

Theorem 3. In a single circular shift-register, AF_1 = [n], fed with an odd bias B, for any ring period T ε $\overline{SD}_{2n}$

$$2^{T/2} = \sum_{T' \,\varepsilon\, \overline{SD}_T} T'\cdot D(T') \qquad (4)$$

This is also a recursive formula since one can find D(T) only after finding D(T') for all periods, T' ε $\overline{SD}_T$. The recursive process starts with D(T) where $T = 2^s$ and s ⩾ 1, because for $T = 2^s$, $\overline{SD}_{2^s} = SD_{2^s} - SD_{2^{s-1}}$ contains only one member 2^s, i.e., $\overline{SD}_{2^s} = \{2^s\}$.

Example. Given shift-register with circular arithmetic format AF_1 = {6}, fed with an odd bias B. Using Theorem 1, one obtains that the set of its ring periods SP = $\overline{SD}_{12} = SD_{12} - SD_6$ = {12,6,4,3,2,1} - {6,3,2,1} = {12,4}. Using Theorem 3, one first finds D(T) for T = 4 as $4\cdot D(4) = 2^2$ giving D(4) = 1; next $12\cdot D(12) + 4\cdot D(4) = 2^6 = 64$ and $D(12) = (2^6-4\cdot D(4))/12 = (64-4)/12 = 5$. Thus, single ring structure, SRS, generated by this shift-register is: SRS = {1(4), 5(12)}

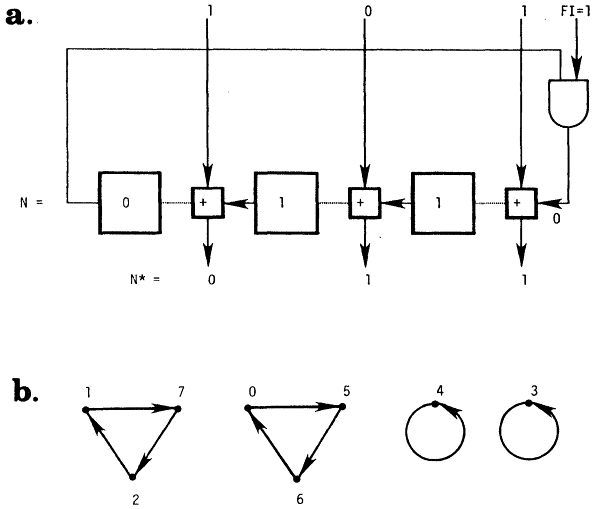


FIGURE 7

Single Ring Structure Generated by the SRVB with AF = [3] Receiving B = 101

7.2. Single Tree Structure

As was indicated above a single shift-register with non-circular arithmetic format generates a single-rooted binary tree (Fig. 8). We will call a single-rooted binary tree generated by a single non-circular SRVB with arithmetic format AF_0 = [n] a *single tree structure*, STS.

As was shown above, to minimize the time of reconfiguration, an adopted direction of succession is from the leaves to the root, R, which then succeeds itself by forming a cycle of period 1.

For tabulation purposes, we will use the following symbols for different single tree structures: if STS is generated by non-circular shift-register with arithmetic format AF_0 = [n], then STS = $\overrightarrow{[n,1]}$, shows that the tree is single and has n levels, and the root R succeeds itself by forming a cycle of length 1, i.e., $\vec{1}$.

For instance, the STS of Fig. 8 is described as STS = $\overrightarrow{\{3, 1\}}$, since this tree is single, it has three levels, and its root, R = 7, forms a cycle of length 1, i.e., $\vec{1}$, because $1[111]_0 \oplus 001 = 110 \oplus 001 = 111$.

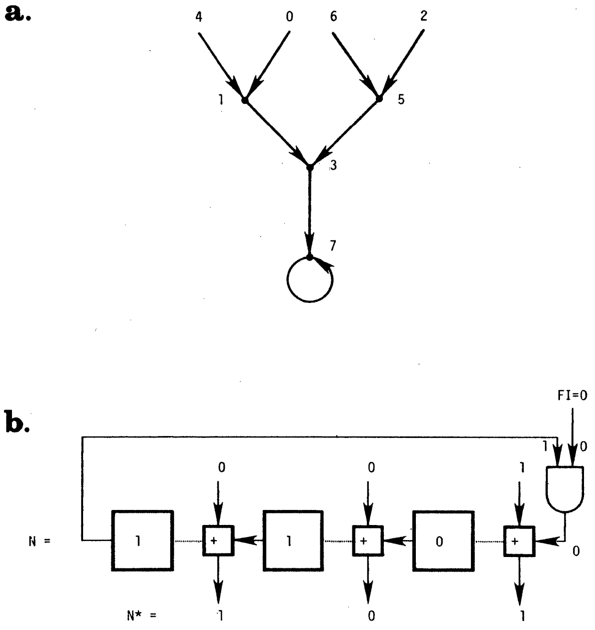


FIGURE 8

STS Generated by the $[3]_0$ *Shift-Register*

7.2.1. *Technique for Finding Root for Single Tree Structures, STS*

Since in a one-rooted tree root, R, may store important information that needs to be transferred to other nodes (such as deactivation of some tree nodes from computation, or other managerial information), it is desirable to provide a programmer with simple techniques for finding root analytically. The problem that is to be solved is: Given non-circular arithmetic format AF = $[n]_0$ and bias B. Find root R. This problem is solved in the following Theorem 4.

Theorem 4. In a shift-register SRVB specified with non-circular arithmetic format AF = $[n]_0$, let $CL(b_i) = 2^i \oplus 2^{i+1} \oplus 2^{i+2} \oplus \ldots \oplus 2^{n-1}$. (For instance for $AF_0 = [4]$, $CL(b_0) = 2^0 \oplus 2^1 \oplus 2^2 \oplus 2^3$; $CL(b_1) = 2^1 \oplus 2^2 \oplus 2^3$; $CL(b_2) = 2^2 \oplus 2^3$; and $CL(b_3) = 2^3$.) Let Bias B be $B = b_{i_1} \oplus b_{i_2} \oplus \ldots \oplus b_{i_p}$. Then the root R is: $R = CL(b_{i_1}) \oplus CL(b_{i_2}) \oplus \ldots \oplus CL(b_{i_p})$.

Example. For the arithmetic format AF_0 = [4], let bias $B = 1 \oplus 4 \oplus 8 = 1101$. Find $CL(1) = 1 \oplus 2 \oplus 4 \oplus 8$, $CL(4) = 4 \oplus 8$ and $CL(8) = 8$. Then root R is: $R = CL(1) \oplus CL(4) \oplus CL(8) = 1 \oplus 2 \oplus \cancel{4} \oplus \cancel{8} \oplus \cancel{4} \oplus \cancel{8} \oplus 8 = 1 \oplus 2 \oplus 8 = 1011$. Indeed, R is succeeded by the following $N^* = 1[R]_0 \oplus B = 1[1011]_0 \oplus 1101 = 0110 \oplus 1101 = 1011$, i.e., $N^* = R$ and it forms cycle of length 1.

Therefore using Theorem 4, a programmer may find a root before hand and assign it with tasks that perform many useful functions in the network.

References

[1] C. R. Vick, S. P. Kartashev, and S. I. Kartashev, "Adaptable Architectures for Supersystems," *Computer*, November, 1980, pp. 17-35.

[2] S. I. Kartashev and S. P. Kartashev, "Problems of Designing Supersystems with Dynamic Architectures," *IEEE Transactions on Computers*, vol. C-29, December, 1980, pp. 1114-1132.

[3] S. P. Kartashev and S. I. Kartashev, "Performance of Reconfigurable Busses for Dynamic Architectures," *Proceedings of the First International Conference on Distributed Computing Systems*, Huntsville, Alabama, 1979, pp. 261-273.

[4] S. I. Kartashev and S. P. Kartashev, "A Multicomputer System with Dynamic Architecture," *IEEE Transactions on Computers*, vol. C-28, no. 10, October, 1979, pp. 704-721.

[5] S. I. Kartashev and S. P. Kartashev, "Dynamic Architectures: Problems and Solutions," *Computer*, July, 1978, pp. 26-40.

[6] R. J. McMillan and H. J. Siegel, "The Hybrid Cube Network," *Proceedings of the Distributed Data Requisition, Computing, and Control Symposium*, 1980, pp. 11-22.

[7] M. C. Pease, "The Indirect Binary n-Cube Microprocessor Array," *IEEE Transactions on Computers*, vol. C-26, no. 5, May, 1977, pp. 458-473.

[8] Y. Paxer and M. Bozyigit, "Variable Topology Multicomputer," *Proceedings of the Second Euromicro Symposium on Microprocessing and Microprogramming*, Venice, 1976, pp. 141-149.

[9] L. D. Wittie and A. M. van Tilborg, "MICROS, A Distributed Operating System for MICRONET, A Reconfigurable Network Computer," *IEEE Transactions on Computers*, vol. C-29, December, 1980, pp. 1133-1144.

[10] A. Despain and D. Patterson, "X-Tree: A Tree Structured Multi-Processor Computer Architecture," *Proceedings Fifth Annual Symposium on Computer Architecture*, 1978, pp. 144-150.

[11] A. Goyal and G. J. Lipovski, "Reconfigurable Hierarchical Rings," *Proceedings of the Distributed Data Acquisition, Computing, and Control Symposium*, 1980, pp. 3-10.

[12] D. DeGroot and M. Malik, "Resource Allocation for Macropipelines," *Proceedings of the Distributed Data Acquisition, Computing, and Control Symposium*, 1980, pp. 23-27.

[13] B. Elspas, "The Theory of Autonomous Linear Sequential Networks," *IRE Transactions on Circuit Theory*, January, 1959, pp. 45-60.

[14] N. Zierler, "Linear Recurring Sequences," *J. Siam*, 7(1), 1965, pp. 31-48.

[15] W. H. Kautz (ed.), *Linear Sequential Switching Circuits*, Holden-Day, 1965.

[16] S. W. Golomb, *Shift Register Sequences*, Holden-Day, 1967.

[17] T. L. Booth, *Sequential Machines and Automata Theory*, 1967.

DESIGN OF A GENERAL-PURPOSE MULTIPROCESSOR WITH HIERARCHICAL STRUCTURE

J. Sasidhar and Kang G. Shin
Electrical, Computer, and Systems Engineering Department
Rensselaer Polytechnic Institute
Troy, New York, 12181

Abtract -- In this paper we consider the design of a Hierarchical Multiprocessor (HMp) for general-purpose applications. The main attribute of the HMp is the simplicity of the interconnection network. The HMp consists of clusters of processors connected hierarchically for both data processing and data distribution.

There are two levels of interprocessor communications in the HMp, an implementation of which is developed on the basis of the monitor concpet. Using queueing network models, the performance falloffs due to shared hardware is also analyzed, and the optimum number of processors in each cluster is then determined.

1. INTRODUCTION

In the past few years multiprocessor architectures have gained considerable attention due to the availability of the powerful but inexpensive microprocessors and memories in the computing arena. The question that still remains to be answered satisfactorily is whether the microprocessor can be utilized as a building block for large general-purpose computer systems, thereby achieving a higher performance/price ratio as compared to traditional uniprocessor architectures. A survey of existing multiprocessor organizations can be found in [1]. The unsolved issues associated with multiprocessors are also well discussed in [2].

The proposed architecture called the hierarchical multiprocessor (HMp) has been considerably influenced by both the Cm* architecture at Carnegie-Mellon University and the Hierarchical Multicomputer Organization at State University of New York, Stony Brook.

The central idea in Cm* [1,3,4] is the grouping of processors into clusters and the concept of a task force [5,12] which is ideal for a cluster organization. The main drawback in Cm* however is the integration of the I/O units into the system. The I/O units are made dependent on individual processors which results in an unstructured operating system and gives rise to reliability and utilization problems. This to some extent has been solved by the Hierarchical Multicomputer Organization [6,7] where the idea of separating the control and data moving functions has been suggested. In the proposed architecture this idea has been extended to include the users interface to the system.

The HMp has been designed to minimize the interconnection complexity of the system and uses only a few types of functional units as building blocks for the system. The aim of the design has been to create a general purpose multiprocessor with no restriction on the types of algorithms which it can exploit.

This paper is organized as follows. Section 2 discusses the HMp architecture of the multiprocessor in some detail. Section 3 describes the structure of the kernels necessary to implement monitor primitives [9,10] for synchronization purposes. Finally Section 4 deals with the performance falloffs due to shared hardware resources and analyzes the performance of the system in terms of the processing rate. Conclusion follows in Section 5.

2. ARCHITECTURE

2.1 Overview

A multiprocessor should be able to exploit the explicit or implicit parallelism given by an algorithm. This is possible only if the number of steps in each parallel path is greater than a fixed minimum so as to offset the communication overhead existing between interacting tasks. Thus the extent of exploitable parallelism depends on the communication overhead between interacting processes. The hardware interconnection which has the lowest associated communication overhead is the shared memory concept. The restriction of this approach is that the communication overhead increases as the number of processors in the system increases.

To circumvent this problem, a system with two levels of communication is developed. At the first level of communication the communication time is kept to a minimum and independent of the total number of processors in the system. At the second level of communication the communication time is sacrificed for extensibility and hardware interconnection costs. The processors in this architecture are grouped into clusters.

The significance of this approach becomes more evident when we examine the property of process locality [2]; which states that inter-

This work is partially supported by the National Institute of Justice, U.S. Department of Justice, under Contract J-LEAA-014-78, and the National Science Foundation under Grant ENG-7911347.

0190-3918/81/0000/0141$00.75 © 1981 IEEE

action within a defined group of processes is frequent, whereas interaction between different groups is infrequent. If processes are allocated to processors such that the processes of the same group reside in any single cluster, then the communication overhead would correspond to that of a closely coupled system.

The HMp consists of two hierarchies, namely the processing hierarchy and the data distribution hierarchy. The data distribution hierarchy handles the file management functions and the processing hierarchy handles the processing functions. To differentiate the processors in the processing hierarchy from those of the data distribution hierarchy, the former are referred to as the P-processors and the latter as the D-processors.

The processing hierarchy consists of processing modules grouped into clusters which are then organized in a hierarchical fashion. Associated with each cluster of processors in the processing hierarchy is a parent processor which is part of the cluster one level higher in the hierarchy. Each cluster in the processing hierarchy has associated with it a D-processor. The D-processors of the system with the secondary memory from the data distribution hierarchy. The cluster organization is presented first and then the system organization is described in some detail.

2.2 Cluster Organization

The cluster consists of processing modules which have a sibling relationship to each other and they share a common memory by means of a time shared common bus (Fig. 1). Conflicts of access to the common bus are resolved by the bus arbiter, and the handshaking required for gaining control of the bus is done by the switch, which is a subsystem of each processing module. Each processing module in the cluster consists of a processor, local memory, a swtich, a DMA interface to the D-processor and serial links to its child and parent processing modules.

2.2.1 The Switch.

The processor does not distinguish between accesses to common memory and its local memory. It is the responsiblity of the switch to recognize a nonlocal reference and initiate the necessary handshaking to perform the memory access. To access common memory, the switch has to gain control of the common bus by handshaking with the arbiter. The switch has been given the capability of buffering a single data word which has to be read from or written into the common memory. Also for ease of implementing process synchronization primitives the switch has been given the capability of requesting the control of the bus at two levels, depending upon the status of the switch (Section 3 will deal with this in some detail). This status is explicitly set by the processor and is alterable only by the processor.

2.2.2 The Bus Arbiter.

The bus arbiter is moderately complex since it can grant control of the bus at two levels and there are certain rules it has to follow in order to preserve the integrity of the interprocess synchronization primi tives (This will be discussed later). The arbiter provides a round robin service to requesting processors to ensure that all requests will be honored in due time. Each of the switches has two individual request lines to the arbiter for requesting control of the bus at the two levels, and correspondingly there are two grant lines to each switch.

2.2.3 The Control Links.

Since the cotrol links are serial in nature, we need additional processing at both ends of the link for buffering a message, generating interrupts and setting up flags at the completion of a message transfer. A parent processor can interrupt its child processor through the serial control link at two levels: one level is maskable and the other is nonmaskable. An interrupt at any of the two levels will cause the child to execute a message receiving routine which is a part of the kernel software. In normal operation a parent interrupts its child at the maskable level. This implies that if the child is inside the kernel, the interrupt will remain pending until the child exits from the kernel. But if the parent has reason to believe that a malfunction has occurred, it interrupts at the nonmaskable level. The child on the other hand can interrupt its parent through the serial control link only at the maskable level. This ensures that the parent can still function with a faulty child processor.

2.2.4 The DMA Interface.

The DMA interface transfers blocks of code/data to and from the local memory of the D-processor associated with the cluster and the local memories of the processing modules. The DMA interface is also used in setting up code/data in the common memory of a cluster. To start a block transfer, the parent processor of the cluster gives the order to the D-processor including the identity of the file, processor number, starting address, the length of the block and the direction of transfer. The D-processor then sets the address registers and the word count register of the DMA interface and initiates the transfer. On completing the transfer the DMA interface informs the D-processor which in turn informs the parent processor of the cluster.

2.3 Data Distribution Hierarchy

For each of the clusters in the processing hierarchy there is an associated D-processor which handles the transfer of code/data into or out of the cluster. Since most of the processing is done at the bottom level of the processing hierarchy, most of the file transfers in the system will be handled by the associated leaf D-processors. Thus we need high capability data links between the secondary storage units and the bottom level D-Processors of the data distribution hierarchy. To perform the file management functions of the system, the D-processors need to exchange short control messages between themselves. The D-processors are interconnected hierarchically by means of serial links and since at times there will be file transfers on these links, a packet switching communication system has to be implemented.

All the human interfaces to the system are connected to the data distribution hierarchy and so it acts as the source of all tasks which need processing power from the processing hierarchy. New processes enter the processing hierarchy via the serial control links interconnecting the two hierarchies and the results enter the data distribution hierarchy in the same way. The D-processors act as command message interpreters in the same sense as the 'shell' of the UNIX system [11] and create processes which execute the command message in the processing hierarchy.

2.4 Root Cluster Organization

The two hierarchies of the system namely the processing hierarchy and data distribution hierarchy are merged at the top by a root cluster whose organization is slightly different from that of the other clusters in the system (Fig. 2). The root cluster consists of both P-processors and D-processors sharing a common memory. The processing hierarchy is attached to the P-processors of the root cluster and the data distribution hierarchy is connected to the D-processors of the root cluster.

Tasks of the operating system executing in the root cluster can oversee both the processing and the data distribution hierarchies. Typically these tasks would consist of cooperating parallel processes and since the processors in the root are tightly coupled, it leads to an efficient implementation.

3. SYNCHRONIZATION AND INTERPROCESS COMMUNICATION

For any multiprocessor architecture it is essential to have an efficient implementation of the synchronization and interprocess communication primitives. Microprocessor architectures being introduced at present have capabilities to support two execution modes, features for memory protection and hardware support for task switching. These hardware supports simplify the implementation of efficient primitives.

3.1 General Approaches

Synchronization and interprocess communication can be implemented by using any of the following mechanisms: semaphores, mailboxes, message queues or monitors. Each of these mechanisms is logically equivalent to the other.

From a software point of view, monitors [5] are an ideal solution since they help in specifying the precedence relationships in a structured fashion. Monitors consist of shared data and procedures which operate on the shared data. A process can operate on the shared data only through the procedures of the monitor and not directly. Since only one process can be inside the monitor at any time operations on the shared data are mutually exclusive. The primitives required to support monitors are: entering a monitor, exiting a monitor, signalling a condition and waiting for a condition [10].

Since we have interprocess communications at two levels: 1) between processors in the same cluster and 2) processors in different clusters, we will first discuss the implementation of the primitives at the cluster level and then at the system level.

3.2 Synchronization at the Cluster Level

To be as general as possible we assume that there can be more than one process assigned to a single processor at any time and that the implementation should handle both static and dynamic creation of tasks.

To limit the loading on the central resources of the cluster (i.e. the common memory, the common bus and the parent processor), we decided to define two kernels; the processor kernel (called the P-kernel) and the cluster kernel (called the C-kernel). The P-kernel resides in each processor and manages the processes residing in that processor. The C-kernel handles the monitors of all processes residing in that cluster and is located in the common memory of that cluster. Since the kernels handle the system queues, they themselves should not be interrupted to assure that no race conditions develop. This is easy to implement for the P-kernel since on entering the kernel it can disable all interrupts (including the interrupts from the parent processor). But mutual exclusiveness for the C-kernel has to be implemented by using additional hardware. This mutual exclusiveness is taken care of by the arbiter and is discussed later.

The C-kernel provides mutual exclusiveness of the monitors by associating with each monitor a flag which records whether the monitor is busy or not. Thus the C-kernel provides a means of having more than one monitor busy at the same time. The C-kernel maintains the queues for processes waiting to enter a monitor and queues for each condition. The P-kernel queues contain the full status of the processes necessary to restart the processes whereas the queues in the C-kernel contain only minimal information to identify the processes. This is to ensure that the loading on the central resources is as minimum as possible.

The tasks running in the processors of the cluster are in the user mode, and execution of any of the synchronization primitives causes a trap to the P-kernel of the processor. The P-kernel saves the status of the user process and then tries to enter the C-kernel and waits if busy until it is free. This does not load the central resources but only idles the processor. Once the C-kernel is free, the P-kernel enters it and performs the operations corresponding to the desired primitive operation. It should be again stressed that the C-kernel can be entered only through the P-kernel and not directly by the user processes. The operations done after entering the C-kernel for the case of one primitive is discussed and the rest are similar.

Exiting from a monitor: The C-kernel checks the queue associated with the monitor and if there is no process waiting to use the monitor, it resets the monitor flag and exists to the p-kernel. If hwoever, there is a process waiting, it sends the identification of the waiting process to the parent processor (to be woken up) and then exits without resetting the monitor flag. The P-kernel then passes control back to the user process.

When the parent processor receives the message for waking up a process, it interrupts the processor which has that process in its wait queue, and thus we have a "positive wakeup of the process" [8]. The parent processor does not require to keep track of where the process is residing since the message from the C-kernel contains both the identification of the process and the physical processor in which it is residing.

For the case of dynamic creation of processes, the technique used is quite similar. Execution of a FORK statement by a user process causes a trap into the P-kernel and the P-kernel then requests the parent processor to create the required number of processes. It is understood that any data to be shared has initially been stored in the common memory of the cluster when the task itself was allocated. Once the Parent processor acknowledges the message, the P-kernel gives control back to the user process.

3.2.1 Functions of the Arbiter. Function of the arbiter are:

1. to give mastership of the bus to a requesting processor, and
2. to keep track of the condition of the C-kernel (i.e busy or not) and thus provide mutual exclusion of the C-kernel.

Each processor can request use of the bus at two levels, one for using the C-kernel and the other for using code/data outside the C-kernel (i.e. the monitor procedures). This is implemented by using an independent set of two request and two grant lines for each processor.

If the processor wants to enter the C-kernel, it sets a status bit in the switch of the processor module. The switch then asserts the C-request line and if the C-kernel is not in use, the arbiter asserts the C-grant line. The switch then sets a flag indicating to the processor that it can now proceed to use the C-kernel. Then for each access to the common memory the switch asserts the B-request and performs the memory access after the arbiter asserts the B-grant line. Once the processor exits from the C-kernel, it resets the status bit in the switch which causes the switch to deassert the C-request line. If the processor wants to access code/data which is outside the C-kernel, then the processor does not set the status bit in the switch. For each access to the common memory the switch only asserts the B-request line.

The arbiter provides mututal exclusion of the C-kernel by asserting C-grant to only one of the processors which has its C-request line asserted and ignores all other requests for the C-kernel until the corresponding processor exits from the C-kernel. The arbiter can give mastership of the bus to a processor with only its B-request line asserted even though there is another processor in the C-kernel. This does not create race conditions but does improve the utilization and availability of the time shared bus. Thus the arbiter provides a round robin service for the use of the bus (by asserting B-grant) and a round robin service for the use of the C-kernel (by asserting C-grant).

3.3 Synchronization at the System Level

We have so far discussed the implementation of the inter-process synchronization primitives at the cluster level. There can be two approaches for implementing these primitives at the system level. One approach would be to have processes residing in different clusters communicate via messages. This involves the complexity of having two types of communication primitives, one at the cluster level and the other at the system level. It suffers from the fact that the architecture is not transparent to the systems programmer.

The second approach is to implement the synchronization primitives at the system level by using the monitor concept. This provides transparency and makes it easier for the system programmer to implement the system software. Since monitor procedures access only data local to the monitor, all interactions between the calling process and the monitor procedure is made via arguments. Thus execution of a monitor procedure whose physical location is in another cluster can be implemented via messages. The monitor procedures will physically reside at a common ancestor cluster of the two clusters in which the communicating processes are present.

The basic kernel of the operating system which handles the processes and the inter-process communication is described below. This kernel code is replicated in all the P-processors of the HMp. The basic kernel consists of essentially two levels. The first level consists of the P-kernel and the C-kernel. The second level consists of the message handler which implements the primitives necessary for a process to switch processors. A process can execute a monitor procedure whose physical location is in another cluster by migrating to that cluster. The message handler can create, destroy or wake up processes residing in the processor. This is necessary for implementing the monitor primitives and also serves to implement the concept of coscheduling the task force [12].

Above this basic kernel, a distributed operating system such as Medusa [12] can be implemented. Medusa consists of a set of disjoint utilities (each of which is a task force) which communicate via messages using a structure called pipes [11]. This structure can be implemented by using the monitor primitives made available by the message handler.

4. PERFORMANCE ANALYSIS

Since the present organization consists of two levels; we first determine the performance of a single cluster treating it as a single unit. Using these results we evaluate the performance of the entire system. In this analysis, performance refers only to the throughput of the system and not to any other factors.

4.1 Performance of a Single Cluster

The resources which are shared by the processors of a single cluster are the time shared common bus, the common memory, the D-processor and the parent processor. Interference in sharing these resources results in a decrease in the performance of each processor.

The reason for analyzing the cluster is to determine the optimum number of processors for a cluster and to find the limiting value of performance due entirely to hardware constraints. We are at present not considering the performance falloffs due to software precedence relationships which do affect the final figure of performance.

The performance of a cluster of processors is being evaluated by using queueing network models. The first queueing network models the performance falloffs due to common memory interference. The second queueing network models the performance falloffs due to the parent processor and the D-processor of the cluster.

4.1.1 Common Memory Interference. Let us define the memory cycle time (Mc) as the time taken to read or write a single word into common memory once the switch has mastership of the time shared common bus. Let us also define the access interval time (Ai) as the time interval between two consecutive accesses to memory by a processor. The accesses can be either for code or data. Even though there is a variation in the access interval times we assume for simplicity that it has a constant value [13].

Let us further denote the integer value [Ai/Mc] by m. The greater the value of m, the less the interference due to the shared resource and thus greater is the performance of the processors in the cluster. If the common memory is implemented in bipolar technology and the processors in MOS technology then the value of m is usually in the range 3 to 10 and thus can be used as a design parameter. Using bipolar memories for the common memory is reasonable since the memory requirements for shared information is small.

To analyze the interference in accessing the common memory, we should have an understanding of the nature of the stochastic process which describes the accesses to common memory by each processor. Reviewing the use of common memory we find that the common memory is used only for monitor procedures, and their associated data and control mechanisms. When a processor starts executing a monitor procedure, all memory accesses will be to the common memory since both code and data will reside in the common memory. Thus successive accesses to common memory by the same processor cannot be modeled as independent random variables.

If a processor executes any of the monitor primitives, it begins executing the code of the C-kernel and then, depending upon the type of monitor primitives desired and the state of the desired monitor, one of the following actions takes place.

1. Processor starts to execute the monitor procedure.
2. It wakes up a process residing in another processor to execute the monitor procedure.
3. It waits for another process to signal it and at that time it continues to execute the monitor procedure.
4. It does not execute the monitor procedure nor does it wake up another process to execute the monitor procedure.

Examining the above cases we find that for the first two cases the monitor procedure is executed either by the same processor or by another immediately following the execution of the C-kernel. In the last two cases the monitor procedures are not executed, and the next time the processor accesses the common memory it would execute the C-kernel. Once a monitor procedure is being executed, the processor has to execute one of the monitor primitives to exit from the monitor. The above four cases can be condensed to the following two cases.

1. The processor executes the C-kernel, then the monitor procedure and finally the C-kernel; each one immediately after the other.
2. The processor executes the C-kernel and then starts executing code from its local memory and then the C-kernel when it comes across a monitor primitive.

Both of these cases can be represented by one unified model which is as follows: the processor first executes the C-kernel and then a monitor procedure and then code from its local memory and then finally the C-kernel again.

4.1.2 Closed Queueing Network Model. We can model the memory contention problem as a closed queueing network model with appropriate service times and scheduling policies. The resource being shared is the common memory and the service time it provides can be measured in terms of the number of common memory accesses.

The number of common memory accesses needed to execute a portion of a monitor procedure sandwiched between two consecutive monitor primitives can be treated as a random variable with an exponential distribution. The number of local memory accesses between two monitor calls can also be treated as a random variable with an exponential distribution. The nunber of common memory accesses needed to execute the monitor primitive by means of the C-kernel is assumed to have an exponential distribution. Even though the actual distributions might not correspond to our assumptions, queueing models are generally robust and do give good results.

The queueing network consists of three nodes two of which consist of parallel servers and the third a single server (Fig. 4). The first node consists of n servers where n is the number of processors in the cluster. The service time for these servers corresponds to the distribution of the number of local memory accesses between two consecutive monitor calls. Node 1 is of type-D [15] since the customers are delayed independently of other customers at this service center.

The common memory can be treated as m virtual parallel servers since effectively there can be m common memory accesses in time period Ai. Also note that we cannot give more than one common memory access to a processor in a given time period Ai (Fig. 5). Of the m virtual servers of the common memory one server services the C-kernel queue which is Node 2 of the queueing network. The rest of the m- 1 virtual servers service the monitor queue and form Node 3 of the queueing network.

In the actual system however, the server servicing the C-kernel queue would service customers in the monitor queue if there are no customers in the C-kernel queue. Therefore the performance characteristics obtained by this queueing network model gives the lower bound of the actual performance. The upper bound of the performance can be easily obtained by having an additional parallel server at Node 3.

The scheduling policy used for Node 2 and Node 3 of the queueing network is FCFS. In the actual system the type of scheduling used to service the monitor queue is round robin. As we are only interested in the mean values of the waiting time and the mean queue lengths, we can assume an FCFS service mechanism. As long as the scheduling is independent of the service requirements of a customer, the mean values do not change [14].

The analytical solution of the queueing network was carried out by the recursive algorithm in [15]. The results shown in Fig. 6 correspond to mean value service times indicated below (values normalized by the mean number of local memory accesses needed to execute a block of local code sandwiched between two consecutive monitor calls):

1. Mean number of common memory accesses needed for executing the C-kernel: Case 1: 2.5%; Case 2: 5%.
2. Mean number of common memory accesses needed for executing monitor code sandwiched between two monitor primitives: Case 1: 10%; Case 2: 20%.

The results give the lower and upper bounds of the performance of the cluster with common memory interference for m = 3 and m = 4.

<u>4.1.3 Parent Processor Interference</u>. To evaluate the type and frequency of demands placed on the parent processor, let us consider its functions. The parent processor basically consists of a message handler and other user or system processes. The message handler handles all incoming messages from the child processors, the D-processor and the parent of the processor itself. There are three types of messages which can occur and their description follows.

The first type is a synchronization request between processes residing in the same cluster. The amount of processing time needed to process this type of message is small but their frequency of occurrence is high. The parent processor should be very responsive to these requests since any delay would entail a decrease in the performance of the cluster.

The second type of message is a request for execution of a monitor procedure residing in the parent processor cluster. This entails the creation of a process by the message handler by inserting its description in the ready queue of the P-kernel. The created process then needs processing time to execute the monitor. Then the message handler has to reconvert the process into a message form and send it back to the child processor. The frequency of occurrence of these type of messages is small but the processing time needed is higher in relation to the messages of the first type.

The third type of message is a request for the transfer of code/data into or out of the local memories of the child processor. This message should be redirected to the D-processor and once the transfer is over the reply from the D-processor should be sent to the child processor. The frequency of occurrence of these type of messages is low and the processing time needed is also low.

<u>4.1.4 Queueing Model</u>. We can now treat the system as a closed queueing network model with the parent processor and the D-processor as servers with the processors in the cluster and the parent of the processor itself as the customers (Fig. 7). The processors are assumed to have a think time which is exponentially distributed. After each think period a processor sends a message to the parent processor which acts as the server. The message can be of any type and it is placed in the message queue. The service time requirements for customers in the message queue are assumed to be exponential. This service time includes the time taken to read the message from the hardware buffer; perform the synchronization or create a new process by entering it in the run queue of the kernel.

The customers coming out of the first server can take three paths where each path has an assigned probability. The three cases are as follows:

1. If the customer needs no further processing, then it returns to the processor from which it originated (P21).

2. If further processing is required, then the customer is put back in the message queue which is serviced by the parent processor (P22)

3. If further processing is required from the data processor, then it is put in the data queue. After it receives service from the data processor, it is put back in the message queue (P23).

Both the queues in the model namely the message queue, and the data queue are served in a FCFS discipline. Synchronization messages are being given higher priority in the queueing model since additional processing needed by a message is being postponed until the backlog of messages have been serviced.

P22 with the average service time for customers in the message queue determines the total service time requirements for messages which need execution of monitor procedures. The service time for customers in the data queue is the total time the system takes to transfer the code/data to or from the primary memory. This service time is also assumed to have an exponential distribution.

The analytical solution of the queueing network was again obtained by the recursive algorithm in [15]. The results shown in Fig. 8. correspond to the following mean value service times (the values are normalized by the service time for executing code residing in the cluster, sandwiched between two consecutive calls to the parent processor):

1. Mean service time taken by the parent processor to perform a synchronization request: 2.5%

2. Mean service time for the D-processor to transfer code/data in and out of the cluster processor's private memory: 20%

The routing probabilities for the closed queueing network was taken as follows:

1. Case 1: P21=0.8, P22=0.1, P23=0.1

2. Case 2: P21=0.7, P22=0.2, P23=0.1

4.2 The System Performance

From the analysis so far carried out we have to arrive at the figure for the optimum number of processors in each cluster. Since m, the figure of merit of the common memory can be varied within a reasonable range; it can be varied such that the parent processor becomes the critical shared resource of the cluster.

Assuming that we desire at least 90% of the ideal performance (i.e. when there is no interference), we come up with the figure of 15 processors from the performance curves of the parent processor (Case 1). Since the parent processor is also a resource for the grandparent of the cluster, the optimum number of processors for the cluster would be equal to 14.

From this optimum number of processors and the curves for the performance falloffs due to the common memory interference we can determine the desired value of m, such that the critical resource is still the parent processor. For m equal to 3 and the number of processors equal to 14 we have the performance due to common memory interference as 97% of the ideal case for Case 1.

The performance falloff when both the shared resources are present can be taken as the sum of the individual performance falloffs as a first approximation. Therefore the performance of the cluster with both the shared resources present will be 87% (=100-(100-90)-(100-97) of the ideal performance. Since the number of processors in the cluster is 14, the net cluster performance will be equivalent to that of 12 processors (14*87%).

Assuming that most of the actual processing takes place in the leaf clusters, the total system performance can be written as the product of the cluster performance (net performance) and the number of leaf clusters in the processing hierarchy of the computer system. The above analysis assumes that the P-processors at the higher levels are busy synchronizing and performing other communication tasks.

5. CONCLUSION

We have presented the architecture of the HMp, the synchronization primitives and finally the performance of the system based on these primitives. Our future work will concentrate on the interesting aspects brought up by this architecture, some of which are given below.

The HMp has an upper bound on the number of levels it can have. This depends on the higher level processors becoming the bottlenecks in the system. This is thus related to the number and locality of the inter-cluster messages which further depends on the operating system structure.

The effects of software precedence has to be introduced into our queueing models for determining the actual performance falloffs. This will be useful in determining the effects of both software and hardware constraints in the system. We know by intuition that the figure for the optimum number of processors in each cluster will increase, when these effects are taken into account.

The files in the secondary memory should be distributed such that the time to access them from any point in the processing hierarchy is a minimum. The modelling of such a system will involve the actual hardware being used and the bandwidth of the interconnections in the data distribution hierarchy.

REFERENCES

[1] P. H. Enslow, "Multiprocessor Organization - A Survey," Computing Surveys, Vol. 9, No. 1, March 1977, pp. 103-129.

[2] S. H. Fuller, J. K. Ousterhout, L. Raskin, P. L. Rubinfeld, P. J. Sindhu and R. J. Swan, "Multi-Microprocessors: An Overview and Working Example," Proc. IEEE, Vol 66, No. 2, Feb. 1978, pp. 216-228.

[3] R. J. Swan, S. H. Fuller and D. P. Siewiork, "Cm* - A Modular Multi-microprocessor," AFIPS Conference Proceedings, Vol 46, 1977 National Computer Conference, pp. 637-644.

[4] R. J. Swan, A. Bechtolsheim, K. Lai and J. K. Ousterhout, "The Implementation of Cm* Multi-microprocessor," AFIPS Conference Proceedings, Vol 46, 1977 National Computer Conference, pp. 645-655.

[5] A. K. Jones, R. J. Chansler, I. Durham, P. P. Feiler and K. Schwars, "Software Management of Cm* - A Distributed Multiprocessor," AFIPS Conference Proceedings, Vol 46, 1977 National Computer Conference, pp. 657-663.

[6] J. A. Harris and D. R. Smith, "Hierarchical Multiprocessor Organization," Conf. Proc. 4th Ann. Symp. Computer Architecture, March 1977, pp. 41-48.

[7] R. B. Kieburtz, "A Hierarchical Multicomputer for Problem Solving by Decomposition," Proc. 1st Int'l Conf. Distributed Computing Systems, Oct. 1979, pp. 63-71.

[8] H. K. Reghbati and V. C. Hamacher, "Hardware Support for Concurrent programming in Loosely Coupled Multiprocessors," Conf. Proc. 5th Ann. Symp. Computer Architecture, April 1978.

[9] C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," CACM Vol. 17, No. 10, 1974, pp. 549-557.

[10] R. C. Holt et al., Structured Concurrent Programming, Addison Wesley, 1978.

[11] D. M. Ritchie and K. Thompson. "The UNIX Time-Sharing System Comm.," ACM, Vol. 17, No. 7, July 1974, pp. 365-375.

[12] J. K. Ousterhout, D. A. Scelza and P. S. Sindhu, "Medusa: An Experiment in Distributed Operating System Structure," Proc. 7th Ann. Symp. Operating Systems Principles, Nov. 1979, pp. 115-126.

[13] D. P. Bhandark . "Same Performance Issues in Multiprocessor System Design," IEEE Trans. Computers, Vol. C-26, No. 5, May 1977, pp. 506-511.

[14] L. Kleinrock, Queueing Systems, Vol. 1 & 2, John Wiley and Sons, 1975.

[15] M. Reiser, S. S. Lavenberg, "Mean Value Analysis of Closed Multichain Queueing Networks," JACM, Vol. 27, No. 2, April 1980, pp. 313-322.

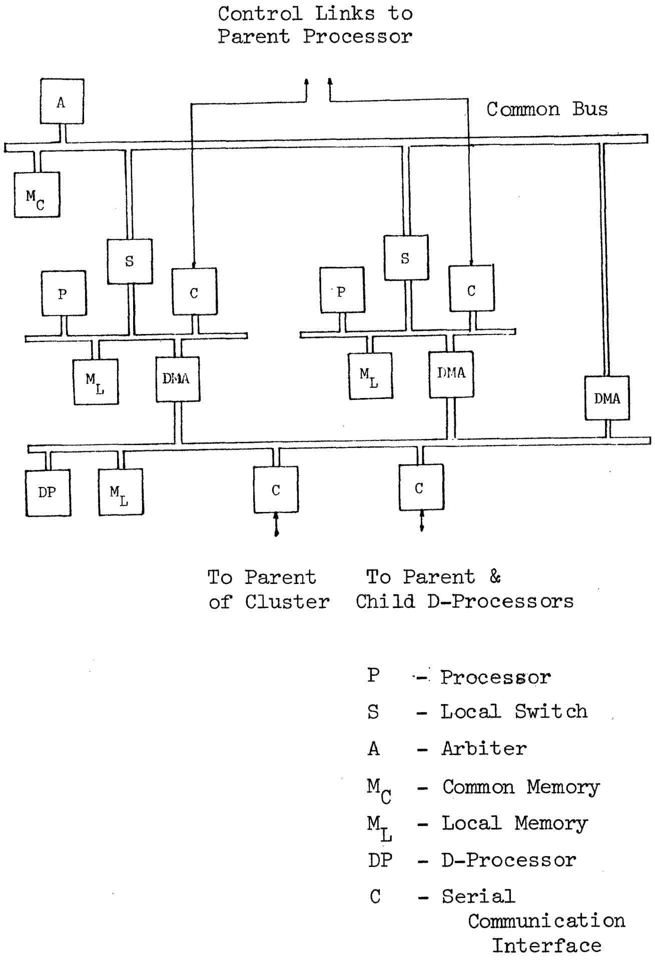


Figure 1. Cluster Organization.

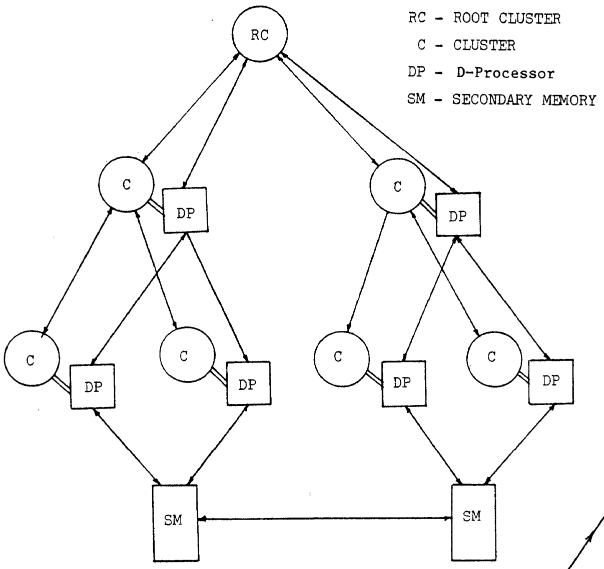


Figure 2. System Hierarchies.

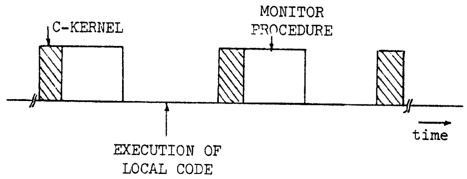


Figure 3. Common Memory Reference Pattern.

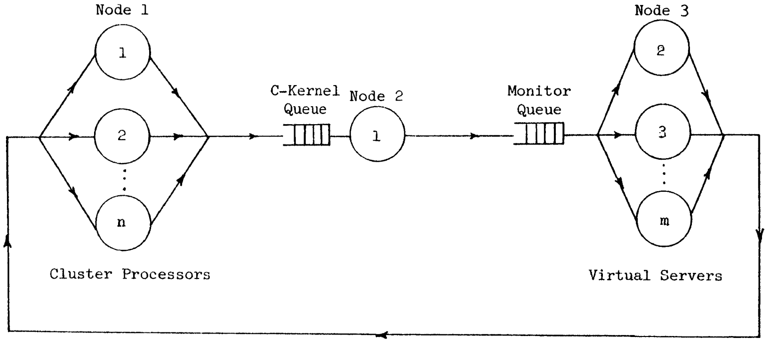
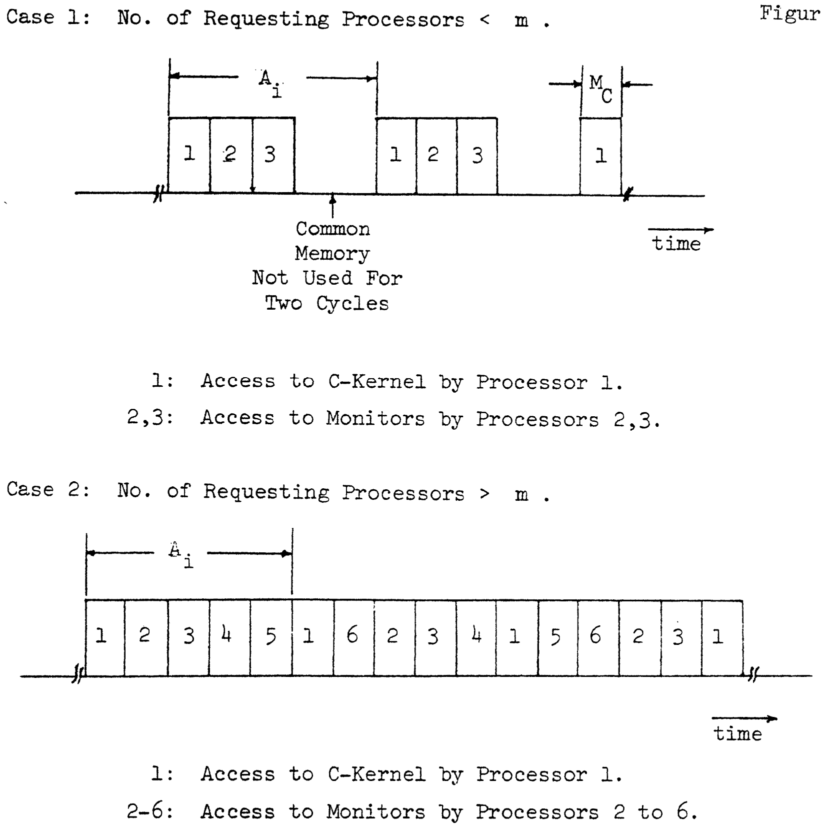


Figure 4. Queueing Model for Common Memory


Case 1: No. of Requesting Processors < m .

1: Access to C-Kernel by Processor 1.

2,3: Access to Monitors by Processors 2,3.

Case 2: No. of Requesting Processors > m .

1: Access to C-Kernel by Processor 1.

2-6: Access to Monitors by Processors 2 to 6.

Figure 5. Individual Access to Common Memory

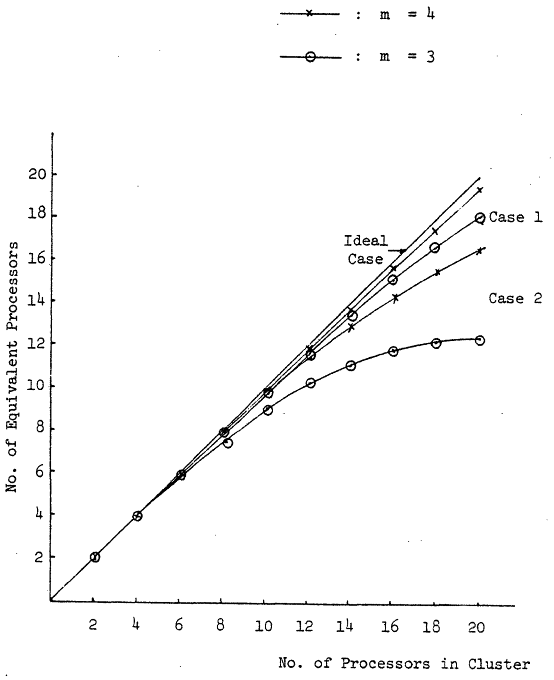


Figure 6. Common Memory Interference.

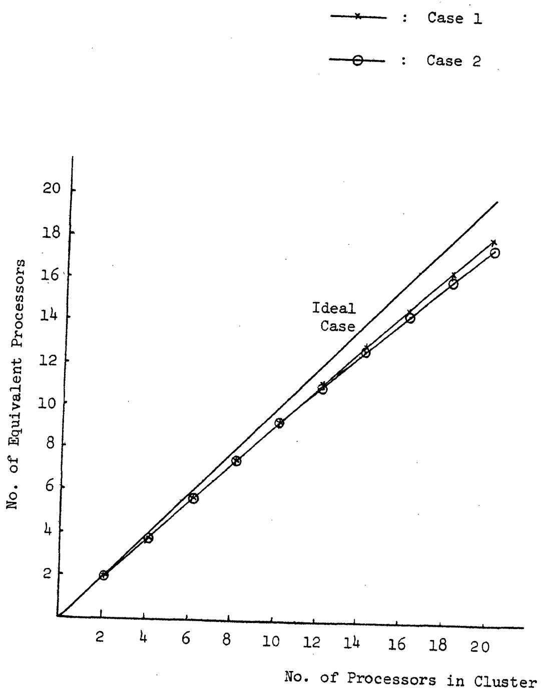
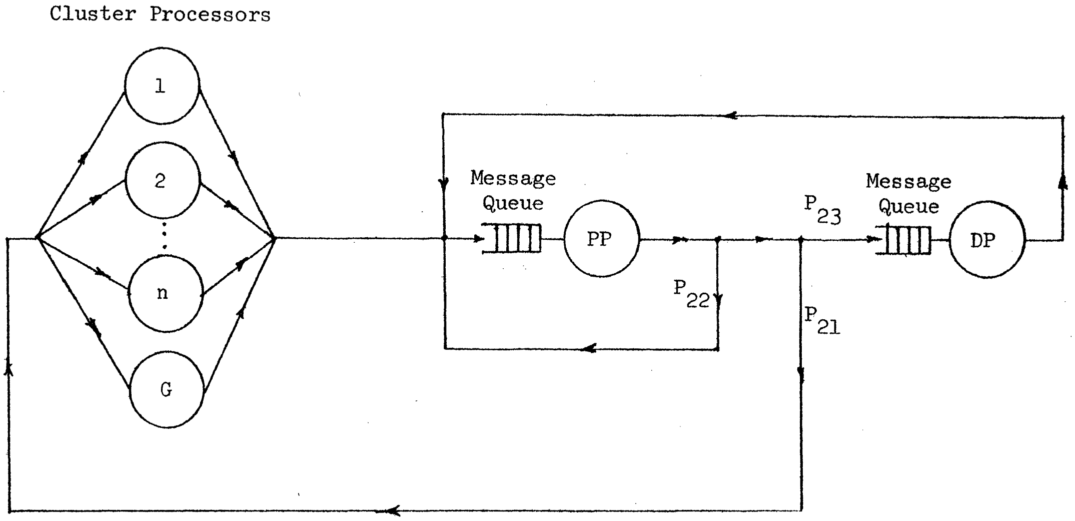


Figure 8. Parent Processor Interference.



G - Grandparent

PP - Parent Processor

DP - Data Processor

Figure 7. Queueing Model for Parent Processor.

A BLOCK-DRIVEN DATA-FLOW PROCESSOR*

By

T.L. Chang, Student Member, IEEE
and
P. David Fisher, Senior Member, IEEE

Department of Electrical Engineering
and Systems Science
Michigan State University
East Lansing, Michigan 48824

Summary

A highly distributed data-flow processor based on block-driven principles is described. Being block-driven, data-flow programs can be executed in functional blocks. As a result, data transfers can be effectively separated into different levels of communication paths. Through the use of a structured computer architecture and a hierarchical data-transfer mechanism within the tree network, this data-flow processor provides programmable communication paths for fast data transfer while at the same time achieving very high levels of concurrency.

Introduction

In a data-driven computing system, the instructions of a data-flow program are normally stored in the instruction memory cells; an instruction cell will be fired whenever the required data are available [1,2]. Based on a variety of forms of parallel routing and parallel computation, a large number of such instructions can be executed simultaneously. However, due to data dependencies, this data-driven approach requires streams of data to be routed back to the instruction memory cells. These data then trigger newly addressed instruction cells. Based on the trend that data transfers are becoming more and more costly when compared to the cost of a unit of computing power, there is a critical need for data-flow structures having efficient data transfer paths [3,4]. Considering the constraint of data dependencies to data-flow instructions, this constraint varies from one instruction to another. Some instructions require one data item from the previous execution and some require two. An instruction with one dependent data item can be executed immediately after the previous operation is completed. However, the execution of an instruction with two dependent data items has to wait until both data items are ready. This results in a different degree of efficiency for transferring these two types of data. A number of data-flow machines, which have been proposed recently or which have already been developed, have paid little attention to this problem. In these machines, data items are transferred on a fixed-length data path basis. As a result, the problem of intercell communication overhead has limited their potential to only a handful of applications [1,2,5].

In an effort to overcome the communication overhead problem, a block-driven approach is explored. By contrast to the "data-driven", the "block-driven" can be best described as the group firing of instructions which belong to a group of composite computations. The result of this group firing is that data paths can be effectively separated. Data transfers among the already executed instructions and the succeeding instruction cells within the same group will have shorter paths compared to those outside the group.

The purpose of this paper is three-fold: First, we describe the block-driven principle and discuss its potential advantages: second, we introduce an hourglass computing model to facilitate different levels of data transfer, and by this model, develop a block-driven data-flow computer architecture; and finally, we present a push-pull data-transfer mechanism.

Block-Driven Data-Flow Principle

The flow of data and control in data-flow programs represent fundamentally a random motion phenomenon. And the requirement for transferring this tremendous number of data links and control signals has made the design of parallel processors extremely complex. Although a large number of algorithms have been designed to make applicative programs more suitable for parallel computations, they have been proved to be efficient only when they are executed in specialized array processors [6,7,8]. Like the structured programming technique being widely applied to the software design, the idea of structural, compound function computations has been gaining a great deal of support recently [9,10]. The block-driven approach is developed here to exploit as much of the structure of data-flow programs and machines as possible. In what follows, we will describe this approach. Two important computational steps in data-flow programs are the branch computation and the joint computation. Branch computations involve a sequence of chain computations with the constraint

*This research was supported in part by the National Science Foundation under Grant No. MCS79-09216.

0190-3918/81/0000/0151$00.75 © 1981 IEEE

that each succeeding computation requires one data item from its previous computation. In other words, a computation branch is represented by a mathematical expression of the form

$$f_n(a_n, f_{n-1}(a_{n-1}, (\cdot\cdot f_2(a_2, f_1(a_1, a_0))\cdot\cdot)$$

in which the f_i's denote the elementary arithmetic function (+,-,*,/) and the a_i's are data. A joint computation is defined as a pair of branch computations which are linked together by a computation node at their ends. A data-flow graph containing two computation branches at a joint is shown in Figure 1. By such linkage, we can form a number of joint computations into a cluster of functional cells or a computational block. Each functional cell is made up of a group of composite instructions and a number of data operands. Data items which are used to initiate the firing of a functional cell are called global data.

In a block-driven computer system, the firing of an instruction cell in the data-flow program is subject to the firing of a functional cell with which this instruction is associated; and the firing of a functional cell is subject to the group firing of a computational block. Through this process, a large number of independent functions can be executed in parallel. Further, complex algorithms can be easily decomposed and thereby effectively executed.

Joint Computation

One important issue of having data-flow programs executed by the block-driven process is concerned with the manner in which a computation is executed locally. To deal with this, we propose a simple and effective processing pair for branch computations. The processing pair consists of a pair of local supervisors and a pair of FLP computing modules connected in a ring configuration (see Figure 2). Data-flow instructions on a pair of branches are executed on an interleaved basis. An example of this interleaved computation is illustrated in Figure 3. The advantages of this approach are: first, the succeeding instructions can be driven by local data with an address field of minimum bits; second, reliability can be greatly improved by connecting the processing elements in this manner.

Hourglass Data-Flow Computing Model

In an attempt to exploit the potential advantages of this block-driven principle, a novel hourglass computing model was developed (see Figure 4). The hourglass is loaded with a pair of "double mirrored" trees and has the computing power elements at one end, the instruction memory cells at the other, and hierarchical tree-structured data paths in between. Based on the block-driven principle, a block of fired functional cells pass through a pipe of instruction buffer units to the block control master, where the functional cells are distributed, and the separated computational branches are executed locally on an interleaved basis in a pair of processing elements. Data flow within the hourglass is guided, primarily depending on whether data are global or local in type, either into transmission paths or reflection paths. The length of the transmission paths is fixed; therefore, there is no preference for all global data transfers. The reflection paths are varied, ranging from the shortest paths, which are localized in the processing pair, to the logarithmic paths within the buffering tree. This hourglass model has highly concurrent activities at both ends, while global data, which link the functional cells, are trickling in between.

Block-Driven Computer Architecture

Various tree-structured computing machines have been proposed [11,12]. The tree machines have very high levels of concurrency and are well suited for implementation with current VLSI technology [11,12,13]. Based on our hourglass model, we propose a data-flow tree machine. It contains two kinds of trees: one is called the buffering tree and the other is the distributing tree (see Figure 5). In what follows, we will briefly describe the structures of these two trees and the associated functional units.

Buffering Tree--The buffering tree is an n-level binary network capable of computing 2^n FLP operations concurrently and transferring the results efficiently. It consists of 2^n-1 nodes and one root node. Each one of these nodes is a controlling buffer, at which each visiting data item will either be buffered down or be transferred out. This choice is based on a number of conditions which will be discussed later. Also associated with each leaf node is a pair of local supervisor and processing element where FLP operations are performed. The buffering tree has two primary roles: first, it is an interconnection network to the 2^n processing elements; second, it is a buffering channel between the processing elements and the instruction memory cells.

Data movements within the buffering tree are based on a two-phase push-pull mechanism. While capable of being pushed forward and pulled backward, local data can be precisely moved from one leaf node to another in a few number of push-pull cycles. Global data, which use the buffering tree as the channel, can be pushed forward through the network and off via the root node in n push cycles.

Distributing Tree--The distributing tree is an m-level pipelined binary switching network. It provides the basic mechanism of routing streams of globally independent data to a set of functional cells. There are 2^m functional cells and data operands tied to the lowest level leaf nodes of the tree. Through the use of an m-bit address header, data items can be routed to their destination cells. As data items enter the root node, they are pipelined through this m-level distributing tree.

As a data item is passed from one node to the lower level node, the select bit in the address header is deleted. As a result, the m-bit header is eliminated from a data item when the routings are concluded.

Block Control Master--The major role of the block control master is to provide the tree machine with concurrent joint computations. The master communicates with all the local leaf supervisors. The master acknowledges when the tree machine is released from one block of computations. Then a following computational block will be sequenced and be distributed over the local leaf supervisors, and a joint computation will be executed.

Instruction Buffer--The instruction buffer unit is used for the queueing of blocks of functional cells which are fired and ready to be executed and is built with intelligent FIFO buffer memories.

Push-Pull Data Transfer Mechanism

Because there are two types of data to be transferred in the same network, each node of the buffering network--a controlling buffer--is designed to work on a two-phase basis. In the push mode, data are pushed forward from the lower-level nodes to the higher-level nodes; whereas, in the pull mode, data are pulled backward in an opposite way. Each data item is tagged with a destination address field and a one-bit data type header. The width of the address is determined by the tree height--the higher the tree height, the wider the field. Specifically, a locally dependent data item has a relative displacement address and a one-bit direction header, while a globally independent data item has an absolute address. The relative displacement address is determined by the distance in which the two communicating leaf nodes are apart and by the relative position in which the two nodes are located (see Figure 6). Data to be pushed forward or pulled backward depend on a one-bit mode control by ORing the one-bit data type header and selected bits in the address field. With a tree height of n, this mode control at the ith level, M_i, is given by

$$M_i = a_0 U(\overset{n-1}{\underset{k>i}{U}} a_k), \quad 0 \leq i \leq n-1,$$

where the notation U stands for the logic OR operation. If the mode control is "1", data will be pushed forward; otherwise, they will be buffered at the node at which they last visited and be ready to be pulled backward.

Global data, which carry a "1" in the data type header a_0, will allow themselves to be pushed forward through the buffering network. However, local data, which carry a "0" in the data type header, can never be pushed forward beyond the root level, because the mode control at the root level is always "0" for these data. Data which have already been buffered down to a node at some level will be pulled backward by one of the two son nodes. The decision is determined by a left-right control--LR_i--at that level, with

$$LR_i = a_n a_i + \bar{a}_n \bar{a}_i , \quad 1 \leq i \leq n-1,$$

where a_n is the direction header which determines whether the data to be directed to their right or to their left. If the left-right control is "1", the data will be pulled backward by the right son node, and if it is "0", they will be pulled backward by the left son node. A three-level buffering tree is illustrated in Figure 7.

Discussion

The block-driven data-flow processor described executes clusters of data-flow instructions in a block of locally tree-structured processing elements. Two classes of data communication paths exist. The first communicates global data among blocks of the locally tree-structured processing elements. The second communicates local data among processing elements within a block. Maximum throughput occurs when the ratio of the local data communication rates within a block is much-much greater than the global data communication rates for the data channels bringing operands into or taking results from a block. So, the granularity of the tasks performed within the structured blocks determines the overall system performance for a fixed number of blocks and processing elements. We are currently assessing the complexity of this data-flow structure in the context of its application to both vector processing and discrete-time filtering.

References

[1] D.P. Misunas and J.B. Dennis, "A Computer Architecture for Highly Parallel Signal Processing," Proceedings of the ACM 1974 National Conference, ACM, N.Y. (Nov. 1974), pp. 402-409.

[2] I. Watson and J. Gurd, "A Prototype Data-Flow Computer with Token Labeling," AFIPS Conf. Proc., Nat'l Comput. Conf., (June 1979), pp. 623-638.

[3] C.H. Sequin, "Single-Chip Computers, The New VLSI Building Blocks," Proc. Caltech Conf. on VLSI, (Jan. 1979), pp. 435-445.

[4] D.A. Patterson and C.H. Sequin, "Design Considerations for Single-Chip Computers of the Future," IEEE Trans. on Computers, Vol. C-29, (Feb. 1980), pp. 108-116.

[5] A.L. Davis, "A Data-Driven Machine Architecture Suitable for VLSI Implementation," Proc. Caltech Conf. on VLSI, (Jan. 1979), pp. 479-494.

[6] H.S. Stone, "An Efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations," ACM Journal, Vol. 20, No. 1, (Jan. 1973), pp. 27-38.

[7] H.T. Kung, "The Structure of Parallel Algorithms," Advances in Computers, Vol. 19, ed. by M.C. Yovits, Academic Press (1980), pp. 65-112.

[8] H.T. Kung, "Let's Design Algorithms for VLSI Systems," Proc. Caltech Conf. on VLSI, (Jan. 1979), pp. 65-90.

[9] D.D. Gajski, D.J. Kuck, and D.A. Padua, "Dependence Driven Computation," Digest of Papers, IEEE Compcon Spring 81, (Feb. 1981), pp. 168-172.

[10] Arvind, "Decomposing a Program for Multiple Processor Systems," Proc. 1980 Int'l Conf. on Parallel Processing, (Aug. 1980), pp. 7-11.

[11] S.W. Song, "A Highly Concurrent Tree Machine for Database Applications," Proc. 1980 Int'l Conf. on Parallel Processing, (Aug. 1980), pp. 259-268.

[12] A.M. Despain and D.A. Patterson, "X-Tree: A Tree Structured Multi-Processor Computer Architecture," Proc. Fifth Symp. on Comp. Arch., (April 1978), pp. 144-151.

[13] E. Horowitz and A. Zorat, "The Binary Tree as an Interconnection Network: Applications to Multiprocessor Systems and VLSI," Proc. Workshop on Interconnection Networks for Parallel and Distributed Processing, (April 1980), pp. 1-10.

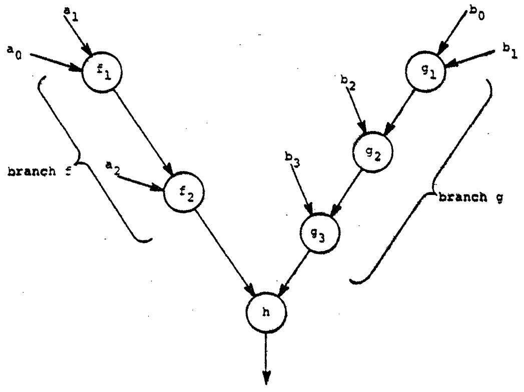


Figure 1. A joint computation data-flow graph h (f,g).

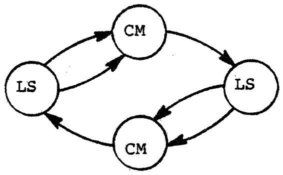


Figure 2. The ring configuration:
CM: computing module and
LS: local supervisor.

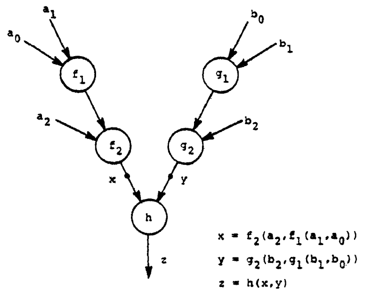


Figure 3a. An example of a joint computation.

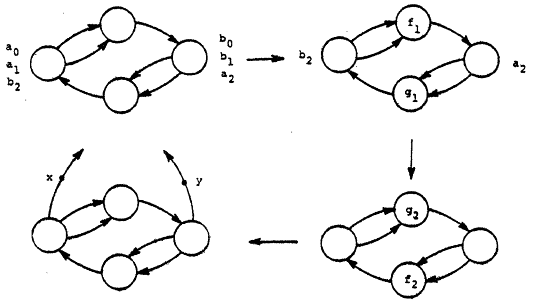


Figure 3b. Steps in an interleaved computation.

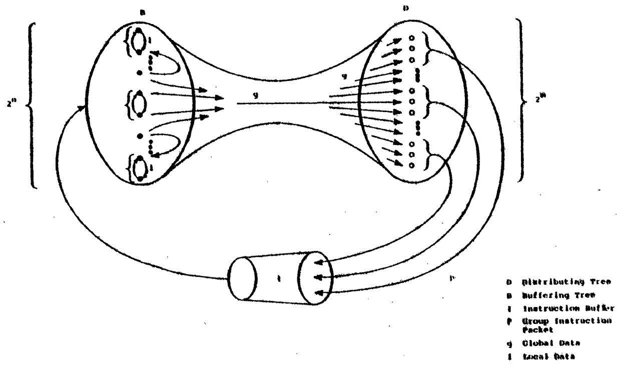


Figure 4. The hourglass computing model.

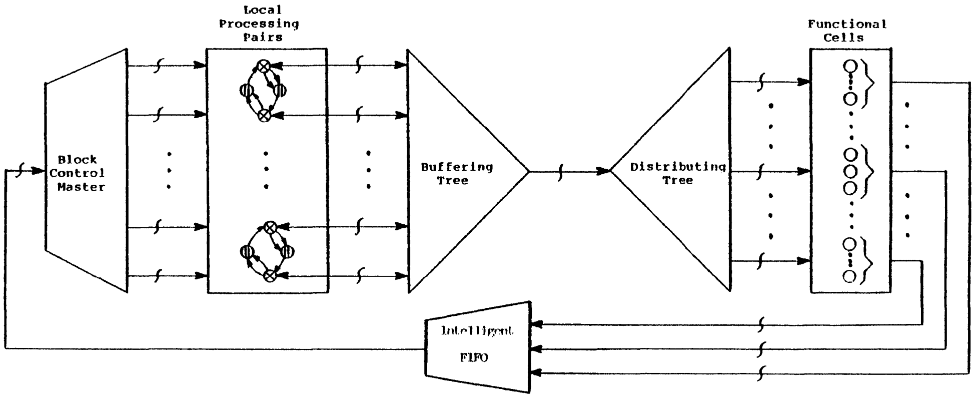
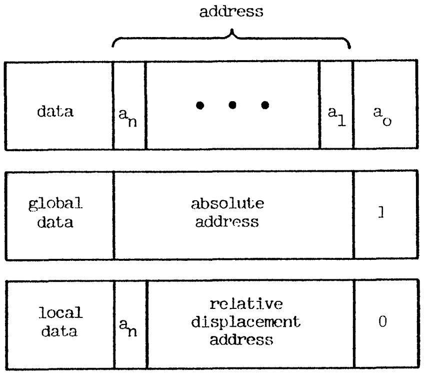


Figure 5. Block-driven data-flow architecture



a_o: data type header

a_n-a_1: address field

a_o = 1

a_m-a_1: absolute address, m $\leq$ n

a_o = 0

a_n: direction header

a_{n-1}-a_1: relative displacement address

Figure 6. Data address syntax.

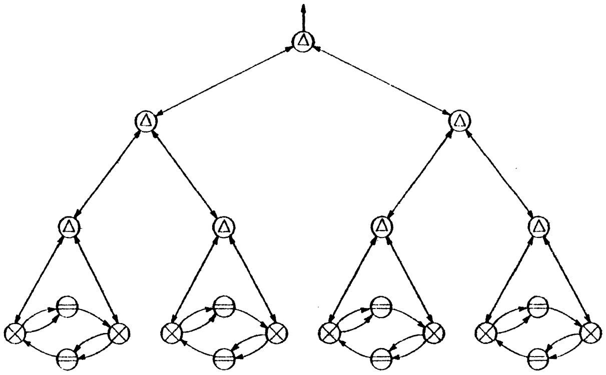


Figure 7. A three-level buffering tree

Processor Allocation in Data Driven Systems - Two Approaches

by

K.J. Mundell, M.W. Linder and S.E. Conry
Electrical and Computer Engineering Department
Clarkson College of Technology
Potsdam, New York 13676

Summary

The topics of data driven computer and program organization have attracted considerable attention in recent years. A number of architectures have been proposed and several prototype machines are now either operational or are being built[1-8]. In addition, various groups have developed languages based on principles compatible with data driven execution[8-13]. In this paper we describe two approaches to the problem of associating operations in a program with the processors which will execute them. The goal is to reduce the time required for program execution by making judicious processor allocations.

In this paper we are concerned with a class of programs written in a textual-form single assignment language. The fundamental control structures assumed are: BEGIN-END, WHILE-DO, and IF-THEN-ELSE. We assume that the parallelism is implicit, rather than explicitly expressed by a COBEGIN-COEND type construct. (The assumptions outlined here are completely consistent with those inherent in most, if not all, of the single assignment languages that have been developed.) The data driven rule for execution implies that unless there is a direct data dependency of one operation on another, two operations can be done in parallel (provided other architectural constraints of the system are met). Thus an operation can be performed at any time after all of its operands are available, independent of any external timing constraints.

An <u>optimal</u> assignment of operators in a program to functional units is one which minimizes execution time. In a data flow system, there are two factors contributing to the overall execution time of a program: the computation time associated with each operation and the time required for transmission of results to the appropriate destinations for use as operands. It is not difficult to show that the problem of obtaining optimal allocations is NP-complete, hence the cost of doing so is prohibitive.

For this reason, the goals of our processor allocation schemes are threefold. First, an allocated program should take advantage of as much of the inherent concurrency as possible. Secondly, since we believe that communication delays can have a significant effect on execution time, the time lost to interprocessor communication should be reduced wherever possible. Finally, the amount of analysis at runtime needed to perform any necessary dynamic allocation should be minimized. Thus we attempt to allocate as completely as possible prior to runtime.

The problem of allocating operations in a program to functional units in a data driven machine can be divided into two phases. The first of these involves reducing the magnitude of the problem by decomposing the program into smaller portions that are easier to analyze for allocation. The second phase deals with actually performing the allocation. In the paragraphs which follow, we describe two sets of algorithms for solving the problem. Each has been implemented in PASCAL and data has been gathered with respect to its efficacy and complexity.

Our first approach to the problem of resource allocation in data flow systems involves decomposing a program written in a block structured single assignment language into smaller segments according to the level of nesting associated with a block and analyzing each "local" subprogram to determine the implicit parallelism and any inherent sequential restrictions. The statements in each program segment are then assigned to processors, and the results of the "local" allocation are propagated for use in allocating operations in its containing block.

As was previously mentioned, an algorithm for performing this analysis and allocation has been developed and implemented in PASCAL. Its implementation is a family of mutually recursive routines which accomplish the decomposition, producing lists of statements that can be done in parallel. These groups of "parallel" statements are then analyzed to determine what data dependencies exist between them. A processor allocation for each block is then produced and returned for use in obtaining a global allocation.

In this approach, allocation within each level of nesting is performed just after that segment has been analyzed. Two criteria are used in determining which processor should perform a given operation. If several statements are constrained to be executed sequentially, they are assigned to the same processor, that processor being the one with the smallest number of statements already assigned to it.

A second approach to the problem of processor allocation has also been investigated. This approach also involves decomposing a program into smaller segments, but in this case the decomposi-

0190-3918/81/0000/0156$00.75 © 1981 IEEE

tion takes a different form. Each block is first analyzed in order to determine the global sequential restrictions. In finding these restrictions, a data dependency graph is constructed which exposes the dependencies that imply sequential restrictions. Given this graph, an allocation is obtained by traversing the graph and taking two factors into consideration at each stage: which operations have their operands available and the amount of communication overhead that would be involved in allocating a statement to a given processor. Algorithms for allocation compatible with this approach have also been designed and implemented in PASCAL. These algorithms are iterative in nature, whereas those mentioned previously are highly recursive.

It is not difficult to see that each of the two approaches outlined in this paper produces allocations that are, in general, not optimal. An analysis of the algorithms developed to implement the first approach mentioned reveals that a bound on the worst case time complexity is $O(N^2)$ where N is the number of statements in the program. It can also be shown that $O(N^2)$ behavior cannot be achieved, since not all of the pathologically difficult conditions can arise at once. In order to obtain the obvious improvement (over the optimal case) in the time required to produce an allocation, this approach sacrifices optimality. Not all of the inherent concurrency is exposed, so it is not possible to use all of this concurrency in determining the processor assignment.

The second approach also provides a way of obtaining processor allocations quickly. The worst case complexity of this algorithm (as it has been implemented) is also bounded by $O(N^2)$, though we believe the average case will exhibit O(N log N) performance. It is evident that these algorithms, too, restrict the inherent parallelism in constructing the data dependency graph. This (potentially) adds some sequential restrictions, so that the allocations produced cannot, in general, be optimal.

Each of these implemented algorithms has its advantages and disadvantages. On small programs, with varying numbers of processors in the simulated system, each appears to produce very good results. The process of gathering more statistical data is, of course, an ongoing one. Research is also proceeding on the allocation problem in the context of an expanded set of linguistic constructs.

IV. References

1. J.E Rumbaugh, "A Data Flow Multiprocessor", IEEE Trans. Computers, Vol. C-26, No. 2, Feb. 1977, pp. 138-146.

2. I. Watson and J. Gurd, "A Prototype Data Flow Computer with Token Labeling", AFIPS Conf. Proc., 1979 NCC, New York, June 1979, pp. 623-628.

3. A. Davis, "A Data Flow Evaluation System Based on the Concept of Recursive Locality", AFIPS Conf. Proc., Vol. 48, 1979 NCC, New York, June 1979, pp. 1079-1086.

4. R.M. Keller, G. Lindstrom, and S.S. Patil, "A Loosely-Coupled Applicative Multiprocessing System", AFIPS Conf. Proc., Vol. 48, 1979 NCC, New York, June 1979, pp. 613622.

5. J.B. Dennis and D.P. Misunas, "A Preliminary Architecture for a Basic Data Flow Processor", Proc. Second Annual Symp. Computer Architecture, Houston, Texas, Jan. 1975, pp. 126132.

6. A. Plas, D. Compte, O. Gelly, and J.C. Syre, "LAU System Architecture: A Parallel Data Driven Processor Based on Single Assignment", Proc. 1976 Int. Conf. On Parallel Processing, P.H. Enslow, ed., August 1976, pp. 293-303.

7. J.B. Dennis, "Data Flow Supercomputers", Computer, Vol. 13, No. 11, Nov. 1980, pp. 48-56.

8. Arvind, K.P. Gostelow, and W. Plouffe, "An Asynchronous Programming Language and Computing Machine", Department of Information and Computer Science, TR 114a, University of California at Irvine, Dec. 1978.

9. W.B. Ackerman, "Data Flow Languages", AFIPS Conf. Proc., Vol. 48, 1979 NCC, New York, June 1979.

10. J. McGraw, "Data Flow Computing: Software Development", IEEE Trans. Computers, Vol. C-29, No. 12, Dec. 1980, pp.1095-1103.

11. G. Darrieu and J.C. Syre, "Extension of the LAU System: Global Specification of Synchronization in a Data Driven Language", Proc. Workshop on Data Driven Languages and Machines, J.C. Syre, ed., Toulouse, France, Feb. 1979.

12. K. Boekelheide, "A High Level, Graphical, Data Driven Language", Proc. Workshop on Data Driven Languages and Machines, J.C. Syre, ed., Toulouse, France, Feb. 1979.

13. A. Davis, "DDNs - A Low Level Programming Schema for Fully Distributed Systems", Proc Workshop on Data Driven Languages and Machines, J.C. Syre, ed., Toulouse, France, Feb. 1979.

DATAFLOW APPROACH TO DISCRETE SIMULATION

Bharadwaj Jayaraman

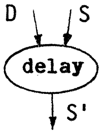
Department of Computer Science
University of North Carolina
Chapel Hill, North Carolina 27514

Summary

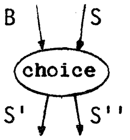
Discrete simulation is the technique of simulating the dynamic behavior of a system at discrete points in time. Languages for discrete simulation, such as GPSS, SIMULA, etc., are based on a sequential and centralized scheduler of events; however, greater concurrency can be achieved by a distributed approach, as proposed in [1], [2] and [4]. In this paper, we explore this idea in the context of a dataflow model [3] for two main reasons: 1) The interconnection and flow of entities in a simulation model, such as in GPSS, closely resemble the flow of streams between operators in a dataflow graph. 2) Dataflow models exploit both pipelined and horizontal concurrency; hence, unrelated or concurrent events can be executed asynchronously and concurrently

In our dataflow simulation model, each data value is tagged with a positive integer, called a time tag. A stream of such tagged data values always has time tags in monotonically increasing or chronological order. Streams may be finite or infinite (As in most data-flow models, the size of an input stream to any operator is assumed to be unbounded). The time tag of a data value represents the local time of some temporary entity in the simulation model. The processing of a data value by an operator is called an event and begins at the local time of the operator. The completion of an event can, but need not always, increase the local time of an operator. The concept of local time is introduced since there is no global clock and no centralized scheduler of events in our dataflow simulation model.

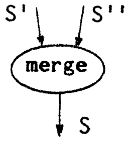
We present some primitive operators for dataflow simulation, and discuss the problem of correct simulation of events. The choice of operators is motivated by a need to consider the dynamic aspects of simulation, rather the functional properties of the entities. Since the dynamic behavior of our dataflow simulation model is determined mainly by the flow of streams, the operators of interest are various functions on streams. Typical operators from this set are:



If D is an untagged stream of nonnegative integers (d_1, d_2 ...) and S is a tagged stream (<v_1,t_1>, <v_2, t_2>,...) then S' is a tagged stream (<v_1,t'_1>, <v_2,t'_2>,...) where $t'_i = d_i + \max(t'_{i-1},t_i)$, and $t'_0 = 0$.



B is a untagged stream of bit values, and S, S', S'' are tagged streams. If the first bit value in B is 1 then the first data value in S is gated out to S' else the first data value in S is gated out to S''.



S is the result of sorting S' and S'' based upon their time tags.

We assume two operators, **source** and **sink**, for creating and destroying tagged streams respectively. The **delay** operator models a single-server queueing system, in which the stream D represents the service times for the objects in S. The **choice** operator is used for splitting up a stream; its control stream B will usually be generated by some probability distribution. The **merge** operator produces an output stream that is in chronological order; hence, it needs a tagged data value on both its inputs before it produces an output. As a consequence, it may be assumed that the **merge**

0190-3918/81/0000/0158$00.75 © 1981 IEEE

operator is determinate, i.e. will produce the same output stream given the same pair of input streams.

There are basically two types of dataflow simulation graphs: acyclic (figure 1) and cyclic (figures 2 and 3). Two types of cyclic graphs may also be distinguished: simple (figure 2) and shared (figure 3), depending on whether or not a **merge** operator is shared between two cycles. Cyclic graphs represent the notion of feedback in the simulation model and therefore are the more interesting case. Deadlock occurs in cyclic graphs due to a circular dependency between the output of a **merge** operator and its input. However, using information from other **merge** and **delay** operators in a cycle, it is possible to break deadlocks in a distributed fashion. In comparison, acyclic graphs do not require any additional mechanism for their correct operation.

We now summarize a method for breaking deadlocks in simple cycles. Four phases may be identified in the execution of a simple cycle: TEST, START, EXECUTE, and RE-TEST. During the TEST phase, a test message is sent around the cycle, by a pre-determined **merge** operator in the cycle, polling information from each **delay** and **merge** operator, in order to determine which **merge** operator must break the deadlock. In the START phase, the **merge** operator chosen to break the deadlock is sent a start message. The EXECUTE phase represents operation of the cycle after the deadlock has been broken. Special exit messages are sent around the cycle, by each **choice** operator in the cycle, to determine when the last data value leaves the cycle, i.e. to detect the recurrence of deadlock. The RE-TEST phase begins when some **choice** operator detects the recurrence of deadlock and sends the pre-determined **merge** of the cycle a re-test message. The receipt of this message re-initiates the TEST phase all over again.

Assuming $M_1 \ldots M_n$ are n **merge** operators in the cycle, d_i is the composite delay of all **delay** operators between M_i and M_{i+1}, loc_i is the local time of this composite **delay** operator, and t_i is the time tag of the first data value of the input arc of M_i that is not in the cycle, then the **merge** operator M_j chosen to break the deadlock is such that $S_{j,n}$ has the minimum value over all $S_{1,n} \ldots S_{n,n}$, where

$$S_{j,k} = \text{if } k < j \text{ then } t_j \text{ else } d_k + \max(loc_k, S_{j,k-1})$$

The deadlock is broken by sending the data value t_j along the output of M_j.

An extension of the above method can be used for breaking deadlocks in shared cycles, but is omitted here due to shortage of space. Work is in progress in formalizing the algorithm for this extension.

References

[1] R.E. Bryant, Simulation on a Distributed System, Computation Structures Group, MIT, Memo 182, (July, 1979).

[2] K.M. Chandy and J. Misra, Distributed Simulation: a case Study in Design and Verification of Distributed Systems, IEEE Transactions on Software Engineering, (September 1979), pp 440-452.

[3] J.B. Dennis. First version of a dataflow procedure language, In G. Goos and J. Hartmanis (eds.), Lecture Notes in Computer Science, Springer-Verlag, 1974, pp 362-376.

[4] J.D. Peacock, J.W. Wong, and E. Manning, Distributed Simulation Using a Network of Microcomputers, Computer Networks, (February, 1979), pp 44-55.

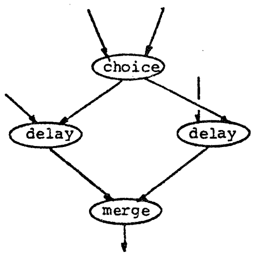


Figure 1

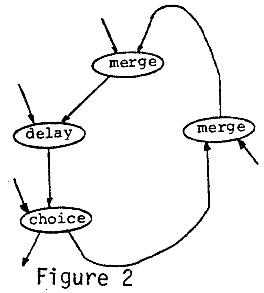


Figure 2

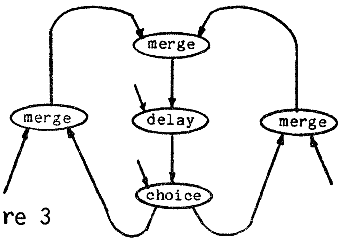


Figure 3

ARCHITECTURE OF A MULTIPROCESSOR USING DATA FLOW AT A PROGRAM BLOCK LEVEL

Marie-Paule LECOUFFE
E.R.A. CNRS 771
U.E.R. d'I.E.E.A. - Informatique
Université de LILLE I
F. 59655 VILLENEUVE D'ASCQ CEDEX

Summary

Data flow architectures bring a great contribution about the parallelism exploitation because they are able to detect, at execution time, instructions which are executable concurrently. However, most of the time, parallelism is exploited at program instruction level [1-3]. That implies a considerable flow of communication in the system because instructions, and sometimes their operands, are communicated separatly.

To minimize the flow of information, it may seem useful to exploit the parallelism at the level of a set of instructions, by grouping the objects used by these instructions and these instructions into an indivisible set that is called a "block". The communication between such sets is reduced to input and output parameters. So, a block forms a module which contains all the information which are necessary for its execution.

In this paper is described MAUD[(a)], a system based on the subdivision of programs into blocks and a data driven execution [4]. A program for MAUD is a set of blocks. Blocks are composed of a set of instructions and of the objects they use. A block can be viewed as a generalized primitive applied to input objects (*I-objects*) which are calculated by other blocks, and providing output objects (*O-objects*) which will be used as *I-objects* by other blocks. Blocks are built before the processing by the system.

Communication between blocks is conducted solely by means of *I-objects* and *O-objects*. A *communication name* is associated to *I-objects* and *O-objects* : it does not point out an explicit memory cell ; it is used only to point out values.

The single assignement rule is applied to the *communication names* at the block level. It allows the natural expression of dependencies existing between the blocks of a program, and therefore the expression of parallelism at execution time. Moreover, it allows the use of a data flow control for the execution of a program : a block is ready for execution when all its *I-objects*, which are *O-objects* of other blocks, have been propagated to this block. Such a block is an *executable block*. If one or several of its *I-objects* have not been assigned values, the block is called a *waiting block*. *Executable blocks* may be executed concurrently.

Two special operations used during block execution have been defined : (i) an *execblock* operation, which allows the execution of a block, a model of this block existing in a library. It has certain similarities to a procedure call (it is an *execution request* for a block which must exist in the library) and to a FORK operation (the execution of the requested block may run concurrently with the execution of the block which made the request) ; (ii) a *wait* operation which allows the calling block to wait *O-objects* calculated by the *execblock* operations. The execution of the block which performs a *wait* operation is then suspended, and this block is transformed into a *waiting block*, waiting for the objects which appeared in the *wait* operation. Thus, the execution processor becomes free.

The use of *execblock* and *wait* operations gives a dynamic characteristic to the execution of a program because of the addition of a number of blocks which are executable concurrently during the program execution. An example of the utilization of these operations can be found in [5].

The multiprocessor is composed of :

- *a set of execution processors* P_i whose function is the execution of a block ; each processor has a local memory and is able to execute a block in an autonomous way. The blocks are not dedicated to the processors. As soon as a processor is idle because it has just executed a *wait* operation or because the block execution is over it searches a new executable block, if there is one left.
- *an updating processor*, UPD, which updates the *waiting blocks* with *O-objects* produced by the execution processors at the end of block execution, and which finds out the *waiting blocks* with all the *I-objects* assigned in order to transform them into *executable blocks*.
- *a builder processor*, BUILDER, whose function is to build a block using the *execution requests* produced by the execution processors when they perform *execblock* operations. It manages the library of blocks.

There is no direct communication path between the processors. The *waiting blocks* and the *O-objects* are sent to the UPD processor, and the *execution requests* to the BUILDER processor. But they

(a) : MAUD : Machine d'Assignation Unique Dynamique.

0190-3918/81/0000/0160$00.75 © 1981 IEEE

are sent through shared memories :

- an *A-memory*, which holds the *waiting blocks* ;
- an *S-memory*, which holds the *O-objects* produced by the execution processors at the end of the block execution
- a *D-memory*, which holds *the execution requests* produced by the execution processors when they perform an *execblock* operation.
- an *X-memory* holds the *executable blocks*.

A functional description of MAUD is shown in Fig. 1. An example of the execution of a program in MAUD can be found in [6].

An implementation of MAUD has been studied. The processors are realized with conventional microprocessors, except the UPD processor which is a very specialized one, because it must be very fast while not necessarily very powerful, and it must be able to have associative accesses to A-memory and S-memory.

Memories have two functions : storage of the various objects of MAUD, and communication. For the realization, it has been chosen to use a unique memory shared by all the processors : it is a ring of circulating memory. That allows simultaneous accesses for reading and writing by all the processors, and it is easy to have associative access for the UPD processor. The ring is divided into logical sectors of the same size called slots. Every slot can hold any kind of objects, but only one at a time ; *execution requests* and *O-objects* circulate temporarily (no more than one lap) ; *waiting blocks* and *executable blocks* are kept circulating in the ring until the former become executable blocks and the latter are picked out from the ring by a free processor. A MANAGER processor is necessary to regulate the load of the ring, i.e. to put some blocks temporarily out of the ring when the number of empty slots gets too small, and to put them back inside when the number of empty slots is increasing. In fact, no new processor is needed, this function may be realized by the UPD processor or the BUILDER processor.

In order to justify this choice, a simulation of the system with the above characteristics of hardware implementation has been done. The obtained results for a simplified version of the system makes it clear that the gain in the processing speed is important compared to a conventional monoprocessor system, if the block execution time is not too short compared to the duration of a complete lap of the ring. Yet, this condition is not necessary because it is possible to have dynamic reconfigurations of the ring allowing the reduction of the access time [7].

References

[1] J.B. Dennis, *"The varieties of data flow computers"*, 1st International Conference on distributed computing systems (Oct, 79).

[2] A. Plas and al, *" LAU system architecture : a parallel data driven processor based on single assignment"*, 1976 International Conference on parallel processing (Aug, 76).

[3] P.C. Treleaven and al, *"a concurrent architecture and a ring-based implementation"*, 6th International symposium on computer architecture (April, 79).

[4] M.P. Lecouffe, *"Etude et définition d'un modèle de machine parallèle dirigée par les données"*, Thèse de 3ème cycle, Université de Lille I, (July, 79).

[5] M.P. Lecouffe, *"MAUD : a dynamic single assignment system"*, Computers and Digital Techniques, (April, 79), Vol. 2, n° 2.

[6] M.P. Lecouffe, *"Dynamic processing with single assignment at a program block level"*, Workshop on data driven languages and machines, (Feb, 79).

[7] B. Petitprez, *"A flexible circulating memory for communication in a multiprocessor"*, Euromicro Congress, (Sept, 80).

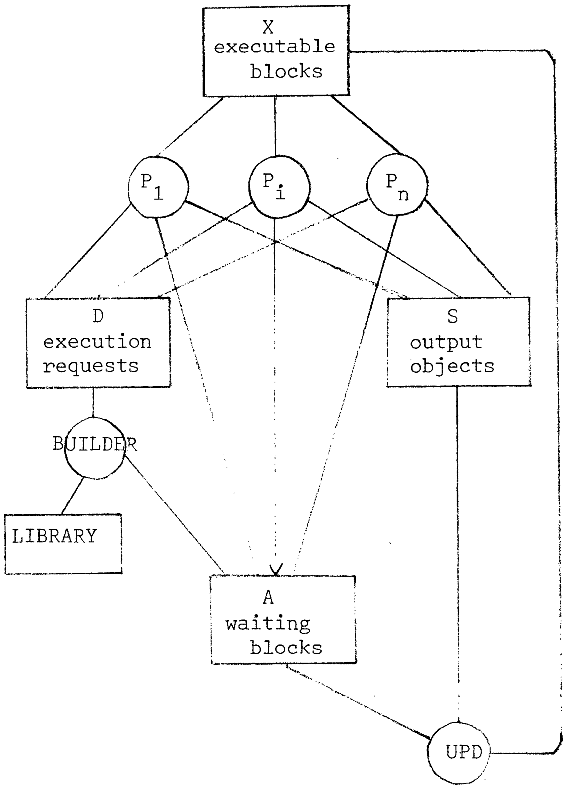


Fig. 1 : MAUD : Functional description.

HIGH LEVEL SPECIFICATION OF RESOURCE SHARING

Dennis W. Leinbaugh
Computer and Information Science Research Center
The Ohio State University, Columbus, Ohio, 43210

Summary

A high level specification language is described making it possible to very concisely specify the orderly sharing of a protected resource [1]. The rules and policies dictating resource usage are specified separately and clearly making it easy to write, understand, and change them. Since the specifications themselves are enforced, no errors in resource sharing are introduced in programming the enforcement of them.

Many schemes have been proposed and developed to aid in resource sharing. Hoare's monitors and Hewitt's serializers were designed primarily to enforce cooperation among users sharing resources. These schemes provide primitives and language structures which make it relatively easy to write code to enforce the necessary rules and desired policies upon resource sharing.

This work describes how to directly specify the resource sharing rules needed and policies wanted. The code to enforce these rules and policies can then be automatically generated from the high level specification provided. The advantages are clear. Since the rules and policies are specified directly, it is known exactly what they are and that they are enforced.

Ramamritham and Keller [2] concurrently with and independent of this work attacked the same problem. Their specification language is at a different level. State variables are conceptually different and the implementation schemes are entirely different for the two systems.

Request messages for a protected resource are sent to its scheduling module. This module uses the high level specifications provided to determine what requests and when requests are sent to for service. These specifications are:

- description of the requests,
- resource constraints,
- ordering policy,
- postponement policy, and
- expedite policy.

The description of the requests defines the fields in request messages that will be used by the resource scheduler to aid in scheduling them. The resource modules that provide the service for each type of request are also specified as well as the updates which the performance of these requests cause to the state variables.

The resource constraints consist of defining those states in which the resource continues to correctly service requests. These states are described in terms of the values of state variables and the requests that can simultaneously be serviced by the resource. A request is acceptable to the resource if its inclusion for service would result in a state described by the resource constraints. State variables are defined local to the resource scheduler and reflect the actual state of the resource.

The ordering policies specify the usual policy used to decide what request should receive service next. Among the requests acceptable to the resource, the ordering policy determines which actually begins service. In case of ties, the older request is granted service. The ordering policy is specified in terms of priorities between request types, ordering based upon some value in the request, or some other standard ordering scheme (e.g., elevator algorithm).

The main purpose of the ordering policies is to achieve efficiency in resource use or efficiency in the processes which use the resource. Efficiency considerations alone, however, can lead to very poor service or no service for some requests. The postponement and expedite policies are used to modify the ordering policies to avoid extremely poor service.

The postponement policies specify under what conditions newly arrived requests are not to be considered for selection even if they would be acceptable to the resource and no other waiting requests are acceptable. If, however, there are no waiting requests then postponed requests may be selected for service. A request can only be postponed when it initially arrives and then only until the postponed condition for it becomes false. The postponement conditions may involve the current resource state and a consideration of other waiting requests.

The expedite policies specify under what conditions the ordering policies are to be violated and a non-postponed request is selected to be the next request in line for service. No other requests are allowed ahead of a request chosen by expedite.

The postponement policy should be used to hold back requests which might otherwise cause starvation of any of several requests waiting for service. The expedite policy should be used to identify a request being starved and make it next for service.

Figure 1 illustrates the scheduling strategy. A process requests service by sending a request message to the Scheduling Module. The Scheduling Module implements the resource sharing specifictions, forwarding the request to the Protected Resource Module when it is to be performed. When service is complete, the process receives a response message. Requests can only reside as postponed requests, ordered requests, expedited requests, or requests being serviced. The un-

0190-3918/81/0000/0162$00.75 © 1981 IEEE

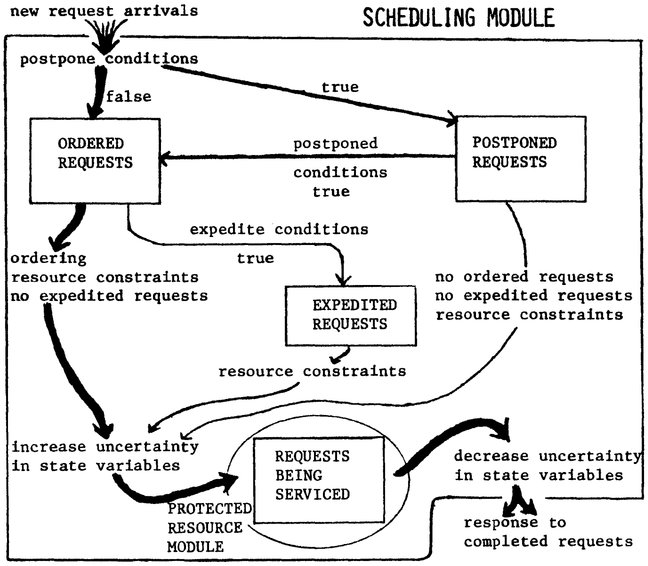


Figure 1: Overall Scheduling Strategy

certainty in state variable values increases when a request begins service and decreases when service is complete. This handling of state variables is faithful to what the scheduling module can know of actions of the resource module allowing for natural specifications of resource constraints.

Figure 2 is a specification of the classical producer/consumer problem. The resource can hold up to 10 items. An insert request message to the insert routine adds another item into the resource and a remove request message removes an item from it. At most one insert request and one remove request can be serviced at the same time. The maximum number of items that can be placed in the resource is 10 and the minimum number is 0. To use the constraints, the preconditions for each type of request are derived. For the case of an insert request, the preconditions are less than 10 items already saved and no insert request receiving service. The number of items is kept track of in the scheduling module through the use of the local state variable #items. The PROCESSING clause indicates that during service of an insert request, the number of items is increased by 1 and during service of a remove request, the number of items decreases by 1. There is, however, uncertainty as to exactly when these changes occur. #items is kept as a range of possible values. For example, if there were 9 items and both a remove and insert request were receiving service, #items is the range [8,10]. If the remove request completes first the range becomes [8,9] and when the insert request subsequently completes the range becomes [9,9].

```
DECLARE STATE VARIABLES #items INITIALLY 0

REQUEST DECLARATIONS
     REQUEST FIELDS  type  CHARACTER(1)
                     item  CHARACTER(99)

          insert  HAS  type = 'I'
          remove  HAS  type = 'R'

PROCESSING
     insert PROCESSED BY insert-routine
            UPON SERVICE #items := #items + 1

     remove PROCESSED BY remove-routine
            UPON SERVICE #items := #items - 1

RESOURCE CONSTRAINTS
     insert.ACTIVE ≤ 1 AND remove.ACTIVE ≤ 1
     AND    0 ≤ #items  AND  #items ≤ 10
```

Figure 2. A Producer/Consumer Problem

Figure 3 is a high level specification for sharing a moving head disk. ORDERING specifies both a primary and secondary ordering policy. If there is more than one request for a disk address, then the write requests are done before the read requests for that address. The elevator algorithm insures that no addresses (at the ends) are ignored. The only way a request can wait forever is if new requests for the same address keep receiving service. The POSTPONE prevents this by not allowing these newly arrived requests to be considered for service until the disk has moved off the address they are requesting (THISREQUEST.addr = ACTIVE.addr).

```
REQUEST DECLARATIONS
     REQUEST FIELDS  type  CHARACTER(1)
                     addr  CHARACTER(6)
                     data  CHARACTER(505)

          read HAS   type = 'R'
         write HAS   type = 'W'

PROCESSING
     PROCESSED BY  disk-driver-routine

RESOURCE CONSTRAINTS
     read.ACTIVE + write.ACTIVE ≤ 1

ORDERING PRIMARY BY ELEVATOR ON addr
         SECONDARY write BEFORE read

POSTPONE read IF THISREQUEST.addr = ACTIVE.addr
        write IF THISREQUEST.addr = ACTIVE.addr
```

Figure 3: Moving-Head Disk Scheduler

References

[1] D. W. Leinbaugh, "High Level Specification and Implementation of Resource Sharing," The Ohio State University, (Feb., 1981), Technical Reprot OSU-CISRC-TR-81-3.

[2] K. Ramamritham and R. M. Keller, "Specification and Synthesis of Synchronizers," Proc. 1980 International Conference on Parallel Processing, (Aug., 1980), pp. 311-321.

Exploitation of Concurrency by Virtual Elimination of Branch Instructions

N. Magid
Dataproducts Corporation
Wallingford, CT 06492

G. Tjaden
Cox Cable Communications
Atlanta, GA 30346

H. Messinger
Illinois Institute of Technology
Chicago, IL 60616

Summary

This paper introduces a technique for the virtual elimination of conditional branch instructions during program execution. The technique, called Multiple Path Exploration (MPE), aims at increasing the potential concurrency between program instructions by, automatically and dynamically, removing procedural dependencies.

There are basically two types of dependencies between instructions: Data Dependency, when one instruction requires data from a previous instruction; and Procedural Dependency, due to the specification of the instructions sequence. There is a procedural dependency between a branch instruction and the instructions following it in sequence. Conditional branch instructions cause a wait until the condition is resolved before the next instruction in the sequence is determined, thus imposing severe limitations on the attempts to detect and exploit concurrency.

In order to eliminate the procedural dependency caused by the presence of a conditional branch instruction in a program, the execution must proceed simultaneously down the two possible paths emanating from the branch. To bypass x conditional branch instructions, as many as 2^x paths must be processed simultaneously.

Instead of bypassing all conditional branch instructions of a program simultaneously, only a subset consisting of a fixed number, m, of branches may be bypassed at any given time. Out of 2^m paths, only one path will remain valid, while all the others may be discarded. Another set of (2^m) paths, generated from the valid path are explored next. This process continues until the program is completely executed.

Branch instructions are grouped into sets. Each set represents a Branch Level (Fig. 1). Each path is uniquely identified by a Path Code. A Branch Code identifies each instruction with at least one path. The concepts of Branch Level, Path Code, and Branch Code provide tools to automate the process of generating and discarding of branch paths dynamically during program execution.

Further performance improvement can be achieved if the instructions of each path are not executed in a strictly sequential order. This becomes possible if there is a mechanism associated with every path, which detects data independent instructions. The Ordering Matrix technique is suitable for the detection of data independent instructions, especially in the absence of branch instructions [1], [2]. The Ordering Matrix (M) for a sequence of N instructions is an N x N Boolean matrix such that:

$$M_{ij} = \begin{cases} 1 & \text{Iff instructions } I_i \text{ \& } I_j \text{ are dependent.} \\ 0 & \text{Otherwise} \end{cases}$$

The presence of branch instructions in general, and backward branches in particular, have complicated algorithms to detect data independence and limited the amount of concurrency detected [2]. Their absence within each path enables the detection of more concurrency using less complex algorithms [1].

Foster and Rieman [3] found that a speed-up in program execution by a factor of 51 may theoretically be achieved if all conditional branches are bypassed. Using the MPE technique, the speed-up factor is expected to be as shown in Fig. 2 as a function of the number of branch levels m (and consequentially the number of streams N) [1]. A speed-up factor of more than 5 may be achieved for the case of m = 4, where 16 paths are processed simultaneously by different instruction streams.

The MPE technique may be implemented using a Multiple Instruction Stream, Multiple Data Stream Organization as shown in Fig. 3. The private data memory is used to enable the discarding of invalid paths. An execution speed of 15 MIPS may be obtainable. The architecture of Fig. 3 is discussed in detail in reference [1].

The proliferation of VLSI and microcomputer technology is expected to make the implementation of such a highly parallel system organization cost-effective in the future.

References

[1] N.F. Magid, High Speed Computer Systems As A Result of Concurrent Execution of Sequential Instructions, Ph.D. dissertation, Illinois Institute of Technology, Chicago, Illinois, (1980).

[2] G.S. Tjaden and M.J. Flynn, "Representation of Concurrency With Ordering Matrices", IEEE Trans. on Computers (August 1973), pp. 752-761.

[3] E.M. Riseman and C.C. Foster, "The Inhibition of Potential Parallelism by Conditional Jumps", IEEE Trans. of Computers, Vol. C-21, (Dec. 1972), pp. 1405-1411.

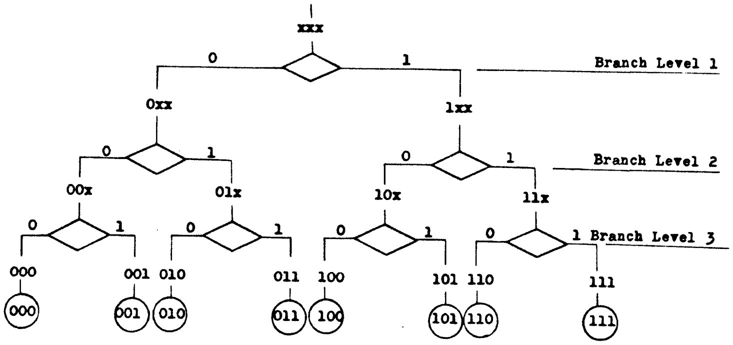


Figure 1. Branch Tree

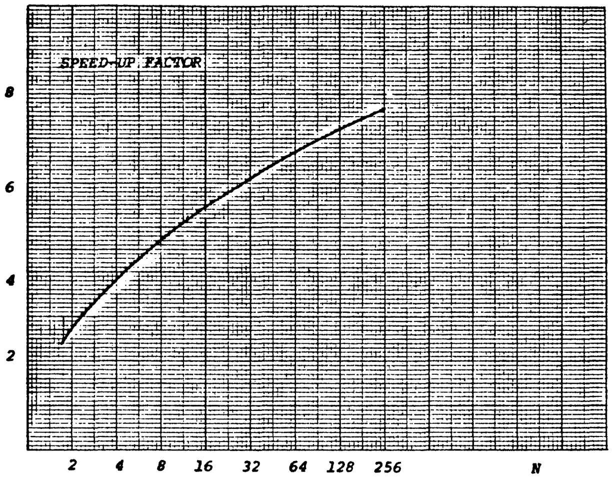


Figure 2. Speed-up Factor vs. Number of Streams

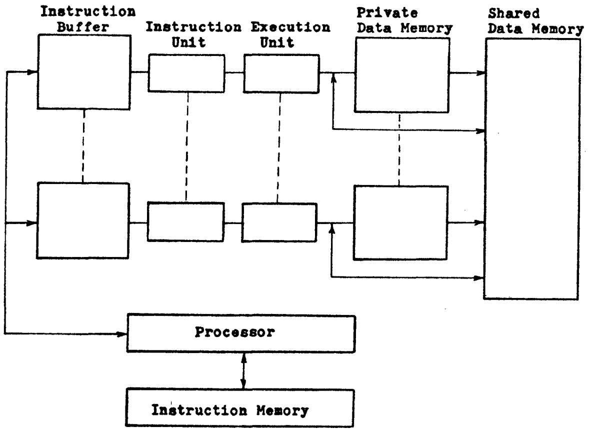


Figure 3. System Block Diagram

EXPERIMENT IN PARALLEL PROCESSING A LARGE SCIENTIFIC CODE

Ingrid Y. Bucher, Bill L. Buzbee, and
Paul O. Frederickson

Computer Research and Applications Group
Los Alamos National Laboratory
Los Alamos, New Mexico 87545

Summary

We wish to report results of a successful initial experiment in our study of the usefulness of multiple processor architectures for large scientific computations.

It is believed that a hundredfold increase in computational speed will be required over the next decade in order to meet a variety of scientific needs[1]. The prospects of speeding up a single processor mainframe by a factor greater than ten beyond the fastest machines available today, seem rather dim. It follows that parallel processing is necessary in order to meet the needs of the scientific community.

Using current technology, interprocessor communication, via either direct communication lines or common memory, is a significant factor to be considered in the design of an algorithm to fit on a parallel architecture. Our guiding philosophy is to divide current computational problems into relatively large tasks with a high degree of independence, thus minimizing the need for interprocessor communication. To avoid common memory contention we are looking at parallel architectures in which each processor is equipped with a reasonable amount of private memory. In that framework we wish to consider the usefulness of a variety of interconnection schemes.

Our initial experiment involved formulating a particle-in-cell simulation of a plasma[2] for a simple star graph architecture with a UNIVAC 1110 at the hub P_o of the star and up to eight Floating Point System 120B array processors at the other vertices P_i. Each of the nodes P_i was equipped with at least 48k words of memory, but there was no fast access common memory available in this system.

Figure 1a illustrates the main computations within one time-step of our model algorithm as carried out on a monoprocessor. Our adaptation of this algorithm to fit on the architecture described above is shown in Figure 1b. As indicated in the figure, the potential ϕ is computed for n cells from the charge distribution C by processor P_o. n values of ϕ are subsequently transmitted to processors P_i, i=1,...,8. The computation of the field E from the potential ϕ is carried out by each of the processors P_i in parallel, in order to reduce the data transfer from 2n to n items (for a 2-dimensional problem). Each processor P_i then moves its share of m_i particles through the electromagnetic field, a step which constitutes the major contribution to the computational process, and computes their contribution C_i to the total charge distribution C. As final steps, n values of C_i have to be transmitted from each processor P_i to processor P_o where they are summed to yield the charge distribution C for the next time-step.

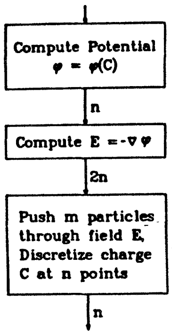

Compute Potential
$\varphi = \varphi(C)$
n
Compute E = $-\nabla\varphi$
2n
Push m particles through field E, Discretize charge C at n points
n

Fig. 1a

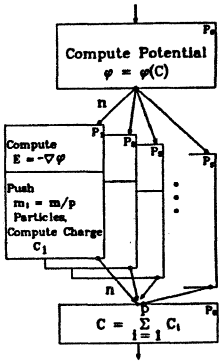


Fig. 1b

The multiprocessor used in our experiment is located at the Naval Ocean Systems Center in San Diego. In a fairly typical run, we moved m = 32,400 particles, distributed evenly over six array processors, in an n=18x18 size grid. Each time step required 0.54 s. Of this 0.265 s was spent solving Poisson's equation in the host at a rate of 0.2 MFLOPS/s. Each array processor P_i pushed 5400 particles in 0.13 s at a rate of 5.7 MFLOPS/s[3]. The remaining time, 0.145 s, was spent in initiating data transfers to and from the array processors and transmitting the data, most of it being system overhead.

0190-3918/81/0000/0166$00.75 © 1981 IEEE

It is fairly obvious that the time spent on solving Poisson's equation could have been reduced to less than 0.01 s by moving the process from the relatively slow host to the array processors. Our experience shows that to speed up interprocessor communications an operating system that allows for parallel data transfers with a minimum of system overhead, or efficient access to common memory, is highly desirable.

We conclude that in spite of the limitations of the system used, significant speedups via parallel computation are achievable for particle-in-cell plasma simulation and related problems.

We thank the staff of the simulation facility of the Naval Ocean Systems Center for their assistance in using their system, particularly Bob Dukelow, John Mayr, and Ron Dahlseid.

References:

(1)
B.L. Buzbee, W.J. Worlton, G. Michael, G. Rodrigue, DOE Research in Utilization of High Performance Computers, Los Alamos Scientific Laboratory Report, LA-8609-MS, December 1980.

(2)
R.L. Morse, C.W. Nielson, One-, Two-, and Three-Dimensional Numerical Simulation of Two Beam Plasmas, Phys. Rev. Letters 23, 1087 (1969).

(3)
I.Y. Bucher, P.O. Frederickson, Experience with a Multiprocessor Based on Eight FPS 120B Array Processors, Los Alamos National Laboratory Report LA-UR-81-1082, March 1981.

ITERATORS AND CONCURRENCY

A. T. Berztiss
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

We propose a modest parallel execution facility for essentially sequential programs, particularly appropriate for very small computers. It consists of three approaches. First, iterators are regarded as a major programming tool. We discuss parallel composition of iterators, and use of iterators as a means for buffered access to elements of composite data structures. Second, Dijkstra's guarded command set is given a new interpretation: all actions for which guards are true may be performed in parallel. Third, we consider the partitioning of iteration sequences into segments that may then be executed in parallel.

Empirical studies of computer programs have shown that most processing time is spent in executing loops, and that most loops are concerned with providing orderly access to elements of composite data structures [1] - [3]. Our aim here is present mechanisms for achieving limited parallelism in the execution of for-loops, and we do so in the context of data abstraction.

We shall use iterators coupled to for-loops. Iterators are provided in CLU [4] and Alphard [5], and their purpose is to deliver the elements of a composite data structure in the particular sequence determined by the specification of the iterator. This achieves separation of the traversal of a data structure from the computations carried out on the objects delivered by the iterator in the course of the traversal. Iterators can therefore be made part of an abstract data type. We have introduced controlled iteration [6]. This generalization in the usage of iterators enables the same for loop to invoke more than one iterator, and the iterators coupled to a given for loop to be synchronized. The form of a for loop with controlled iteration is

```
for J1,J2,...,Jn loop
  loop body;
end loop;
  loop tail;
end for;
```

This construct is described in [6], and familiarity with this reference is being assumed.

Because iterators can appear only in a specific context, namely the for loop, a single stack suffices for runtime control (as long as the iterators are nonrecursive and do not invoke other iterators): On reaching a for statement, create a vector of n activation records, one for each of the J1, J2, ..., Jn, where it is assumed that an activation record contains space for all temporary locations needed by the corresponding iterator. The vector of activation records is treated as a single unit until it is discarded as a single unit on reaching the end for. Iterators are a special type of coroutines, but they have advantages over general coroutines. This has been discussed in [6]. Here we note that it is the confining of iterators to for loops that permits look ahead and buffering in the implementation of an iterator.

As regards difficulty of program proofs, our generalized for loop occupies an intermediate position between a conventional for loop and a while loop. Because one can interpret iterators as coroutines, our generalized for loop can be translated in a mechanical manner to a while loop that contains coroutine calls. At the very worst, then, the proof rules of the while loop can be put to use. Most instances, however, advantage can be taken of the fact that the generalized for loop is rather closer to the conventional for loop than to the while loop.

The use of controlled iteration implies that some decisions are made within the loop body. Consequently the entire loop body may be an if statement. We now introduce a change of notation for the if statement:

```
   if  B1  then  S1;
elsif  B2  then  S2;
  ...  ...   ...
elsif  Bn  then  Sn;
   end if;
```

becomes

```
cond  B1:  S1 □
      B2:  S2 □
      ...  ...
      Bn:  Sn
end cond;
```

The latter has been made to resemble Dijkstra's guarded command set [7]. It then suggests new interpretations. When the conditional is interpreted as equivalent to the if statement we call it a sequential conditional. Dijkstra's interpretation is that any Si that corresponds to a true Bi may be executed, but this interpretation can be carried further: All Si for which guards Bi are true may be executed concurrently. The conditional then becomes a concurrent conditional with the following informal semantics:

a. All Bi are evaluated before the execution of any Si begins.

b. All Si following true Bi are executed fully, or, if any such Si contains an exit, up to the exit, and if an exit has been encountered, exit from the loop body takes place after all this has been done.

0190-3918/81/0000/0168$00.75 © 1981 IEEE

Example: Determine whether or not a given key is present in a linear list that is being simultaneously traversed from both ends. Here we can have concurrent advances in the list, and concurrent evaluation of the Boolean expressions (guards).

```
for X in FILE.UP
    Y in FILE.DOWN loop
    cond
        X.KEY = Y.KEY     : exit □
        X.KEY = GIVEN_KEY: exit □
        Y.KEY = GIVEN_KEY: exit
    end cond;
end loop;
    if X.KEY = GIVEN_KEY or Y.KEY = GIVEN_KEY then
        PUT("Given key is in the list");
    else
        PUT("Given key not found in the list");
    end if;
end for;
```

We now consider partitioned iteration sequences in the context of matrix multiplication: Matrix C is to receive the product of matrices A and B, and it is assumed that C has already been initiated to zeros. In conventional Ada syntax this can be written as follows:

```
for I in A'FIRST..A'LAST loop
    for K in A'FIRST(2)..A'LAST(2) loop
     for J in B'FIRST(2)..B'LAST(2) loop
            C(I,J):= C(I,J) + A(I,K)*B(K,J);
        end loop;
    end loop;
end loop;
```

Here matrix C is built up one row at a time. In building up a row in C, the corresponding row in A is traversed just once, but B is traversed in its entirety. What matters is that in generating a particular row of C only the one corresponding row of A is needed, i.e., the traversal of A can be partitioned into independent traversals of its rows. Consequently we now consider matrix A as a set of vectors (rows of the matrix). The matrix multiplication code is reformulated to make use of the separation of the matrix into a set of vectors to induce parallelism. It is the declaration of the data structure as a set before the for loop is entered that enables the system to recognize the opportunity for concurrency. The loop itself contains no indication to this effect.

```
for X in ROWSETA.TRA loop
    I:= C'FIRST(2);
    for controlled A in X.FORWARD,
        B in MATB.ROWWISE(ENDROW) loop
        C(X.ROWNO,I):= C(X.ROWNO,I) + A.VAL*B.VAL;
        I := I+1;
        if ENDROW then
            promote A;
            I:= C'FIRST(2);
        end if;
    end loop; end for;
end loop; end for;
```

Here we have three iterators: (i) Iterator TRA delivers a complete row of the matrix. It is understood that the object denoted by ROWSETA functions as a set of vectors. The guise of this matrix as a set permits parallelism. The body of the outer loop can be executed concurrently by as many processors as there are rows in the matrix. An attribute of the row delivered by TRA is the index of this row in reference to the matrix as a two-dimensional array (ROWNO). It establishes correspondence between the rows of the input matrix and of C. (ii) FORWARD delivers the elements of the row supplied by TRA. (iii) ROWWISE is associated with the second input matrix in its guise as a proper matrix (MATB). It delivers elements of MATB one by one in roworder. Parameter ENDROW associated with ROWWISE is normally false, but it becomes true for any pass through the loop in which an element that terminates a row in MATB is being accessed.

One problem that remains is the synchronization of components of several partitioned iteration sequences. Such is the case when matrices A and C are both regarded as sets of row vectors. Then it has to be ensured that the row in C generated using a particular row in A properly corresponds to this row in A (for example, that the row generated using the second row of A becomes in fact the second row of C). Iterators are used to enforce the required correspondence. If the same iterator ranges over two sets, then an ordering of the elements of the sets into sequences has to be assumed, and corresponding elements from the sequences are assigned to the same instance of execution of the loop body.

Note that our purpose has not been to solve general concurrency and synchronization problems. Nevertheless, as regards the execution of such programs as are currently executed on very small computers, use of our mechanisms can lead to substantial reduction in execution time by enabling the computational load to be spread over several processors.

References

[1] D.E. Knuth, "An Empirical Study of Fortran Programs", Software--Practice and Experience 1 (1971), pp. 105-133.

[2] T.W. Pratt, "Control Computations and the Design of Loop Control Structures", IEEE Trans. Software Eng. SE-4 (1978), pp. 81-89.

[3] D. Grune, "Statistics of Algol 68 Programs", SIGPLAN Notices (ACM) 14, 7 (July 1979), pp. 38-46.

[4] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction Mechanisms in CLU", Comm. ACM 20, 8 (Aug. 1977), pp. 564-576.

[5] M. Shaw, W. Wulf, and R. London, "Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators", Comm. ACM 20, 8 (Aug. 1977), pp. 553-563.

[6] A.T. Berztiss, "Data Abstraction, Controlled Iteration, and Communicating Processes," Proc. ACM Annual Conf., Nashville, TN, 1980, pp. 197-203.

[7] E.W. Dijkstra, "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs", Comm. ACM 18, 8 (Aug. 1975), pp. 453-457.

OPTIMAL PARALLEL ALGORITHMS FOR THE CONNECTED COMPONENT PROBLEM[(a)]

Francis Y. Chin

John Lam

I-Ngo Chen

Department of Computing Science
University of Alberta
Edmonton, Alberta T6G 2H1

Abstract -- In this paper, we study a parallel algorithm for computing the connected components of an undirected graph, using the Single Instruction Stream-Multiple Data Stream model. We assume that the processors have access to a common memory and that no memory or data alignment time penalties are incurred. We derive a general time bound for a parallel algorithm which uses K processors for finding the connected components of an undirected graph. In particular, an $O(\log^2 n)$ time bound can be achieved using only $K = n \lceil n/\log^2 n \rceil$ processors. This result is optimal in the sense that the speedup ratio is linear with the number of processors used. The algorithm can also be modified to solve a whole class of graph problems with the same time bound and fewer processors than previous parallel methods.

I. INTRODUCTION

The dramatic drop in the cost of computers encourages the use of parallel computers. Parallel computers are capable of performing several independent operations concurrently. In the following discussion, we assume that (1) processors share the same memory; (2) each processor can perform any arithmetic, Boolean or logical operations in one time unit, and all instructions executed in parallel are identical (Single Instruction Stream-Multiple Data Stream [7]); (3) simultaneous read operations on the same location are allowed, but not simultaneous write operations; (4) no memory or data alignment time penalties [15] are incurred.

Parallel algorithms for sorting and numerical applications have received substantial attention recently [1,9,11,17,23]. Much work has been done on the development of efficient parallel graph algorithms [5,6,8,10,12,14,18,19]. A parallel algorithm which uses n^2 processors to find the connected components of an undirected graph with n vertices in $O(\log^2 n)$ time was proposed in [10]. Recently, it has been shown by Hirschberg et al. [12] that an $O(\log^2 n)$ time bound can also be achieved using only $n \lceil n/\log n \rceil$ processors. In this paper, we present a modified version of this algorithm which requires $O(n^2/K + \log^2 n)$ time if only K processors are available. In particular, the $O(\log^2 n)$ time bound can be achieved with only $n \lceil n/\log^2 n \rceil$ processors. This modified algorithm is optimal in the sense that the speedup ratio [21] is linear with the number of processors used. Furthermore, we demonstrate that this algorithm can be used to solve a class of graph problems with the same time and processor bounds in a forthcoming paper [4].

Section 2 presents definitions used in this paper. Section 3 studies the modified algorithm and derives its time and processor bounds. Section 4 summarizes these results and discusses further research possibilities.

II. DEFINITIONS

An undirected graph $G = (V,E)$ consists of a finite, non-empty set V of n vertices and a set E of unordered pairs of vertices called edges. We represent G by its adjacency matrix A, which is an nxn symmetric Boolean matrix where $A(i,j) = 1$ if and only if $(i,j) \in E$. G is connected if there exists a path between every pair of distinct vertices in V. A connected component of G is a maximal connected subgraph of G.

If T_K is the time required by a parallel algorithm using $K \geq 1$ processors, the speedup ratio of the K-processor computation over the corresponding uniprocessor computation (taking time T_1) is defined as $S_K = T_1/T_K$.

Throughout the paper, log n denoted $\lceil \log_2 n \rceil$.

III. CONNECTED COMPONENTS OF AN UNDIRECTED GRAPH

Figure 1 shows the algorithm MOD.CONNECT for finding the connected components of an undirected graph. The actions of algorithm MOD.CONNECT can be described briefly as follows. Each vertex belongs to exactly one connected component. Array D is used to specify the connected component for each vertex, thus $D(i) = D(j)$ if and only if vertices i and j belong to the same component. Step 1 initializes the arrays D and Flag whose function will be discussed later. During the first iteration, step 2b selects the smallest numbered vertex among all the vertices incident upon vertex i and assigns it to C(i). Step 3 eliminates the isolated vertices. Steps

[(a)] This research was supported in part by Natural Science and Engineering Research Council Grant NSERC-A4319 and A7133.

0190-3918/81/0000/0170$00.75 © 1981 IEEE

Algorithm MOD. CONNECT

Input: The nxn adjacency matrix A for an undirected graph.

Output: The vector D of length n such that D(i) equals the smallest-numbered vertex in the connected component to which i belongs.

Comment: Each of the following steps is executed in parallel for all i, $0 \le i < n$ or for all $i \in S$. The assignments in the various steps are considered to be done simultaneously for all i.
The vector Flag of length n such that Flag(i) = 1 indicates vertex i is a current supervertex. Current supervertices are stored in set S.

```
1   for all i, 0 ≤i <n do comment:  Initialization
        D(i) <--i
        Flag(i)<--1
    do step 2 through 8 for log n iterations
    comment:  Uniform Smallest Incident Node
    Selection
2a  S<--{i|Flag(i) = 1} comment: D(i) = i for i∈S
2b  for all i∈S do
        C(i) <-- Min{J | A(i,j) = 1}
                 j∈S
                 if none then i
    comment:  Eliminate the isolated supervertices
3   for all i∈S do
        if C(i) = i then Flag(i) ←0
    comment:  Path Compression
4   for all i∈S do D(i) <--C(i)
5   for log n interations do
        for all i∈S do C(i) <--C(C(i))
6a  for all i∈S do D(i)<-- Min{C(i), D(C(i))}
6b  for all i, 0 ≤i <n do D(i) <--D(D(i))
    comment:  Clean Up (by column contraction)
7a  for all i∈S do
        for all j S s.t. j=D(j) do
            A(i,j) <--OR{A(i,k)  D(k)=j}
                        k∈S
7b  for all j∈S s.t. j=D(j) do
        for all i∈S s.t. i=D(i) do
             A(i,j) ←OR {A(k,j) |D(k)=i}
                       K∈S
7c  for all i∈S do A(i,i) <--0
8   for all i∈S do if D(i)≠ i then Flag (i)<--0
```

Figure 1 Algorithm MOD.CONNECT

4-6 perform path compression and merge vertices which are known to be in the same connected component into a single "supervertex". Steps 7 and 8 eliminate the merged vertices and store all the information about their edges into the supervertices. In succeeding iterations, S contains the indices of the supervertices and the whole process is repeated on the graph represented by the adjacency matrix A restricted on S. Supervertices are merged to form super-supervertices, and so on. This merging process is repeated log n times until each connected component is represented by a single vertex. Array D contains the information about which vertices are in the same component and step 6b updates D(i) for all i, i.e. updates the supervertex into which i is merged. The main difference between algorithm MOD.CONNECT and algorithm CONNECT in [12] is the introduction of the vector Flag of length n, the set S and the clean-up steps (steps 7 and 8) in algorithm MOD.CONNECT. Flag(i) = 1 indicates that vertex i is a supervertex. Flag(i) = 0 indicates that vertex i has been merged into a supervertex or is an isolated supervertex and should not be used in subsequent iterations. Thus only those vertices with Flag(i) = 1 (or in S) are involved in any given iteration.

In order to visualize how the algorithm works, an informal description of the set S and the arrays A,D, Flag and C during each iteration are given as follows:

"Flag" - a boolean vector of length n.
Flag (i) = 1 iff vertex i is a supervertex representing the group of vertices being merged to it. Vertex i is always the smallest numbered vertex of the supervertices.

"S" - contains the indices of the supervertices.

"D" - a vector of length n. D(i) specifies the supervertex into which vertex i is merged.

"C" - a vector of length n. C(i) specifies the smallest supervertex to which supervertex i is adjacent.

"A" - an n x n symetric boolean matrix. Usually we are only interested in the restricted A over the current S.
A(i.j) = 1 if and only if there is an edge connecting supervertex i and supervertex j.

Algorithm MOD.CONNECT is a modified version of the algorithm CONNECT given in [12], a more detailed proof for the correctness of the algorithm can be found there.

Since the number of flagged vertices (the number of elements in S) is reduced by a factor of at least two after each iteration, we shall show that by the technique of problem decomposition [13] the same time bound $O(\log^2 n)$ can still be achieved by using less than $n \lceil n/\log n \rceil$ processors.

The reduction on the number of processors is based on the fact that certain operations have to be performed on the set S of the supervertices and not on all the vertices. However, in order for the processors to set themselves up so as they know which vertices are in S and be selected to perform the various operations, an array, say Q, can be set up such that Q(0),Q(1), ... , Q(M-1) represents the elements in S, where $m=|S|$. Q(j) for j runs from 1 to m would be used to replace the condition "for all $i \varepsilon s$" in the algorithm. The array Q and m can be updated at each iteration in 0(log n) steps by applying the fast parallel sorting algorithm described in [11,17] on the array Flag. Thus, step 2a in algorithm MOD.CONNECT can be replaced by calling the sorting procedure in [11,17] as

2a SORT(FLAG,Q,n)

Procedure SORT sorts the input binary array Flag in time 0(log n) with n processors, returns array Q with the property that Flag (Q(j)) = 1 for $0 \le j < m$ and 0 elsewhere, where m is the number of 1's in Flag (i.e. $m = |S|$). Besides the above changes, steps 2(b) and 7 are required to be considered accordingly based on array Q.

The following lemmas are useful in proving our results.

Lemma 1: Given n elements $\{a_0,a_1,\dots, a_{n-1}\}$and K processors, $A(n) = a_0*a_1*\dots*a_{n-1}$ can be computed in T time units, where * is any associative binary operation and

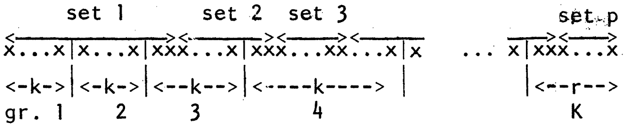
$$T = \begin{cases} \lceil n/K \rceil - 1 + \log K & \text{if } \lfloor n/2 \rfloor > K \\ \log n & \text{if } \lfloor n/2 \rfloor \le K \end{cases}$$

Proof: If $K \ge \lfloor n/2 \rfloor$,it has been shown that A(n) can be computed in log n time units by the technique of recursive doubling [9,22]. If $K < \lfloor n/2 \rfloor$ we partition $\{a_0,a_1, \dots,a_{n-1}\}$ into K groups, each of $\lceil n/K \rceil$ elements, except that the last group has $r = n - (K-1)\lceil n/K \rceil$ elements. Assign one processor to each group and then compute the groups in parallel. This takes $\lceil n/K \rceil - 1$ time units. These K results, one from each group, are then combined in parallel by the K processors, which takes another log K time units. Hence, the total time requirement is $\lceil n/K \rceil - 1 + \log K$ time units.

[]

Lemma 2: Let the n elements be partitioned into p sets and assume K processors available. The p products, one for each set, can be computed in at most T time units, where T is the same as given in Lemma 1 and * is an associative binary operation.

Proof: Align the p sets of elements as shown in Figure 2 and partition the elements into K groups as in the proof of Lemma 1.



set 1
set 2
set 3
set p
x...x|x...x|xxx...x|xxx...xx...x|x ... x|xxx...x
<-k->|<-k->|<--k-->|<----k----> | |<--r-->
gr. 1
2
3
4
K

where k = $\lceil n/K \rceil$ elements
r = n - (K-1) $\lceil n/K \rceil$ elements

Figure 2: Partition of p sets into K groups

Assign one processor to each of the K groups to compute the products in that group. If all the elements in a group belong to the same set, one answer will result from that group. If the elements in a group belong to several sets, say b sets, then b answers, one for each set, will be obtained. If b >2, at least b - 2 answers are final products and at most 2 answers in each group will be combined with answers in other groups to give a final product (the first and last groups each contribute at most one answer and the final product). For instance, (see Figure 2) groups 1 and 2 have one answer, group 3 has 2 answers and group 4 has 3 answers, one of them being a final product. It is obvious that no more than $\lceil n/K \rceil - 1$ time units are needed for computing answers in each group.

Let us assume that n_i answers will be combined to give the produce of set i. (In figure 2, $n_1 = 3$, $n_2 = 2$, and $n_3 = 1$.) Assign $\lfloor n_i/2 \rfloor$ processors to each set to compute the product of that set. Since $\Sigma_{i=1}^{p} \lceil (n_i-1)/2 \rceil \le K$, the total number of processors required will be less than K. Each set will take another $\log n_i \le \log K$ time units to obtain the final product. Thus the total time requirement is still no more than T as given in Lemma 1.

[]

Lemmas 1 and 2 give an upper bound on the parallel time complexity for computing a product of n elements and products of sets of n elements. As a matter of fact, it can be shown easily that this bound is at most one time unit from optimal [16]. Since the "Min" operation in step 2b and the "OR" operation in step 7 are associative binary operations, Lemmas 1 and 2 give an upper bound on the total number of time units spent in these steps.

Lemma 3: Given nK processors, step 2b in algorithm MOD.CONNECT takes at most 0(n/K+log nlog K) time if $1 \le K < \lfloor n/2 \rfloor$ and $0(\log^2 n)$ time if $K \ge \lfloor n/2 \rfloor$

Proof: As mentioned earlier or from [10,12], further iterations of steps 2-7 merge supervertices. Step 8 eliminates those merged vertices which are no longer supervertices. It is proved in [12] that the number of supervertices (flagged elements) i.e. |S|, in each connected component decreased by a factor of at least two after each iteration until the connected component is represented by a single supervertex. Moreover, if the whole connected component has merged to a single supervertex (i.e. the supervertex will be isolated), that supervertex will not be considered in the succeeding iterations since its flag is set to zero at step 3 in the iteration at which it becomes isolated. Thus, we have n flagged elements at the first iteration (i.e. m=n) and have at most $\lfloor n/2^i \rfloor$ flagged elements after i iterations. At step 2, in order to compute all C(i), K processors are assigned to each i to compute the minimum value among at most|S| elements. Since "Min" is an associative binary operation, we can apply Lemma 1 to evaluate the time complexity.

The program for step 2b can be described as follows:

2b The following steps are performed in parallel for $0 \leq i < m, 0 \leq j < K$, since $m \leq n$, the maximum number of processors is nK. It is assumed that $M = \lceil m/K \rceil$. $m = |S|$ and $S = \{Q(0), Q, (1), \ldots, Q(m-1)\}$ after step 2a.

```
(1)  for k<-- 0 until M-1 do
        for all i,j  do
           if (A(Q(i),Q(jM+k))=1 AND
             Flag (Q(jM+k))=1) then
             Temp(i,jM+k) <-- Q(jM+k)
           else Temp(i,jM+k) <--- γ

(2)  for k<--1 until M-1 do
        for all i,j  do
           Temp(i,jM) <-- min{Temp(i,jM),
            Temp(i,jM+k) }

(3)  for k<-- 0 until (log K)-1 do
        for all i,j, do
           Temp(i,jM) <-- min{Temp(i,jM),
            Temp(i,((j+2^k)mod K)M) }

(4)  for all i do
        if Temp(i,0) = γ then C(Q(i)) <--Q(i)
        else C(Q(i)) <-- Temp(i,0)
```

In the above program, γ stands for any number exceeding n-1. In step (1), the elements whose minimum is to be computed are stored in the array Temp. In step (2) the minimum values for all the groups (the number of groups $\leq$ K and the size of each group $\leq$ M) are found in time $O(m/K)$ via sequential search and all these groups are processed in parallel. Then, the overall minimum of the K minima is found (step (3)) in time $O(\log K)$ using at most mK processors at each step. The details for the time complexity are as follows:

Case 1: $1 \leq K < \lceil n/2 \rceil$, since $|S|$ is reduced by at least half after each iteration, $|S|$ is at most 2K after $t = \log n - \lceil \log K \rceil$ iterations. Thus, we have the following time bound, T.

$$T = \Sigma_{K=0}^{t-1} (\lceil (\lfloor n/2^k \rfloor)/K \rceil - 1 + \log K) + \Sigma_{K=t}^{\log n-1} \log(n/2^k)$$
$$\leq \lceil 2n/K \rceil + t \log K + (\log K)^2$$
$$\leq O(n/K + \log n \log K)$$

Case 2: $K \geq \lfloor n/2 \rfloor$, we have

$$T = \Sigma_{K=0}^{\log n-1} \log (n/2^k) = O(\log^2 n)$$

[]

Lemma 4: Given nK processors, step 7 in algorithm MOD.CONNECT takes at most $O(n/K + \log^2 n)$ time units if $1 \leq K < \lfloor n/2 \rfloor$ and $O(\log^2 n)$ if $K \geq \lfloor n/2 \rfloor$.

Proof: After sets of supervertices are merged in steps 4-6, the adjacency information among the supervertices is updated in step 7. Basically, step 7a puts an arc from vertex i to new supervertex j (i.e., $A(i,j)=1$) if there is an edge between i and a vertex merged into j. Step 7b puts an arc from supervertex i to supervertex j if there is an arc to j from a vertex merged into i. In step 7a or 7b those columns or rows in the adjacency matrix A corresponding to those vertices which are merged to supervertex j or i, are "OR"ed together to give the new column j or row i. Since "OR" is an associate binary operation, Lemma 2 can be applied to derive the time bound for step 7. There are m rows in A which correspond to S and these rows of elements are handled in parallel. As in step 2b, K processors are assigned to each row i to compute $A(i,j)$ in step 7.

Since the application of Lemma 2 assumes that the elements in the same set are grouped together, we have to apply the parallel sort algorithm in [12,17] on array D. As a consequence, all the elements which have the same value in the array D are grouped together by the following procedure call

SORT (D·FLAG,Q,m)

The input array is the inner product of the arrays D and Flag (i.e. the i'th element equals D(i) if Flag(i)=1 otherwise 0). Array Flag is used such that only those elements corresponding to the supervertices are considered. Since the information corresponding to the isolated supervertices need not be merged with any other supervertices, their corresponding Flag values have been assigned to 0 in step 3 and they will effectively be ignored. Procedure SORT basically arranges the supervertices according to their values in D and all those elements with Flag(i)= 0 are put at the end of the list. After the procedure call, array Q has the property that $D(Q(j)) \geq D(Q(i))$ if $j > i$ and $Flag(j)=Flag(i)=1$. The program for step 7a (similarly for step 7b) can be described as follows:

7a The following steps are performed in parallel for $0 \leq i < m$, $0 \leq j < K$. Since $m \leq n$, the maximum number of processors required is nK. It is also assumed that $M = \lceil m/K \rceil$.

```
(1)  SORT(D·Flag,Q , m)

(2)  for all j do Temp(j)<-- Q(jM)

(3)  for k<-- 1 until M-1 do
        for all i,j  do
           if D(Temp(j))=D(Q(jM+k) AND
           Flag(Q(jM+k))=1 then
           A(Q(i),Temp(j))<--OR{A(Q(i),Temp(j)),
                                A(Q(i),Q(jM+k))}
           else Temp(j)<-- Q(jM+k)

(4)  for all i,j do
        if D(Temp(j))=D(Q((j+1)M)) AND
        Flag(Q((j+1)M))=1 then
        A(Q(i),Temp(j))<-- OR{A(Q(i),Temp(j)),
                              A(Q(i),Q((j+1)M))}
```

```
(5)  for k<--0 until (log K)-1 do
        for all i,j  do
           if D (Temp(j)) = D(Temp(((j+2^k) mod
             K)M)) then A(Q(i), Temp(j))<--OR
             {A(Q(i), Temp (j)),
              A(Q(i),Q(((j+2^k)mod K)M))}

(6)  for all i,j  do A(Temp(j),Q(i))<--
      A(Q(i),Temp(j))
```

As in the proof of Lemma 2, the elements are partitioned into K groups each of which has M elements. In step (1) the elements are stably sorted such that all the elements with the same value in D (the same D-value) are grouped together (this refers to those elements which will later be merged together). The first element in each group is assigned to the array Temp in step (2). The columns of A with the same D-value in each group are "OR"ed together sequentially in step (3). The resultant column is stored at A(*,Temp(j)), where Temp(j) always remembers the index of the first column in the j'th group among all the columns which have the same D-value. In step (4), the two resultant columns which have the same D-value in two adjacent groups are "OR"ed together. In step (5), all the resultant columns with the same D-values are "OR"ed together and the final resultant column is stored in the smallest indexed resultant column. Since step (1) evokes a stable sort [11,16], it is easy to show that the smallest numbered column, say j, has the property that j = D(j), (i.e. it will become the supervertex in the later iterations).

During the first iteration, the K processors of one row must deal with n elements; and for each succeeding iteration, the number of elements to be dealt with by the K processors is at most half of the number in the previous iteration. Thus, using Lemma 2 and applying the same kind of analysis as in the proof of Lemma 3, we derive the time bound T as stated in the lemma.

[]

Theorem: Algorithm MOD.CONNECT finds the connected components of an undirected graph with n vertices in time $0(n/K + \log^2 n)$ using nK processors where $K \geq 1$.

Proof: The time and processor requirements are as follows:

Step	Total Time		Processors
	$1 \leq K < \lceil n/2 \rceil$	$K \geq \lceil n/2 \rceil$	
1	$0(1)$	$0(1)$	n
2a	$0(\log^2 n)$	$0(\log^2 n)$	n
2b	$0(n/K+(\log n)$ $(\log K))$	$0(\log^2 n)$	nK
3	$0(\log n)$	$0(\log n)$	n
4	$0(\log_2 n)$	$0(\log n)$	n
5	$0(\log^2 n)$	$0(\log^2 n)$	n
6	$0(\log n)$	$0(\log_2 n)$	n
7	$0(n/K+\log^2 n)$	$0(\log^2 n)$	nK
8	$0(\log n)$	$0(\log n)$	n

Thus, nK processors suffice to determine the connected components of an undirected graph with n vertices in time $0(n/K + \log^2 n)$.

[]

As a by-product of our main theorem, we have the following result.

Corollary: Given $n\lceil n/\log^2 n\rceil$ processors, algorithm MOD.CONNECT determines the connected components of an undirected graph with n vertices in time $0(\log^2 n)$

This method uses the least number of processors yet to find the connected components of an undirected graph in time $0(\log^2 n)$. The previous method [12] needs $n\lceil n/\log n\rceil$ processors to achieve the same time bound. It can be shown easily that if K = 1, i.e. n processors are available, algorithm MOD.CONNECT takes 0(n) time. If less than n processors are available (i.e. K <1), each parallel step will be repeated $\lceil 1/K \rceil$ times and the total required time will be $0(\lceil n/K \rceil)$. As a matter of fact, this algorithm takes $0(n^2)$ time with 1 processor (i.e. K = 1/n) and also, this algorithm is optimal in the sense that the speedup ratio is linear with the number of processors available as long as the total number of processors is no more than $n \lceil n/\log^2 n \rceil$.

IV. CONCLUSION

We have proposed algorithm MOD.CONNECT to find the connected components of an undirected graph and have derived a time bound for the algorithm using a fixed number of available processors. We can also show that several related problems can be solved in the same time and processor bounds [4]. In particular, these problems can be solved in $0(\log^2 n)$ time using $n\lceil n/\log^2 n\rceil$ processors. This method is superior to the previous methods [19,20] because it uses the least number of processors for the same time bound. The technique employed in our algorithm is a kind of problem decomposition which is similar to what is used in [19] for finding the mininum element in an array of n elements. It exploits the property that the problem size is reduced by at least half after each iteration and thus the processor requirement can be reduced by a factor of log n over existing algorithms. However, other problems, such as finding the transitive closure of an asymmetric Boolean matrix and the strongly connected components of a directed graph, can be shown to be reducible to the matrix multiplication problem [3,18], whose time complexity is $0(n^{2.81}\log n/k)$ using $K \leq n^{2.81}/\log n$ processors with $S_k=0(K/\log n)$ Since the size of the problem remains constant after each iteration, the idea of reducing the number of processors by a factor of log n is not directly applicable. It remains an open problem to determine whether there exist algorithms for these problems whose speedup ratios are linear with respect to the number of processors available.

ACKNOWLEDGEMENTS

The authors wish to thank one of the referees for correcting one mistake in this paper and the referees of our other paper [4] in particular for their constructive comments and correcting several errors in our original manuscript.

V. REFERENCES

[1] Baudet G. and D. Stevenson, "Optimal Sorting Algorithms for Parallel Computers, " IEEE Trans. on Computers, Vol. C-27, Jan. 1978, pp.84-87.

[2] Berge C. and Chouila-Houri, A., Programming, Games and Transportation Networks, Wiley, 1965, p. 179.

[3] Chandra A.K., "Maximal Parallelism in Matrix Multiplication, "IBM Research Report, RC 6193 1976.

[4] Chin F., J. Lam and I. Chen, "Efficient Parallel Algorithms for Some Graph Problems," Technical Report, University of Alberta, (Submitted to CACM).

[5] Csanky L., "On the Parallel Complexity of Some Computational Problems," Ph.D. Dissertation, Comp. Sci. Division, U. of California, Berkeley, 1974.

[6] Eckstein D.M. and D.A. Alton, "Parallel Graph Processing Using Depth-First Search," Conf. on Theoretical Comp. Sci., U. of Waterloo, 1977, pp.21-29.

[7] Flynn M., "Some Computer Organizations and Their Effectiveness, "IEEE Trans. on Computers, Vol. C-2], Sept. 1972, pp. 948-960.

[8] Golaschlager L.M., "Synchronous Parallel Computation," Ph.D. Dissertation, TR 114, Dept. of Comp. Sci., U. of Toronto, 1977.

[9] Heller D., "A Survey of Parallel Algorithms in Numerical Linear Algebra," SIAM Review, Vol. 20, Oct. 1978, pp.740-777.

[10] Hirschberg, D.S., "Parallel Algorithms for the Transitive Closure and the Connected Component Problems," Proc. of 8th Annual ACM Symposium on Theory of Computing, 1976, pp.55-57.

[11] Hirschberg D.S., "Fast Parallel Sorting Algorithms," CACM, Vol. 21, Aug. 1978, pp.657-661.

[12] Hirschberg, D.S., A.K. Chandra and D.V. Sarwate, "Computing Connected Components on Parallel Computers," CACM, Vol 22, Aug. 1979, pp.461-464

[13] Hyafil L. and H.T. Kung, "Parallel Algorithms for Solving Triangular Linear Systems with Small Parallelism," Dept. of Comp. Sci., Carnegie-Mellon U., Pittsburgh, Pa., 1974.

[14] Ja'Ja', J. and J. Simon, "Parallel Algorithms in Graph Theory - Planity Testing" T.R. Penn State University, June 1980.

[15] Kuck D.J., "A Survey of Parallel Machine Organization and Programming," ACM Computing Surveys, Vol. 9, March 1977, pp.29-59.

[16] Munro I. and M. Paterson, "Optimal Algorithms for Parallel Polynomial Evaluation," JCSS, Vol. 7, April 1973, pp. 189-198.

[17] Preparata F.P., "New Parallel-Sorting Schemes," IEEE Trans. on Computers, Vol. C-27, July 1978, pp.669-673.

[18] Reghbati E. and D.G. Corneil, "Parallel Computations in Graph Theory, "SIAM J. Computing, Vol. 7, May 1978, pp.230-237.

[19] Savage,C.D., "Parallel Algorithms for Graph Theoretical Problems," Ph.D. Dissertation, R-784, Dept. of Math., U. of Illinois, Urbana, 1977.

[20] Savage, C.D. and J. Ja'Ja', "Fast, Efficient Parallel Algorithms for Some Graph Problems," Technical Report, Penn State University, 1980.

[21] Stone H.S., "Problems of Parallel Computation," Complexity of Sequential and Parallel Numerical Algorithm, Academic Press, 1973, pp.1-16.

[22] Stone H.S., "An Efficient Parallel Algorithm for a Tridiagonal Linear System," JACM, Vol. 20, Jan. 1973, pp.27-38.

[23] Thompson C. and H.T. Kung, "Sorting on a Mesh-Connected Parallel Computer," CACM, Vol. 20, April 1977, pp.263-270.

SPEEDUP BOUNDS FOR CONTINUOUS SYSTEM SIMULATION ON A HOMOGENEOUS MULTIPROCESSOR

E.H. D'Hollander
State University of Ghent
Department of Applied Mathematics
Coupure Links 533
B-9000 Ghent, BELGIUM

Abstract -- This paper explores the benifits and the bounds of multiprocessors for the simulation of continuous systems. Different types of parallelism are defined describing the stepwise refinement of a problem into parallel executable tasks. Invariant simulation systems have a great parallelism in time, due to their periodic execution for each integration step. When a problem can be partitioned into tasks which are independently scheduled, it has natural parallelism. A problem structure having precedence constraints among tasks exhibits functional parallelism and finally a task which further is split-up in atomic operations exploits the operator parallelism. For each of these forms the processor utilization and speedup bounds are analysed with respect to the structural characteristics of the simulation problem.

1. Introduction

The idea to apply multiprocessor systems in the domain of continuous system simulation is motivated mainly by the following considerations. First, the need for fast simulation power is recognized in many applications, but it is most stressed in the field of interactive simulation and in real time systems. Second, most digital simulation is cpu-bound, since numerical integration of a complex set of differential equations is calculation-intensive. Because of the heavy cpu-load, the use of several processing units is likely to produce a faster solution. However, the final speedup is bound by the processor-system as well as the problem characteristics. The aim of the following sections will be to focus on the problem dependent characteristics which influence the potential speedup on a MIMD-machine. After a general problem formulation in section 2, the different types of parallelism will be defined in section 3. In section 4 the problem-dependent factors, limiting the unconstrained use of parallelism are discussed and some useful bounds on cpu-utilization will be derived. Attention is given to the architectural aspects where they might constitute a potential bottleneck.

2. Problem Definition

A simulation model S, is described by the following set (Fig. 2.1) :

- the time, t ;
- the input-set, $\underline{x}$;
- the state variables, $\underline{q}$;
- the output-variables, $\underline{y}$;
- the derivative functions, $\underline{f}$;
- the output functions, $\underline{g}$.

We consider a general multiprocessor MP, consisting of :

- n identical processors ;
- m memory-modules ;
- an interconnection system I, which describes the coupling between processors and memories.

A multiprocessor is coded by the software P, yielding a programmed computer system, K :

$$K = \{n,m,I,P\} \qquad (2.1)$$

When a computer system K solves the simulation problem S, the code for executing S is partitioned and allocated to the different processors by the mapping π :

$$P = \pi(S) \qquad (2.2)$$

A programmed multiprocessorsystem K, has an execution time, t_e which is function of

- the machine dependent characteristics, n,m,I;
- the implementation of the problem, $\pi(S)$:

$$t_e = F\,[\,n,m,I;\ \pi(S)\,] \qquad (2.3)$$

The minimization of t_e therefore depends on machine- and on problem-characteristics. Whereas the architectural aspects have received ample considerations in the literature, especially for the possible interconnection structures, this paper contributes to the equally important partitioning problem. From the results obtained, it should be possible to select the proper n, m and I, in order to tailor the multiprocessor to the type of problems it will solve.

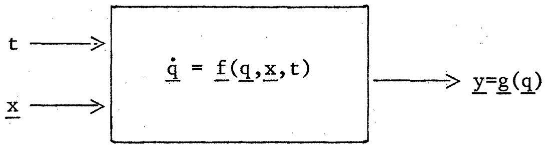

t
$\underline{x}$
$\dot{\underline{q}} = \underline{f}(\underline{q},\underline{x},t)$
$\underline{y}=\underline{g}(\underline{q})$

Fig. 2.1. The continuous system $S = \{t,\underline{x},\underline{q},\underline{y},\underline{f},\underline{g}\}$

3. Types of parallelism

3.1. Partitioning steps

The partitioning, $P = \pi(S)$ proceeds in two distinct steps (Fig. 3.1).

$$S \xrightarrow{\pi_1} C = \{J,<\!\cdot\,\} \xrightarrow{\pi_2} P$$

Fig. 3.1. Simulation system (S), Task system (C) and Processor-coding (P)

In the first pass π_1, the model S is transformed into a task system $C = \{J, <\!\cdot\,\}$. A task system consists of a set of tasks, $J = \{T_i\}$ subject to an ordering $<\!\cdot$. This ordering indicates the precedence constraints governing the execution of J. A task T_i, operating on the results of task T_j, requires the prior termination of T_i before task T_j can initiate. This execution ordering is

0190-3918/81/0000/0176$00.75 © 1981 IEEE

denoted by $T_i <\cdot T_j$. In the second step π_2, the task system C is scheduled and programmed on the available processors, by the coding P. This implies compilation and downloading of the tasks T_i, together with the necessary synchronization primitives. The scheduling strategy has to take into account the precedence constraints of the task system, the duration of each task, and eventually the parallelism within a task T_i. Given an unsupervised scheduling algorithm, step π_2 is totally transparent to the user, whereas normally a limited user-assisted partitioning occurs in step π_1. Since the number of processors n, is only introduced in the unsupervised partitioning step π_2 this approach permits a graceful degradation.

3.2. Parallelism in time

Whenever a simulation problem S has a deterministic structure, its task system will be identical for each time step. Consequently, the scheduling and compilation is the same for all integration steps and needs to be done only once.
On the other hand, if the structure of the problem varies according to state changes during execution, the parallel implementation requires a rescheduling of the task system, in order to account for any variations in the ordering $<\cdot$, the duration times or the task set J. The influence of these structure-variations is estimated by the 'parallelism in time'. The time-parallelism is defined as the average number of integration steps during which the problem structure is fixed. A high parallelism in time justifies an elaborated optimization of the scheduling strategy.

3.3. Natural parallelism

Each integration step of a set of differential equations requires successively :

1) the numerical integration of the state vector ($\underline{q}$)
2) the calculation of the derivative functions $\dot{\underline{q}} = f(\underline{q},\underline{x},t)$.

The corresponding set of tasks J, generally can be partitioned into several subsets B_j of tasks T_i, which do not interact during the execution of an integration step, i.e. : $C = \{B,O\}$ with $B=\{B_j\}$
Consequently, all sets B_j can run concurrently without synchronization. This form of parallelism is termed 'natural', and it is quantified by the number of subsets in B. Many simulation systems exhibit a natural parallelism, since they can split up in logically independent subsystems.
A prominent example of natural parallelism is given by the independent state equations (Fig.2.1). First all state variables can be integrated in parallel. Then the state vector is communicated to the different processors and finally each derivative function is evaluated simultaneously [7].

3.4. Functional parallelism

When there exists precedence constraints between the tasks of a simulation system or subsystem, the ordering relation $<\cdot$ is not empty. The parallelism which respects this ordering $<\cdot$ is termed 'functional'. It is not possible to extract this functional parallelism immediately from the problem description S, or during phase π_1, where the task system C is created. It has to be recognized in the phase π_2, and the partitioning algorithm has to take into account the absolute priorities between executing tasks, the task duration times and the communication- and synchronization-overhead. Typically this parallelism is applied to the parallel execution of derivative function calculations.

3.5. Operator parallelism

A fourth form of parallelism, also occuring in the partitioning phase π_2, arises when a task T_i, is further split up, in order to increase parallel execution. The operator parallelism gives rise to two subforms : the micro- and the macro-operator-parallelism, depending on its effect on the structure of the whole task system C. In the micro-form, the task T_i is searched for parallel executable basic-operators, such as multiplication and addition ; the structure of the task system is then changed only locally [17],[18],[19]. The macro-form has a profound effect on the global task system structure. There the splitting of operator T_i results in a split-up of the whole task system. This happens for certain parallel integration algorithms, e.g. when the integration formulas allow the system to be evaluated simultaneously for two consecutive timesteps [7],[21],[23].

4. Problem-dependent performance bounds

4.1. The importance of time-invariant systems

Clearly the major advantage of most simulation systems with respect to parallel processing, is the repetitive execution of the same calculations during each timestep. Ideally this requires that task duration times and problem structure (i.e. the precedence constraints) remain invariant during the whole integration interval. The condition of a time-invariant structure, however, can be relaxed to include continuous systems which switch over during execution time between a limited number of alternating structures. In this case all possible structures are partitioned in advance and downloaded into the different processor-memories. In this way the jump to a new structure, even as a result of a state-change in the model, can be realized with minimal overhead, similar to a subroutine-call in sequential processing. However, this method fails in two cases. First the 'context-switching' becomes predominant whenever the model rapidly alternates between several different structures, i.e. when the time-parallelism is low. Second, the number of possible structures grows exponentially with the number of 'switchpoints' in the model : these are the points where a selection is made between alternate functions to evaluate (compare with the switches in an analog blockscheme). Few analytical results exist on the influence of variable task duration times. Several simulation results, however, seem to indicate that slight variations on the estimated task length have only a marginal effect on the scheduling efficiency [1]. Task lengths can be estimated at compile time from the duration of the individual instructions [20].

4.2. Automatic partitioning algorithms for natural and functional parallelism

The general assignment problem can be stated as follows. We are given :

1) a task system $C = \{J, <\!\cdot\}$ in which $J = \{T_1, \ldots, T_N\}$ equals a set of tasks and $<\!\cdot$ is the partial ordering relation : $T_i <\!\cdot T_j$ denotes that T_j cannot start execution prior to the complētion of T_i ;
2) a weighting function $\alpha(T_i)$, representing the execution time : $\tau_i = \alpha(T_i)$;
3) a fixed number of identical processors, n.

The objective is to find a partition $A_1, \ldots, A_n$ of J, such that the largest execution time on any processor

$$t_{max} = \max_{\forall i} \{ \sum_{\tau_j \in A_i} \tau_j \} \qquad (4.1)$$

is minimized, subject to the precedence constraints, $<\!\cdot$.

It is well known that for general values of n and m, this problem is NP-complete [14],[22] . Therefore considerable attention has been given to the development of fast heuristics, yielding suboptimal results [1],[5],[9] . The general problem formulation above, involves the detection of natural parallelism ($<\!\cdot = 0$) as well as functional parallelism ($<\!\cdot \neq 0$). In the following paragraphs, two common heuristics for this partitioning problem will be analyzed.

Natural parallelism ($<\!\cdot = 0$). In this case, the tasks are independent. Intuitively it seems useful to assign the longest tasks first. In this way the smaller tasks are reserved for the end, and can be used to reduce the irregularities of the distribution. This leads to the 'Longest Process Time' algorithm [4] :

1) arrange the tasks in a list, in decreasing order of execution times ;
2) assign each task from this list consecutively to the first available processor.

This is a so-called 'list-algorithm'. The list algorithms differ only in the way a list of tasks is arranged. In contrast to an optimal search by enumerative techniques, the LPT-algorithm is relatively efficient. The sorting of the list takes $O[N.\log(N)]$ steps, and the assignment phase requires $O[N(n-1)/2]$ operations. For large N and constant number of processors n, the algorithm is of order $O[N.\log(N)]$.

Functional parallelism ($<\!\cdot \neq 0$). The ordering relation $<\!\cdot$, governing the execution priority of the tasks, is represented by a task graph. This is the tupple $[J, \alpha(T), <\!\cdot]$ and consists of vertices denoting tasks, and edges denoting the precedence constraints. Again a list algorithm is applied for the automatic scheduling. In order to account for the precedence relations however, the weight of a task T_i is measured by its level. A task T_i has level ℓ, when the longest path from that task to a terminal task requires ℓ time-units. Consequently, $\ell(T_i)$ is the minimal time needed to terminate the execution of the task graph, from the beginning of task T_i. It is intuitively appealing to assign the highest priority of execution to the tasks with highest level, i.e. to those tasks with the largest workload ahead. This leads to the following 'level algorithm', which originates from the optimization of assembly lines [13] :

1) arrange the tasks by decreasing levels ;
2) whenever a processor becomes free, assign that task of which all predecessors are executed, and which has the highest level of the remaining tasks. Ties are arbitrarily resolved.

The workload of this algorithm depends on the number of tasks N, the average number of predecessors of each task, Npred, and the number of processors n ; for moderate values of Npred and n, however, the algorithm is $O[N^2/2]$.

4.3. Bounds on processor utilization and speedup

It is not a rule that a multiprocessor of n identical processors will perform n times faster than a single processor. Although several architectures bear this potential, even the best equipped systems will be more or less seriously limited by the constraints of the problem. It is the aim of this section to derive the lower and upper bounds of processor utilization with respect to the problem characteristics. Several authors have demonstrated the suboptimality of the LPT- and level-algorithms [11],[13],[15] . Here we concentrate on the suboptimality conditions for the processor utilization taking into account algorithmic-, problem- and processor-characteristics. We define the following performance measures.
The effective execution time t_e, is the processing time of the longest operating processor.
The minimal execution time t_{min}, of any problem on an n-processor system is :

$$t_{min} = E/n \qquad (4.2)$$

with $E = \sum_{\forall i} \tau_i$ the total workload of the task system.

The processor utilization U is defined as the average fraction of time that the processors are busy during t_e :

$$U = t_{min}/t_e \qquad (4.3)$$

The speedup S of a n-processor system over a uniprocessor is $S = U.n$.

Upper bounds. The effective calculation time of a task system is bound below both by the longest chain of tasks to be executed serially and by the number of processors n, i.e. the degree of hardware parallelism. When there are no precedence constraints and we do not allow pre-emption, one has

$$t_{e,min} = \max\{\tau_{max}, E/n\} \qquad \text{for } <\!\cdot = 0$$

with τ_{max} the longest task duration. In the case of precedence constraints, the effective execution time is bounded below by the longest path in the task graph. According to the previous definition, this is the highest task-level L :

$$t_{e,min} = \max\{L, E/n\} \qquad \text{for } <\!\cdot \neq 0$$

For obvious reasons it is assumed that the number of tasks exceeds the number of processors, i.e. $N \geqslant n$. Define the average task length $\bar{\tau} = E/N$. Then the processor utilization and speedup have the following upper bounds :

$$U_{max} = \min \{\frac{E}{n.\ \tau_{max}}\ ,\ 1\} \quad \text{for} \lessdot = 0 \qquad (4.4)$$

$$S_{max} = \min \{\frac{E}{\tau_{max}},\ n\}$$

and

$$U_{max} = \min \{\frac{E}{n.L}\ ,\ 1\} \qquad \text{for} \lessdot \neq 0 \qquad (4.5)$$

$$S_{max} = \min \{\frac{E}{L}\ ,\ n\ \}$$

Lower bounds. First we consider the independent task system ($\lessdot = 0$). Denote by t_i the starting time of task T_i. The minimal execution time t_{min} given by (4.2) requires that all processors remain active during the interval $[\,0,t_{min}\,]$.
In each of the list-algorithms, the last task T_N is started on the first available processor, at t_N. Thus, till t_N all processors are busy executing the previous N-1 tasks, yielding an upper bound for t_N :

$$t_N \leqslant (\sum_{i-1}^{N-1} \tau_i)/n \qquad (4.6)$$

Let $T_{\ell_1},\ldots, T_{\ell_n}$ be the last tasks executed on each of the n processors. These tasks start ultimately at t_N. From that moment the execution will not last longer than the maximal duration of these n tasks, $\tau_{\ell,max}$ with $\tau_{\ell,max} = \max_{j=1,n} \tau_{\ell_j}$.
This gives an upper bound for the execution time of each list-algorithm : $t_e \leqslant t_N + \tau_{\ell,max}$. Taking into account inequalities (4.6) and (4.2), $t_{min} > t_N$ and the execution time of the LPT-schedule is bounded by

$$t_{LPT} < t_{min} + \tau_{\ell,max} \qquad (4.7)$$

demonstrating that the LPT-schedule always terminates within $\tau_{\ell,max}$ of the absolute minimal execution time. With $t_{opt} \geqslant t_{min}$, equation (4.7) also yields a suboptimality bound for the algorithm :

$$\frac{t_{LPT}}{t_{opt}} \leqslant 1 + \frac{\tau_{\ell,max}}{t_{opt}} \qquad (4.8)$$

This bound is comparable with the Graham bound [11] :

$$\frac{t_{LPT}}{t_{opt}} \leqslant \frac{4}{3} - \frac{1}{3n}$$

Both bounds are represented in Fig. 4.1, for large values of n ($n > 10$) and with the normalization $\tau_{\ell,max} = 1$. This figure illustrates that for $t_{opt} > 3n/(n-1)\ \tau_{\ell,max}$, equation (4.8) gives a lower bound. Moreover, (4.7) can be written as :

$$\frac{t_{LPT}}{t_{min}} \leqslant 1 + \frac{n.\ \tau_{\ell,max}}{N.\ \bar{\tau}} \qquad (4.9)$$

When the number of tasks N, grows indefinitely, and the mean task length $\bar{\tau}$ exceeds an arbitrary value $\varepsilon > 0$, (4.9) yields $\lim_{N\to inf} t_{LPT} = t_{min}$. Therefore, the LPT-algorithm is asymptotically optimal.

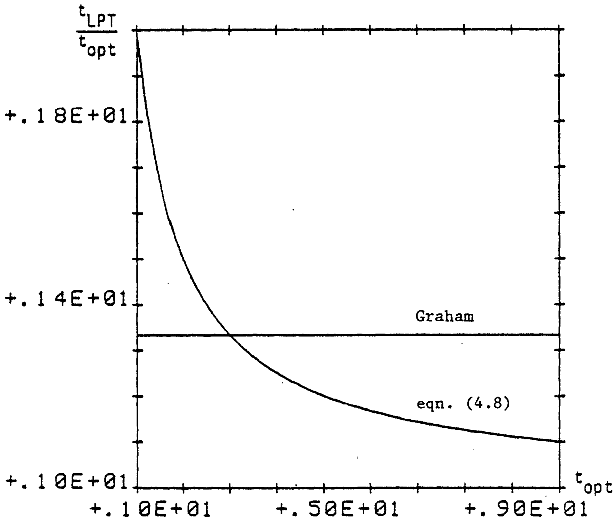


Fig. 4.1. Comparison of the execution-time bounds for independent-task systems ($\lessdot = 0$), scheduled by the LPT-algorithm

With $U = t_{min} / t_{opt}$, the lower bound for the processor utilization is given by the inverse of the upper limit in (4.9) and

$$S_{min} = U_{min}.n = n/\ \{1 + \frac{n.\tau_{\ell,max}}{E}\ \}$$

for $\lessdot = 0$.

For the general case where $\lessdot \neq 0$, the longest path, L, defines the minimal execution time $t_{e,min}$, which is independent of the number of processors n. When there is no bound on the number of processors, all other tasks can run concurrently with the longest chain tasks, yielding a total execution time L. In the worst case, however, due to precedence constraints, no tasks can be executed in parallel with the longest path. In order to be consistent with the definition of longest path, this requires that the remaining tasks can be executed in zero time, by an even partitioning over an infinite number of processors. This collection of remaining tasks is called an 'impulse task'. Consider a task system having a total task duration E = 1, and a longest path $L \leqslant 1$. In the worst case, this problem requires the execution of an impulse-task of 1-L time units, after the execution of the longest path. Since an impulse-task can be divided evenly over n processors, this gives an additional workload of $\delta = (1-L)/n$ time units (Fig. 4.2). The total execution time is $t_e = L + \delta$. With $t_{min} = 1/n$, the processor utilization of the worst case, $U = t_{min}/t_e$, gives the lower bound

$$U_{min} = 1/\ [\ 1 + (n-1)L\ ]\ .$$

This lower bound is also related to the mean width

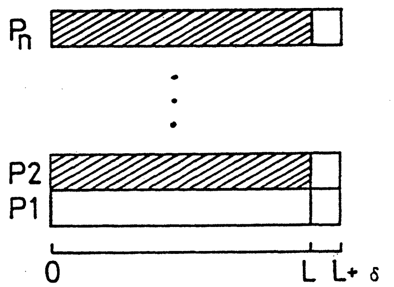


Fig. 4.2. Worst case task system subject to precedence constraints $(\lessdot \neq 0)$

of the task graph, W, which is defined as follows. Suppose the task graph is executed on an unlimited number of processors. At each instance of time, t, $t \in [0,L]$, there are w(t) processors active, w(t) equals the number of parallel executed tasks at time t, and is conveniently called the width of the task graph. The mean width, W, is now defined:

$$W = \frac{1}{L} \int_0^L w(t)\, dt$$

The integral value represents the total active processor time, which clearly equals the total task duration time E, so W = E/L and consequently with E = 1,

$$U_{min} = 1/ [1 + (n-1)/W].$$

This is the best possible bound, since one always can construct a taskgraph with longest path L, giving minimal utilization on n processors.
The lower bound for speedup becomes :

$$S_{min} = n/ [1 + \frac{n-1}{W}] \qquad \text{for } \lessdot \neq 0$$

4.4. Parallelism and Communication

The degree of parallelism and the degree of communication are strongly interconnected. Therefore, the interconnection network - mainly used for traffic between processors and memories - should be tuned to the type of parallelism which is exploited. Parallel program execution can be static or dynamic. Analytical and simulation studies [3],[12] have demonstrated that simultaneous execution of programs in an n processor, n memory system, coupled through a crossbar switch, can result in a significant loss of efficiency when processors randomly access memories other than their preferred memories for the execution of instructions. Parallel continuous system simulation, however, mainly involves the repetitive execution of static programs which are assigned permanently to the same processors. Therefore, the programs can be stored in private memories, one for each processor. Using private memories for program execution, the total bandwith of the interconnection system becomes available for data communication. Memory access can be scheduled or arrive at random. Some authors suggest the scheduling of tasks of a highly structured taskgraph could take into account the possibly hierarchically structured communication paths, in order to program highly interactive tasks on 'nearby'-processors [2]. However, for general simulation problems this complicates unnecessarily the scheduling algorithm. Indeed, communication conflicts should be more an exception than a rule, so their minimization through a well-balanced partitioning will normally have only a marginal effect. In fact, the hardware interconnections have to provide the required support for the statistically expected traffic load within reasonable efficiency bounds. Therefore, a homogeneous multiprocessor system with a non-hierarchical bus-structure is considered. In order to estimate the impact of the interconnection structure, a single bus-structure queuing model is considered as an example (Fig. 4.3).

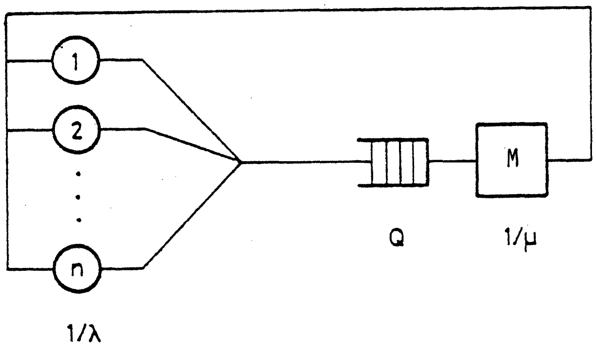


Fig. 4.3. Queuing model of an n-processor, 1 shared memory architecture

The model consists of n identical processors, requesting information from the shared data-memory, which is the server. The service time 1/μ, depends on the number of variables which are written into or read from the memory. These are the numbers of input variables of a task plus one output variable i.e. typically 0 - 10 memory cycles. The system is self regulating (closed loop), since requesting processors in the queue become non-active until they receive service. Since a processor requests service at the end of each task, the mean request-interarrival time, 1/λ equals the mean task-execution time. Assuming exponential service and interarrival times, this queuing model yields an estimate for the effective response time R, which is function of λ and μ [16] :

$$R = \frac{n}{\mu(1-p_o)} - 1/\lambda \qquad (4.10)$$

with

$$p_o = \{ \sum_{i=o}^{n} \frac{n!}{(n-i)!} \rho \}^{-1}$$

the probability that the shared memory stays idle, and

$$\rho = \frac{\text{average communication time per task}}{\text{mean task duration}}$$

The impact of bus conflicts on the effective speedup of n parallel operating processors is given by the efficiency factor

$$\eta_{bus} = \frac{1/\lambda + 1/\mu}{1/\lambda + R} \quad \frac{(1+\rho)(1-p_o)}{n.\rho}$$

and which is shown in Fig. 4.4 for various values of n.

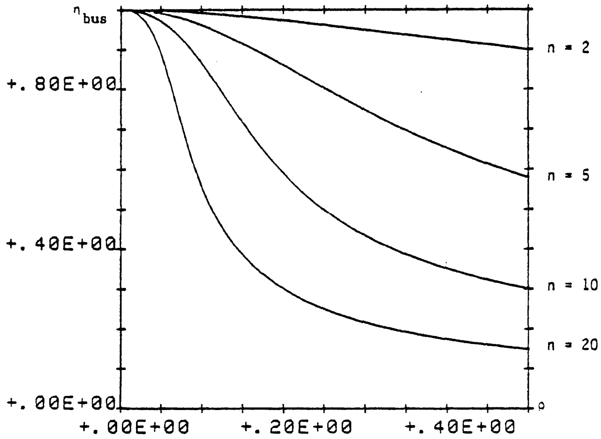


Fig. 4.4. Bus efficiency in function of inter-task communication (ρ) and number of processors (n)

The figure illustrates that low values of ρ (< .1) reduce the influence of bus-conflicts. In order to hold the queueing time below 10%, ($\eta_{bus} > .9$), critical values of ρ are given in table I.

n	ρ_{cr}
5	.16
10	.09
20	.05

Table I. Maximal ρ-values for $\eta_{bus} > .9$

From this table a rule of thumb, $\rho.n < 1$ can be derived. This inequality stresses the bounds for parallel task execution in a one-bus interconnection structure. Moreover the result is robust with respect to varying distributions of arrival and service times as is shown analytically [16] and by simulation [6]. From the rule it is possible to predict quantitatively the communication performance of a task system on a single bus-multiprocessor in terms of the average duration and mean communication time of a task. The bound especially applies to micro-operator-parallelism, since the duration time of basic operators such as addition and multiplication may be of the same order as the data-transfer time. Unfortunately, few publications take into account this communication overhead, which may well exceed the effective calculation time [8]. It is noted that an m-port memory or m multiple memories interconnected by a crossbar reduce the bus-conflicts significantly, provided the memory accesses are spread equally over all communication paths by an appropriate partitioning of the shared variables over the available memories. Using the queuing network theory of Gordon and Newell [10] for exponential distributions one finds a similar result:$\rho.n/m \ll 1$ [6]. In order to estimate the relative influence of communication and precedence constraints, we consider a unit task with longest path L = .05, or mean taskgraph width W = 20. The speedup is the inverse of the execution time of a unit task. The maximal and minimal bounds on a single bus, n-processor system are shown in Fig. 4.5, using the values ρ = 0 (no communication overhead), ρ = .025 and ρ = .05. Apparently bus conflicts cause a serious efficiency loss when n > 1/ρ. The maximal speedup is 20, ideally achieved with 20 processors. Interestingly however, in the worst case a doubling of the processors allows the speedup to become 12, i.e. 60% of its maximum, provided the connection system is not saturated (ρ = <.025).

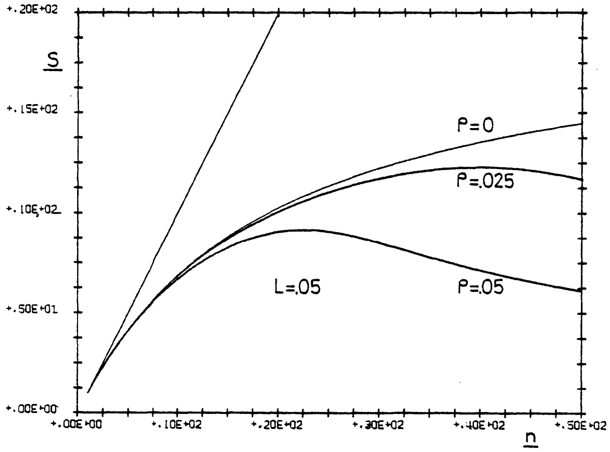


Fig. 4.5. Ideal and worst case performance of a taskgraph with mean width W = 20 (longest path L = .05) on n processors, 1 shared memory.
Upper line : maximal speedup S.
Lower lines : minimal speedup S with no (ρ = 0), moderate (ρ = .025) and high (ρ = .05) bus traffic.

5. Conclusion

Continuous simulation constitutes a fruitful application to parallel processing techniques, mainly because of its invariant and repetitive tasksystem, i.e. its parallel structure in time. The state equations are independent tasks that can be distributed evenly over the available processors by a simple, efficient and asymptotically optimal LPT-scheduling algorithm. More parallelism can be gained in building up a tasksystem of the derivative functions, thereby introducing precedence constraints between tasks. This functional parallelism can be scheduled transparently to the user by a numerically simple, yet powerful level algorithm. Minimal and maximal speedup bounds have been derived in function of the longest path L, or equivalently the mean task graph width W. Further refinement of tasks into basic operators raises the problem of communication overhead. Queuing analysis of a single bus interconnection between processors and a shared data memory reveals that the duration of the average task should exceed n times its communication time, where n equals the number of active processors. This result can be generalized for other interconnection structures by the theory of closed queuing networks.

References

[1] Adam, T.L., Chandy, K.M. and Dickinson, J.R., "A comparison of list schedules for parallel processing systems", Comm. ACM 17, 12, 1974, pp. 685-690.

[2] Arvind and Bryant, R.E., "Parallel Computers for Partial Differential Equations", Proc. Sc. Conf. Inf. Exch. Meeting, Livermore, California, 9, 1979, pp. 94-102.

[3] Bhandarkar, D.P., "Some performance issues in multiprocessor system design", IEEE Trans. Comp. 26, 5, 1977, pp. 506-511.

[4] Coffmann, E.G. Jr. and Denning, P.J., "Operating Systems Theory", Prentice Hall, 1973.

[5] 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", Int. Conf. Par. Proc., Ed. J.L. Baer, 8, 1977, pp. 95-99.

[6] D'Hollander, E.H., "Multiprocessors for Continuous System Simulation", PhD. thesis, State University of Ghent, Belgium, 1980.

[7] Franklin, M.A., "Parallel solution of ordinary differential equations", IEEE Trans. Comp. 27, 1978, pp. 948-960.

[8] Gentleman, W.M., "Some complexity results for matrix computation on parallel processors", J.ACM 25, 1, 1978, pp. 112-115.

[9] Gonzalez, M.J., "Deterministic processor scheduling", Computing Surveys 9, 9, 1977, pp. 173-204.

[10] Gordon, W.J. and Newell, G.F., "Closed Queuing Systems with Exponential Servers", Op. Res. 15, 1967, pp. 254-265.

[11] Graham, R.L., "Bounds on multiprocessing timing anomalies", SIAM J. Appl. Math. 17, 2, 1969, pp. 416-429.

[12] Hoogendoorn, C.H., "A general model for memory interference in multiprocessors", IEEE Trans. Comp. 26, 1977, pp. 998-1005

[13] Hu, T.C., "Parallel sequencing and assembly line problems", Op. Res. 9, 6, 1961, pp.841-848.

[14] Karp, R.M., "Reducibility among combinatorial problems", Complexity of computer computation, Plenum Press, N.Y., 1972, pp.85-104.

[15] Kaufman, M.T., "An almost optimal algorithm for the assembly line scheduling problem", IEEE Trans. Comp. 23, 11, 1974, pp.1169-1174.

[16] Kleinrock, L., "Queuing Theory II", John Wiley and Sons, 1977.

[17] Kuck, D. and Muraoka, Y., "Bounds on the parallel evaluation of arithmetic expressions using associativity and distributivity", Acta Informatica 3, 1974, pp. 203-216.

[18] Kuck, D., "A survey of parallel machine organization and programming", Computing Surveys 9, 1, 1977, pp. 29-59.

[19] Muller, D.E. and Preparata, F.P., "Restructuring of arithmetic expressions for parallel evaluation", J.ACM 23, 1976, pp. 534-543.

[20] Rodeheffer, T.L., Hibbard, P.G., "Automatic exploitation of parallelism on a homogeneous asynchronous multiprocessor", Intl. Conf. Par. Proc., 1980, pp. 15-16.

[21] Shampine, L.F. and Watts, H.A., "Block Implicit one-step methods", Math. Comp. 23, 1969, pp. 731-740.

[22] Ullman, J.D., "Polynomial complete scheduling problem", 4th Symp. Operat. System Principles, 1973, pp. 96-101.

[23] Worland, P.B., "Parallel methods for the numerical solution of ordinary differential equations", IEEE Trans. Comp. 25, 1976, pp. 1045-1048.

ANALYTICAL MODELS TO EXPLAIN ANOMALOUS BEHAVIOR OF PARALLEL ALGORITHMS

Bruce W. Weide
Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210

Abstract -- A probabilistic model of a class of parallel programs is used to investigate the counterintuitive behavior observed for some parallel algorithms. Two main points are made: (1) It may, in general, be beneficial to consider using more logical processes than physical processors in a parallel algorithm; and (2) Results from order statistics are useful tools in analyzing parallel systems.

1. Introduction

Certain strange phenomena have been reported recently regarding the behavior of parallel algorithms on real multiprocessors, such as C.mmp and Cm* [2,6,8,10], and an interesting problem is the development of models and analytical techniques to explain them [8,9]. Our goal here is to develop a realistic probabilistic model for describing how members of a class of "decomposable" problems behave when solved by certain parallel algorithms. Our contribution is not simply in the explanation of observed phenomena, but also in the introduction of order statistics as an analytical tool not ordinarily used in performance evaluation of computer systems.

Anomalous problems that have been reported in the literature can be classified into two categories. On the one hand are problems that are solved in parallel by decomposing them into a number of subproblems, the successful completion of any one of which solves the original problem. For instance, suppose it is necessary to search a table for the occurrence of an item known to be in the table somewhere. One possible algorithm is to search the positions of the table in random order. A search by n such processes, operating independently and in parallel, can be conducted, and the first process that finds the item is the one that determines the total running time.

Such an algorithm could, in theory at least, exhibit the following strange behavior. With one processor, the average solution time is T_1; with n processors and n independent processes, the average solution time is T_n; and $T_n < T_1/n$. In other words, n processors can exhibit a speed-up of the average time that is more than a factor of n.

A plausible explanation of such an apparently unlikely phenomenon is that the algorithm's running time is a random variable, having a distribution F(x), for which the expected value of the minimum of n observations is less than 1/n times the expected value of a single observation. For instance, $F(x) = x^{\delta}$, $0 \le x \le 1$, $0 < \delta < 1/2$, has the required property for all $n \ge 2$. This phenomenon is described in more detail in [8]. It should be noted that while such behavior is theoretically possible, we know of no practical algorithms for which it has been observed.

We consider here a different problem, where the decomposition is into a number of subproblems, the successful completion of all of which is required to solve the original problem. An example (in fact, the one that motivated development of the model proposed here) is a discrete optimization problem, such as integer programming, in which the space of possible solutions is partitioned into n disjoint subsets that are searched in parallel for the optimum feasible solution. The curious behavior here seems more believable than that described above, but still not entirely intuitive. It has been observed that average running times can sometimes be reduced by partitioning into more subproblems than there are processors, and by sharing the processors among the active subproblem-solving processes [8].

In late 1975, when the C.mmp multiprocessor at Carnegie-Mellon University [10] was configured with 5 PDP-11's sharing access to a single large memory, experiments with a parallel implementation of an implicit enumeration algorithm for 0/1 integer programming were conducted. In this problem, the goal is to

$$\text{minimize:}\quad c_0 + \sum_{j=1}^{n} c_j x_j$$

$$\text{subject to:}\quad \sum_{j=1}^{n} a_{ij} x_j \ge b_i,\ 1 \le i \le m$$

$$x_j \in \{0,1\},\ 1 \le j \le n.$$

A complete enumeration of the 2^n possible solution vectors can be avoided by making use of "branch-and-bound" techniques, but the general approach still looks much like a tree search: "branch" on x_1, say, and solve the two subproblems (each with n-1 variables) in which, respectively, x_1 is replaced by 0 and by 1 in the original problem. The subproblems are smaller instances of the same type problem, and can be solved by further division. If r variables are chosen initially for branching, there are 2^r subproblems, each with n-r variables, and they can be solved independently and in parallel. In practice, it improves the solution time if

Research supported in part by the National Science Foundation, MCS-79-12688.

0190-3918/81/0000/0183$00.75 © 1981 IEEE

certain global information (a bound on the objective function value) is shared, but there is no requirement for any interaction except for one final comparison of the subproblem's optimum solution value with the best solution found so far for another subproblem.

The experiments on C.mmp were not intended to determine how much speed-up could be obtained by parallel decomposition, but rather to exercise the C.mmp hardware and software in the system's early days of operation. Consequently, few precise timing runs were made. The limited experience offered by the parallel integer programming algorithm and the timings observed for it indicated, however, that even with only 5 processors, the average solution times tended to fall as the number of subproblems solved in parallel increased well beyond 5.

Figure 1 shows the behavior observed for a typical 20 variable, 20 constraint problem. Several runs were made with the r initial branching variables chosen at random, for r from 0 to 4, and the solution times averaged. The fact that average solution times tended to fall, even with 8, 16, and 32 subproblems being solved in parallel on only 5 processors, led to speculation regarding the underlying reasons for this phenomenon and to development of the analytical model described below.

Explanation of this behavior in a fairly realistic model, accounting even for overhead associated with processor sharing, is the purpose of this paper. Section 2 describes the hardware, scheduling, and problem models. Section 3 presents the analysis of the ideal (no overhead) case, and Section 4 extends it to include scheduling overhead. We conclude in Section 5 with a brief discussion of possible applications and future directions.

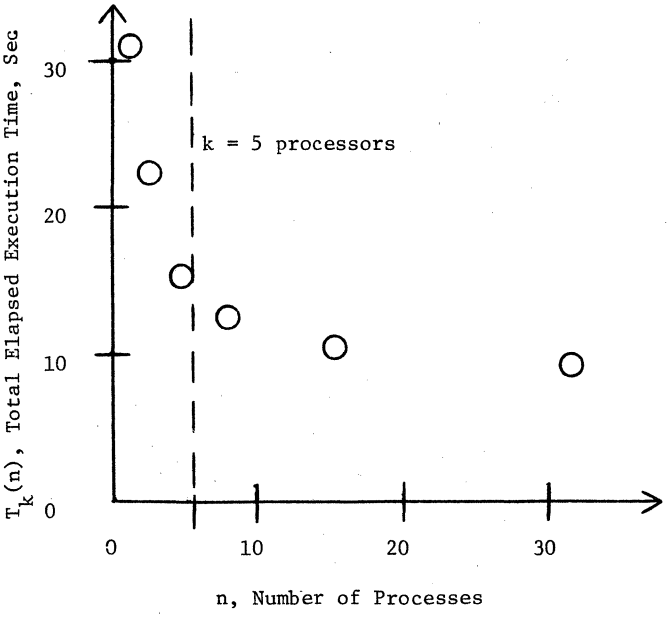


FIGURE 1 - Average execution times for a typical 0/1 integer programming problem on C.mmp system

2. The Model

A simple multiprocessor model is that shown in Figure 2. The k identical processors operate asynchronously and in parallel, and communicate via a shared memory. For simplicity we assume there is no contention for this memory (an assumption that is entirely reasonable for certain hardware configurations and reference properties), and that no overhead is involved in locking shared data for exclusive access. In short, each processor operates as fast as if it alone were executing without the other k-1 processors. This assumption is necessary to make the model at all tractable; analysis of contention effects is a difficult problem in its own right. In addition, we assume that the problem we are solving accounts for the entire computing load on the system; more about this later.

A schedulable entity is called a *process*. In our model, each subproblem solution is computed by a separate process, and each process is either *active* (i.e., not completed) or *inactive*. Whenever there are at most as many active processes as processors, each process is bound to one processor, and no scheduling is necessary. If there are more active processes than processors, scheduling is by *processor sharing*, which means that each process effectively has only a fraction of a processor's computing power. For instance, with k processors running n > k active processes, the computation of each process progresses at k/n times the rate it would progress if it had its own dedicated processor.

In Section 3, the analysis assumes that there is no overhead associated with processor sharing; in Section 4, we relax this restriction. It should be noted, however, that if the system is shared with other tasks, if our job is allocated a fixed percentage of the system resources, and if the system-wide scheduling policy is processor sharing, then the results of Section 3 hold *even though they do not account for overhead*. This is true because all running times are multiplied by the same constant,

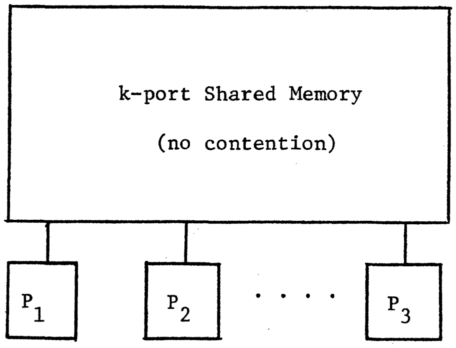


FIGURE 2 - k-processor model with shared memory

namely, the reciprocal of the fraction allocated to our job, since processor sharing takes place even when our job has fewer active processes than processors. Section 4, then, is necessary only because a dedicated multiprocessor system could refrain from processor sharing at that point, so the overhead would be paid only during a part of the computation.

The most controversial (i.e., unrealistic) aspect of our model is the problem model. We assume the problem to be solved can be decomposed into a number of subproblems, for parallel solution, with the following properties:

(1) The time required by the algorithm to solve a random instance of the problem on a single processor is a random variable X having the distribution F(x). We assume that $F(x) = 0$ for $x < 0$ since processing times are non-negative, and that $\mu = E(X)$ is finite.

(2) The problem can be solved by solving <u>all</u> of any finite number n of subproblems, each of which is of the same type as the original problem, but is probabilistically smaller (in solution time) by a factor of n. Therefore, the solution time for each subproblem on a single processor is a random variable having the distribution F(nx).

(3) The subproblem solution times are independent of each other and independent of the solution time of the original problem.

Property (2) seems questionable at first glance, but is in fact quite reasonable, especially for many numerical linear algebra problems, sparse matrix manipulations, discrete optimization problems, and queries in large data bases, for instance. Property (3) is unreasonable in most cases, but this is the price we must pay for the ability to get analytical results. Actually, the subproblem solution times may be almost independent if n is very large, or may truly be independent if randomness is induced by the algorithm itself [8].

The problem to be considered here is how to determine n such that the expected solution time on k processors is minimized. Each subproblem is allocated one process, and the total solution time is the elapsed time to completion of the <u>last</u> process that finishes, since all subproblems must be solved in order to solve the original problem. Although in the model n can be arbitrarily large, presumably any real problem can be subdivided only so far before the assumptions fail. A conclusion such as "make n as large as possible" means "make n as large as possible such that the assumptions (1) through (3) above are satisfied".

3. "No Overhead" Analysis

Let $T_k(n)$ be the total solution time on k processors when the problem is divided into n subproblems; let $X_{j:n}$ be the j^{th} smallest of n independent random variables from the distribution F(x); and let $\mu_{j:n} = E(X_{j:n})$. Define $Y_{j:n}$ to be the solution time of the j^{th} subproblem if each subproblem (process) had its own processor. Then $E(Y_{j:n}) = \mu_{j:n}/n$ because of property (2) in the problem model. Finally, let S_j be the elapsed time to completion of the j^{th} subproblem using processor sharing on k processors. Throughout this analysis we assume $n \geq k$, since it is clear that it is always worthwhile to have at least k subproblems.

In order to solve the problem, we must find an expression for $T_k(n)$. By definition, $T_k(n) = S_n$. Furthermore, note that $S_1 = (n/k)Y_{1:n}$ (since there are n active processes before time S_1 and each has k/n effective processors) and that

$$S_j = S_{j-1} + (Y_{j:n}-Y_{j-1:n})(n-j+1)/k$$

for $2 \leq j \leq n-k$. This follows from the fact that between the completion of the j-1st and j^{th} subproblems, there are n-j+1 active processes. Solving the recurrence we find that

$$S_{n-k} = Y_{n-k:n} + (1/k) \sum_{j=1}^{n-k} Y_{j:n} .$$

After time S_{n-k} there are at most k processes still active, so each has its own processor and there is no sharing. The time remaining until all finish is simply $Y_{n:n} - Y_{n-k:n}$, so

$$T_k(n) = S_n = Y_{n:n} + (1/k) \sum_{j=1}^{n-k} Y_{j:n}.$$

Taking expectations of both sides gives

$$E(T_k(n)) = \mu_{n:n}/n + (1/k) \sum_{j=1}^{n-k} \mu_{j:n}/n$$

which can be rewritten as

$$E(T_k(n)) = \mu/k + \mu_{n:n}/n - (1/k) \sum_{j=n-k+1}^{n} \mu_{j:n}/n.$$

Note that μ/k is the best value of $E(T_k(n))$ we could hope for, and that the expression above exceeds this by only

$$\sum_{j=n-k+1}^{n} (\mu_{n:n}-\mu_{j:n})/(kn),$$

which for most distributions F tends rapidly to zero as n increases. For example, for the exponential distribution

$$E(T_k(n)) = \mu(1/k + (\sum_{i=2}^{k}(1/i))/n);$$

and for the uniform distribution

$$E(T_k(n)) = \mu(1/k+(k-1)/(2n(n+1))).$$

It is clear that for a fixed number of processors $k \geq 2$, each of these is a decreasing function of n. What we need to show is that the same is true regardless of the underlying distribution F. In order to do this, we form $\Delta E(T_k(n)) = E(T_k(n)) - E(T_k(n-1))$, and then investigate its behavior for various values of $n > k$. This will allow us to determine for what values of n the expected total solution time is increasing and for which it is decreasing. If $\Delta E(T_k(n)) < 0$ then it is better to have n subproblems than n-1; otherwise, it is better to have only n-1.

Forming the expression for $\Delta E(T_k(n))$, then applying an identity from order statistics [4]

$$(n-k)\ \mu_{k:n} + k\mu_{k+1:n} = n\mu_{k:n-1}$$

to get all variables in terms of expected values of order statistics from a sample of size n, and finally simplifying the sums, gives the surprisingly simple expression

$$\Delta E(T_k(n)) = (\mu_{n-k:n} - \mu_{n-1:n})/(n(n-1))$$

for $n > k$. Since $\mu_{n-k:n} \leq \mu_{n-1:n}$ for any distribution, $\Delta E(T_k(n)) \leq 0$ for all $n > k$. In fact, unless $k = 1$ or the distribution F is degenerate and all problems have identical solution times, $\Delta E(T_k(n)) < 0$, which means that we should make n as large as possible to minimize the expected total solution time.

4. Accounting for Overhead

As users of real multiprocessor systems such as C.mmp well know, processor sharing is not implemented without overhead. Nevertheless, experiments on that system with parallel solution of integer programming problems led naturally to the model used and to the conclusion reached in Section 3. In this section, we explore the effects of overhead on tractability of the model and see why it is not a serious problem.

Fortunately, it is possible for our model to account for the overhead in a simple manner. An approximation to processor sharing is achieved by allowing each process to run for a small length of time (a "quantum") using round-robin scheduling of the active processes. The overhead is associated with "context swapping" from one process to the next. If we define c to be the ratio of the context swapping time to the quantum size, the total overhead incurred is cS_{n-k}, and

$$T_k(n) = S_n + cS_{n-k}$$

for $n > k$. This follows from the fact that sharing is necessary only so long as there are more than k active processes (see Section 2).

Proceeding in a fashion similar to that used above we find

$$\Delta E(T_k(n)) = (c(k+1)(\mu_{n-k:n} - \mu_{n-k-1:n}) +$$

$$\mu_{n-k:n} - \mu_{n-1:n}) / (n(n-1))$$

where $\mu_{0:n} = 0$ by definition.

It is clear that we cannot make the same strong statement that we should always create as many subproblems as possible, regardless of c and F(x). Obviously, for large values of c we should avoid creating more processes than we have processors in order to avoid processor sharing. What is not so obvious is how to compute the optimum number of processes we should create for given c and F.

It turns out that it is helpful to define a new quantity $\delta_{j:n} = \mu_{n-j+1:n} - \mu_{n-j:n}$ which is the expected value of the difference between the j^{th} and $j+1^{st}$ largest (not smallest, as before) of n random variables from the distribution F. Writing $\Delta E(T_k(n))$ in terms of $\delta_{n:n}$, we have

$$\Delta E(T_k(n)) = (c(k+1)\ \delta_{k+1:n} -$$

$$\sum_{j=2}^{k} \delta_{j:n}) \ /\ (n(n-1))$$

where $\delta_{n:n} = \mu_{1:n}$. This means that

$$\Delta E(T_k(n)) < 0 \text{ iff}$$

$$c < (1/(k+1)) \sum_{n=2}^{k} (\delta_{j:n}/\delta_{k+1:n}).$$

For certain forms of the distribution F there exist simple expressions for $\delta_{j:n}$, allowing us to compute $\Delta E(T_k(n))$ explicitly for certain cases. For example, for the exponential distribution $\delta_{j:n} = \mu/j$ and for the uniform distribution $\delta_{j:n} = 2\mu/(n+1)$. Therefore,

$$\Delta E(T_k(n)) < 0 \text{ iff}$$

$$c < \sum_{i=2}^{k} (1/i) \qquad \text{(exponential)}$$

$$\Delta E(T_k(n)) < 0 \text{ iff}$$

$$c < (k-1)/(k+1) \qquad \text{(uniform).}$$

Since these conditions are independent of n, we may still reach the (rather strong) conclusion that if c satisfies the appropriate condition above then we should create as many subproblems as possible. If c is too large, then we should create exactly k subproblems. These conditions on c are extremely weak, since the value of c for a real system might be on the order of a few percent, while $c < 1/3$ suffices here even for only two processors.

For most other distributions, no such closed-form expressions for $\delta_{j:n}$ are available. However, we can divide the possible values of n into three mutually exclusive and exhaustive ranges:

(1) $1 \leq n \leq k$: $\Delta E(T_k(n)) < 0$ always

(2) $n = k+1$: $\Delta E(T_k(n)) < 0$ iff

$$c < (\mu_{k:k+1} - \mu_{1:k+1})/((k+1)\mu_{1:k+1})$$

(3) $n > k+1$: $\Delta E(T_k(n)) < 0$ iff

$$c < (1/k+1)) \sum_{n=2}^{k} (\delta_{j:n}/\delta_{k+1:n}).$$

It is noteworthy that for $n > k+1$, the critical value of c (call it C) depends only on ratios of differences between expected values of order statistics. These ratios are, of course, distribution dependent, but are independent of location and scale parameters. Thus, if the distribution F is a gamma distribution, for instance, we can calculate the values of C for $n > k+1$ without knowledge of the actual mean and variance for F.

For the sake of argument, let us assume that the distribution of problem solution times is adequately represented by a gamma distribution. Tabulating the critical values for this distribution, we find that $c < 1/4$ is a sufficient condition for $\Delta E(T_k(n)) < 0$ for all $k \geq 2$ and $n > k+1$. Since c should be much smaller than that for a real system, we will assume that this condition is satisfied.

Now the only question is whether $\Delta E(T_k(k+1)) < 0$. If it is, then we should create as many subproblems as possible. If it is not, then we need to know whether the increase in solution time at $n = k+1$ can be offset by the known decreases thereafter. The second problem is easy, since $E(T_k(n)) \to \mu(1+c)/k$ as $n \to \infty$. Therefore, creating as many subproblems as possible is better than creating just k subproblems iff $c < \mu_{k:k}/\mu - 1$. This apparently makes the answer to the former problem irrelevant, for if $\Delta E(T_k(k+1)) < 0$ we would certainly have $c < \mu_{k:k}/\mu - 1$. Hence, we can conclude that if the ratio c of overhead to quantum size is at most 1/4, then the expected total solution time is minimized by letting $n = k$ whenever $c \geq \mu_{k:k}/\mu - 1$, and by making n as large as possible if $c < \mu_{k:k}/\mu - 1$.

The value of μ really only determines the time unit and may therefore be set to 1 without loss of generality. In this case, $\mu_{k:k} = V^2 z_{k:k}$, where $z_{k:k}$ is the expected value of the largest of k random variables from a gamma distribution with parameter $1/V^2$. The coefficient of variation of this distribution is V. In order to determine in practice how many subproblems to create we could estimate c and V from experimental data, look up $z_{k:k}$ in a table of expected values of order statistics [7], and decide on the basis of the criterion above.

5. Conclusions

Order statistics are a natural for analysis of many parallel algorithms, since total running times depend on those of the minimum or maximum running times of subproblem solutions. It is only reasonable that computer science should make good use of the large body of knowledge statisticians have already compiled regarding their behavior.

Other scheduling models lead to interesting problems. Coffman and Denning [3] note that, in general, processor sharing may be better or worse than a simple list schedule (in which the n processes queue up to the k processors, and when one finishes another begins). While it is possible to construct examples where each is superior, which is better on the average? Analysis of list schedules seems to require use of renewal theory rather than classical queueing theory. In any event, we feel we have made a good case for further exploration of the application of statistical methods to performance models of parallel algorithms.

6. References

[1] M. Abramowitz and I. A. Stegun, eds., Handbook of Mathematical Functions, Dover Publications, New York (1965).

[2] G. Baudet, The Design and Analysis of Algorithms for Asynchronous Mutliprocessors, Department of Computer Science, Carnegie-Mellon University, Ph.D. Thesis, (April 1978).

[3] E. G. Coffman and P. J. Denning, Operating Systems Theory, Prentice-Hall (1973).

[4] H. A. David, Order Statistics, Wiley, (1970).

[5] E. J. Gumbel, Statistics of Extremes, Columbia University Press, (1958).

[6] A. K. Jones and P. Schwarz, "Experience Using Multiprocessor Systems -- a Status Report," Comp. Surv. 12, (June 1980), pp. 121-125.

[7] E. S. Pearson and H. O. Hartley, eds., Biometrika Tables for Statisticians, Cambridge University Press, (1972).

[8] B. W. Weide, Statistical Methods in Algorithm Design and Analysis, Department of Computer Science, Carnegie-Mellon University, CMU-CS-78-142 (August 1978).

[9] M. V. Wilkes, "Beyond Today's Computers," Information Processing 77, (1977), pp. 1-5.

[10] W. A. Wulf and C. G. Bell, "C.mmp -- A Multi-mini-processor," Proc. FJCC 72, (1972), pp. 765-777.

PARALLEL ALGORITHMS FOR THE MINIMUM SPANNING TREE PROBLEM

Narsingh Deo and Year Back Yoo
Computer Science Department
Washington State University
Pullman, Washington 99164

Summary

Sequential algorithms for minimum spanning tree (MST) fall into three categories--all using greedy strategies. They are: (1) Prim-Dijkstra nearest-neighbor method, (2) Kruskal's lightest-edge-first method, and (3) Sollin's lightest-edge-from-each-vertex method. In this paper we study the parallelizability of these algorithms.

Recently some research effort has been reported on parallel algorithms for solving the MST problem [1], [4], [5]. Savage [4] proposed a parallel MST algorithm based on Sollin's, which runs in $O(\log^2 n)$ time on a parallel machine with $O(n^2/\log n)$ processors (where n is the number of vertices in the graph). Bentley [1] has given a parallel version of Prim-Dijkstra algorithm which runs in $O(n \log n)$ on a special-purpose parallel computer, called the tree machine.

In this paper, we design three parallel MST algorithms under the assumption that (i) the number of available processors is no more than n and that (ii) available machine is an MIMD-type general-purpose parallel computer.

PRIM-DIJKSTRA ALGORITHM: With weight matrix as the data structure used, sequential Prim-Dijkstra algorithm has a time complexity of $O(n^2)$. We parallelize this algorithm with p processes as follows: P processes are created. Each process takes n/p vertices and finds its nearest vertex in parallel. The processes are synchronized, and the nearest vertex is found. Then another set of p processes are created for updating. The processes are synchronized again and one iteration is complete. This parallel algorithm requires the same number of total iterations as the sequential one, but the work is divided among p processes. If we choose $p = \sqrt{n}$, the time complexity becomes $O(n^{1.5})$. This performance compares well with Bentley's $O(n \log n)$ time on n/log n - processor tree machine [1]. Both have processor-time product of $O(n^2)$. This also compares well with Savage's processor-time product of $O(n^2 \log n)$ [4]. The parallel Prim-Dijkstra algorithm may be described in an Algol-like language as follows:

```
Process MAIN
  T:= ∅;
  for i:= 1 to n do
  begin NEAR[i]:= 1; DIST[i]:= W[1,i] end;
  NEAR[1]:= 0; (* Start with vertex 1 *)
  while |T| < n-1 do (* edges in MST is n-1 *)
  begin
    Vmin:= ∞; syn:= 0;  (* syn : semaphore *)
    for i:= 1 to p do create TASK1(i);
    while syn < p do wait;
    T:= T ∪ (jj,NEAR[jj]); NEAR[jj]:=0; syn:=0;
    for i:= 1 to p do create TASK2(i);
    while syn < p do wait;
  end;

Process TASK1(j)
  ii:= 1;
  for i:= j to n by p do
    if NEAR[i] > 0 and DIST[i] < DIST[ii]
      then ii:= i;
  lock Vmin;
    if DIST[ii] < Vmin then
    begin jj:= ii; Vmin:= DIST[ii] end;
  unlock Vmin;
  lock syn; syn:= syn + 1; unlock syn;

Process TASK2(j)
  for i:= j to n by p do
  begin k:= NEAR[i];
    if k > 0 and W[i,k] > W[i,jj] then
    begin NEAR[i]:= jj; DIST[i]:= W[i,jj] end
  end;
  lock syn; syn:= syn + 1; unlock syn;
```

KRUSKAL'S ALGORITHM: With a heap as the data structure used, the time complexity of Kruskal's algorithm is O(m log m), where m is the number of edges. One way to parallelize Kruskal's algorithm is to use two processes, Producer and Consumer, which run asynchronously. A circular queue, Q, is used as a message buffer. The two processes operate as follows:

Producer maintains a min-heap and sends the top item, which is the next lightest edge to be considered, to Q. If Q is full, Producer waits until Consumer takes out an item from Q. As long as Q is not full, Producer continuously produces the next lightest edges and sends them one by one to the rear of Q.

Consumer takes out items continuously one by one from the front of Q as long as Q is not empty. Then it examines whether or not the current edge creates a cycle. If not, Consumer adds the edge to MST, combining the two subtrees.

Heap adjusting step cannot be done in parallel because of the inherent precedence constraint. Therefore, the complexity still remains O(m log m). Furthermore, the degree of parallelism is at most two. These limitations make parallel version of Kruskal's algorithm less attractive. This version is given below:

0190-3918/81/0000/0188$00.75 © 1981 IEEE

```
Process MAIN
  T:= ∅; syn:= 0; num:= 0;
  make initial heap;
  create PRODUCER;
  call   CONSUMER;

Process PRODUCER
  last:= m; rear:= 0;
  while CONSUMER is live do
  begin
    if Q is full then wait;
    rear:= (rear+1) mod b; (* b = |buffer| *)
    send top item of the heap to the rear of Q;
    lock num; num:= num + 1; unlock num;
    move last item of the heap to the top;
    last:= last - 1;
    call HEAP(1,last)     (* Adjust heap *)
  end;

Process CONSUMER
  while |T| < n-1 do
  begin
    if Q is empty then wait;
    front:= (front+1) mod b;
    u,v,w := Q[front];
    lock num; num:= num - 1; unlock num;
    r1:= FIND(u); r1:= FIND(v);
    if r1 <> r2 then (* (u,v) is in MST *)
    begin T:= T ∪ (u,v); call UNION(r1,r2); end
  end;
```

SOLLIN'S ALGORITHM: Sollin's algorithm [5]can be parallelized as follows: The lightest edges incident on each vertex are selected simultaneously. If the resulting forest does not form a spanning tree, the same procedure is applied to the forests until only one tree is formed. In the worst case this algorithm will require log n iterations with n processes. Each iteration requires $O(n^2/p)$ units of time if p processors are available. Therefore time complexity becomes $O(n^2/p \log n)$. Processor-time product is $O(n^2 \log n)$ which is the same as that of Savage's. A large transportation network is often in the form of a grid in which the degree of each vertex is four or less. Such a graph if stored in the forward star form would require at most three comparisons per vertex to find the lightest edge incident on it (rather than n-1). In that case the time complexity of this algorithm will be $O(3n/p \log n)$. A detailed description of the parallel Sollin's algorithm is given below:

```
Process MAIN
  T:= ∅;
  while |T| < n-1 do
  begin
    syn:= 0; Vmin:= ∞;
    for i:= 1 to p do create TASK(i);
    while syn < p do wait;
    for i:= 1 to n do
    begin
      if Vmin[i] <> ∞ do
      begin
        r1:= FIND(Vi[i]); r2:= FIND(Vj[i]);
        if r1 <> r2 then
        begin
          T:=T ∪ (Vi[i],Vj[i]); call UNION(r1,r2)
        end
      end
    end
  end;

Process TASK(ii)
  for i:= ii to n by p do
  for j:= 1 to n do
  begin
    r1:= FIND(i); r2:= FIND(j);
    if r1 <> r2 then
    begin
      lock Vmin[r1];
        if W[i,j] < Vmin[r1] then
        begin
          Vi[r1]:=i; Vj[r1]:=j; Vmin[r1]:= W[i,j]
        end;
      unlock Vmin[r1]
    end
  end;
  lock syn; syn:= syn + 1; unlock syn;
```

REMARKS and CONCLUSION: A parallel version of Cheriton and Tarjan's O(m log log n) algorithm turns out to be Sollin's. The reason is that Cheriton and Tarjan's algorithm was derived from Sollin's [2], [5].

Sollin's algorithm is easily parallelized; Kruskal's is not. Prim-Dijkstra algorithm falls in between the two.

These parallel algorithms have been coded in HEP Fortran and an empirical study is underway for comparing their average case performances. Details are given in [3].

References

[1] J.L. Bentley, "A parallel algorithm for constructing minimum spanning trees", J. of Algorithms (Jan. 1980), pp. 51-59.

[2] D. Cheriton and R.E. Tarjan, "Finding minimum spanning trees", SIAM J. Comput. (Dec. 1976), pp. 724-742.

[3] N. Deo and Y.B. Yoo, Parallel algorithms for the minimum spanning tree problem, Computer Science Department, Washington State University, CS-81-072 (Mar. 1981), 20 pp.

[4] C. Savage, Parallel algorithms for graph theoretic problems, Ph.D. Thesis, Mathematics Dept., Univ. of Illinois at Urbana-Champaign, Report ACT-4 (Aug. 1977).

[5] M. Sollin, An algorithm attributed to Sollin , in Introduction to the Design and Analysis of Algorithms, S.E. Goodman and S.T. Hedetniemi, Section 5.5, McGraw-Hill, (1977).

This work was supported by U.S. Dept. of Transportation contract #DOT-RC-92042 and by NSF grant #MCS78-25851.

PARALLEL IMAGE CORRELATION

Leah J. Siegel, Howard Jay Siegel, and Arthur E. Feather

Purdue University
School of Electrical Engineering
West Lafayette, IN 47907

Abstract -- Image correlation is representative of a wide variety of window-based image processing tasks. The way in which multimicroprocessor systems (e.g., PASM) can use SIMD parallelism to perform image correlation is examined. Two fundamental algorithm strategies are explored. The "time / space / interprocessor-transfer" complexities of the two algorithm approaches are analyzed in order to quantify the differences resulting from the two strategies. For both approaches, the asymptotic time complexity of the N-processor SIMD algorithms is (1/N)-th that of the corresponding serial algorithms.

1. INTRODUCTION

Image correlation is a widely used procedure in many areas of image and picture processing. This process, also known as template matching, is used in some forms of edge detection [13], or in image registration, to match pieces of two pictures to one another [12]. In digital photogrammetry, image correlation is used to find the corresponding points of two images of a stereomodel. In this application, image sizes are typically at least 4096 by 4096 with match areas on the order of 64 by 64.

Because image correlation requires comparing portions of two images in a large number of relative positions, it is an extremely time consuming process. The time required to complete the calculations can be reduced by exploiting the parallelism inherent in the task. The way in which multimicroprocessor systems (e.g., PASM [16]) can use "SIMD" parallelism to perform this task is examined here.

The SIMD (single instruction stream - multiple data stream) [8,20] machine model used here consists of a control unit, interconnection network, and N PEs (processing elements), where each PE is a processor-memory pair [15]. In an SIMD machine of size $N = 2^n$, the PEs are addressed (numbered) from 0 to N-1. In proposed systems, N is as large as 1024 [16] to 16,384 [11]. The control unit broadcasts an instruction to all PEs, and all active (enabled) processors simultaneously execute the instruction, each on data in its own memory. The interconnection network provides inter-PE communication. SIMD parallelism has been shown to yield significant reductions in computation time for image and speech processing tasks [e.g., 9,14,18]. Here, window-based image processing tasks are considered.

In the complexity analyses that follow, it is assumed that each required parallel inter-PE data move can be done in one transfer step. This will be true if the interconnection network used is a multistage network such as: (a) one employing the generalized cube topology with individual box control [17] (e.g., omega [10], n-cube [11]); (b) the data manipulator network [6]; or (c) the augmented data manipulator [17]. This is because each required transfer is either a type of exchange (cube connection [15]) or a "uniform shift" (i.e., from PE i to PE i+k mod N, $0 \leq i < N$, k fixed).

Only those SIMD machine features needed for the algorithms that follow have been described. The model is intended to provide a general framework in which SIMD algorithms can be developed. In section 6, the performance of the algorithms using an alternative model will be discussed.

The objectives of this study are as follows:

1. To demonstrate the applicability of the SIMD mode of parallelism to a class of image processing tasks. The operations performed in image correlation are representative of the types of data manipulations needed for a wide variety of window-based image processing tasks.
2. To explore two fundamental parallel algorithm strategies. In one approach, all of the data that will be needed by a PE is transferred to the PE and processed there. In the other, each PE performs all possible operations on its local data, generating partial results which are then transferred to the PE in which they are needed.
3. To analyze and compare the computational requirements of the alternative algorithms. In serial algorithms, there is often a tradeoff between computation time and space. In parallel algorithms, the tradeoff may be a function of three parameters: computation time, space, and inter-PE communications.

In the next section, image correlation is defined. In the subsequent sections, parallel algorithms for image correlation are presented and analyzed.

2. IMAGE CORRELATION

A. Definition and Serial Algorithms

An image is represented by a two-dimensional array where each element ("pixel") has an unsigned integer value representing the "gray level" of the pixel. Image correlation involves determining the

This work 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 U.S. Air Force Rome Air Development Center Information Sciences Division under Contract No. F30602-78-C-0025 through the University of Michigan. The United States Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation hereon.

0190-3918/81/0000/0190$00.75 © 1981 IEEE

position at which a relatively small match area best matches a portion of an input image. Correlation measures are used to measure the degree of similarity or disagreement between the match area and an equivalent size area on the input image. Let the symbols x and y denote single elements of arrays X and Y, where X is the match image and Y is an area of the input image which has the same dimensions as X. Let M be the number of elements in the match area X. Two representative correlation measures are:

$$SXY = \Sigma xy - \Sigma x\ \Sigma y/M$$

$$RXY = SXY/(SXX*SYY)^{(1/2)}$$

Correlation measure SXY is the covariance of the match area with a portion of the input area. Large positive values indicate similarity, while large negative values indicate similarity between a positive and a negative image. Values near zero indicate little or no similarity. Correlation measure RXY is the linear correlation coefficient of statistics. This measure is a normalized version of SXY, with values ranging between +1 and -1. A value of +1 indicates exact similarity while values near zero indicate little similarity. In general, a correlation value will be computed for every possible position where the match area will fit on the input image. The match position where the correlation measure is maximized corresponds to the best placement of the match area on the image.

The computation time for image correlation is dominated by the time to compute the Σxy, Σy, and (for measure RXY) the Σy^2 values for all possible match positions. The Σx and Σx^2 values involve only the match area elements, and need to be computed (or precomputed) only once.

The way in which data elements are combined to obtain the Σxy values is similar to operations performed in a variety of important image processing tasks, including convolution and filtering. For an input image having R rows and C columns and a match area having r rows and c columns, there are (R-r+1)(C-c+1) match positions. Serial computation of the Σxy terms over the entire image, performed by simply sliding the match area over the image and calculating the value of Σxy for each overlap position, requires (R-r+1)(C-c+1)rc multiplications and (R-r+1)(C-c+1)(rc-1) additions.

In computing the Σxy values, each match position generates a new set of terms to be summed. No terms from one match position can be reused in a different match position. In computing the Σy and Σy^2 values, two (or more) input image elements summed for one match position may also be summed for another match position. The algorithms considered for calculating the Σy and Σy^2 values therefore attempt to avoid "redundant" operations, e.g., performing a sum for one match position which has already been performed for another. The operations performed in computing the Σy and Σy^2 values, i.e., the summing of elements under a window where the window moves over an image, are typical of operations required for a variety of image processing tasks. These include image smoothing, edge enhancement, and convolution using a rectangular window.

Consider the following serial (uniprocessor) algorithm for computing the Σy's, i.e., summing the pixel values in each match area. This algorithm will be used as a basis for parallel algorithms.

Assume that for input image I, the position of the match area is defined by the coordinates of the input image pixel covered by the upper left corner of the match area. Let "colsum" be a vector of length C, where

$$\text{colsum}(j) = \sum_{i=k}^{k+r-1} I(i,j)$$

where k is the row coordinate of the current position of the match area, and $0 \le j < C$. Let SUM be an R-r+1 by C-c+1 array, where SUM(i,j) is the sum of pixels of I for the match area position (i,j), $0 \le i < R-r+1$, $0 \le j < C-c+1$.

The algorithm is shown in Fig. 1. First, colsum is initialized for row 0 of the image. The colsum values for columns 0 to c-1 are summed to compute SUM(0,0). SUM(0,j) for $1 \le j < C-c+1$ is computed from SUM(0,j-1) by subtracting colsum(j-1) and adding colsum(j+c-1). A similar strategy is used to compute SUM(i,j) for $1 \le i < C-c+1$ and $1 \le j < R-r+1$. To do this, each colsum(j) is first updated by subtracting I(i-1,j) and adding I(i+r-1,j), $0 \le j < R-r+1$.

The complexity of this serial algorithm, in terms of additions, is

$$4RC - Rc - 3Cr + rc + 5C + 3R - 2c - 3r + 4.$$

(For simplicity, the additions required for loop counting and indexing have not been included. In the SIMD algorithms, these would be performed in

```
/* initialize values of colsum */
for j = 0 to C-1 do
    colsum(j) = I(0,j)
    for i = 1 to r-1 do
        colsum(j) = colsum(j) + I(i,j)
/* compute SUM(0,j) for 0 ≤ j < C-(c-1) */
SUM(0,0) = colsum(0)
for j = 1 to c-1 do
    SUM(0,0) = SUM(0,0) + colsum(j)
for j = 1 to C-(c-1) do
    SUM(0,j) = SUM(0,j-1) - colsum(j-1) + colsum(j+c-1)
/* compute SUM(i,j) for 1 ≤ i < R-(r-1)
    and 0 ≤ j < C-(c-1)) * /
for i = 1 to R-(r-1) do
    /* compute SUM(i,0) and update associated
        colsum values */
    for j = 0 to c-1 do
        colsum(j) = colsum(j) - I(i-1,j) + I(i+r-1,j)
    SUM(i,0) = colsum(0)
    for j = 1 to c-1 do
        SUM(i,0) = SUM(i,0) + colsum(j)
    /* compute SUM(i,j) and update associated colsum
        values for 1 ≤ j < C-(c-1)) */
    for j = 1 to C-(c-1) do
        colsum(j+c-1) = colsum(j+c-1) - I(i-1,j+c-1)
                                  + I(i+r-1,j+c-1)
        SUM(i,j) = SUM(i,j-1) - colsum(j-1)
                                  + colsum(j+c-1)
```

Fig. 1: Serial algorithm to compute Σy terms.

the control unit, and could be overlapped with the PE operations.) This algorithm moves the match area along the rows of the input image. Depending on C, R, c, and r the algorithm complexity may be less by moving along columns.

Computation of the Σy^2's is similar. In this case, the y^2 values subtracted from the colsum's in the update process (see Fig. 1) must be saved when they are first calculated. This increases the space required for the algorithm by rC. The arithmetic complexity is increased by RC multiplications.

If the Σxy, Σy and Σy^2 values for a given match position are computed together, the correlation measure for that match position can be calculated, and is saved only if it is the current maximum over the correlation measure values computed so far. Thus, the Σxy, Σy, and Σy^2 values for each position do not have to be saved.

B. Parallel Image Correlation

In section 3, a parallel algorithm for computing the Σx and Σx^2 values is given. In sections 4 and 5, parallel algorithms for the Σxy, Σy and Σy^2 computations are presented. For the Σxy, Σy and Σy^2 operations, two algorithm strategies are explored. For both, the input image data will be divided among the PEs, and each PE will compute the values of the correlation measure for a portion of the input image. In the first, "complete sums" approach, all of the data which will be needed for the computations performed in a given PE is transferred into that PE. All subsequent operations can then be performed locally, so that each PE computes the "complete sums" for a set of match positions. In the second, "partial sums" approach, each PE performs as much of the computations as possible using its own data, then transfers partial results to the PE in which they are needed.

In order to distribute the input image, the N PEs of the system are logically configured as an NR by NC rectangular grid, on which the R by C image is superimposed. Thus, with the possible exception of the rightmost column and bottommost row of PEs, each PE holds an R' by C' subimage, where $R' = \lceil R/NR \rceil$ and $C' = \lceil C/NC \rceil$. This is shown in Fig. 2. The values for NR and NC will be chosen to minimize execution time of the algorithms, and will be discussed in section 4.A.

(An alternative to these approaches is to assume that the SIMD machine has the capability to load the image data into several PEs simultaneously. With this capacity, an element of the input array which is needed in several PEs could be loaded into the appropriate PEs (with little or no cost) simultaneously. This would eliminate the need for inter-PE transfers. However, the memory management necessary to place each image point in the appropriate location in each PE may be significantly more complex than the memory management needed to load the PE memories with disjoint subimages. This approach will require an "intelligent" memory management system and more storage in each PE, and will not be considered here.)

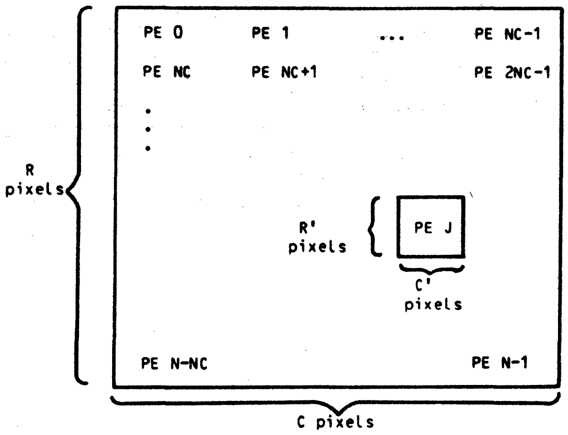


Fig. 2. Data assignment of R by C image to N PEs.

In the algorithms to compute the Σxy, Σy and Σy^2 values, it will initially be assumed that the results calculated (i.e., the Σxy, Σy and Σy^2 values) are saved. For the calculation of RXY and SXY, this will not be necessary, as will be described in subsections 4.C and 5.C. However, so that each of the Σxy, Σy, and Σy^2 algorithms can be applied to other related computations, in the presentations it will be assumed that the results for the whole image are to be stored.

3. Σx AND Σx^2 COMPUTATION

The Σx and Σx^2 values may be precomputed and stored with the match area, or computed in a straightforward manner in parallel before calculating the Σxy, Σy, and Σy^2 values. Simply assign to each PE M/N of the match area pixels. Each PE first computes x^2 for all the elements it holds. It then sums its x values and sums its x^2 values. All of these local Σx and local Σx^2 sums are then combined using a recursive doubling approach [19]. Each even numbered PE J sends its local Σx result to PE J+1. Simultaneously, each odd numbered PE J+1 sends its local Σx^2 to PE J. The odd numbered PEs add the received data to their local Σx and then compute the whole Σx using recursive doubling, with the result saved in each odd numbered PE. Similarly, the even numbered PEs compute Σx^2. These two recursive doublings can occur simultaneously. The odd and even PEs then exchange results, so that each PE contains both Σx and Σx^2. This requires M/N multiplications, n+(2M/N)-2 additions, and n+1 inter-PE data transfers. (A serial algorithm will require M multiplications and 2M-2 additions.) Each PE will store Σx and Σx^2 for later use.

4. COMPLETE SUMS APPROACH

A. Σxy Computation

In the complete sums approach, each PE will compute the correlation measure for overlap posi-

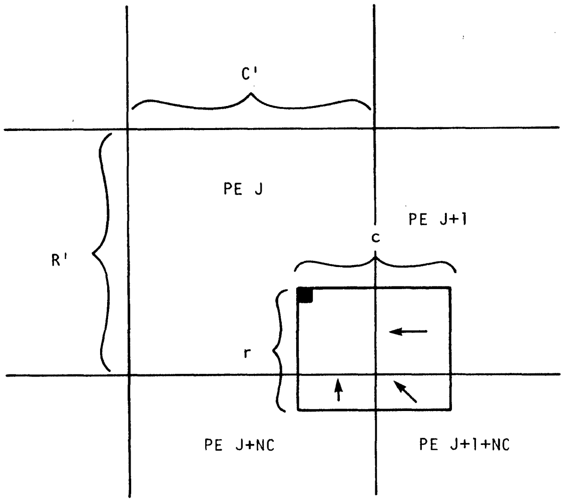


Fig. 3. Example of overlap position requiring data transfers. The shaded pixel represents the "beginning" of the overlap position. The arrows indicate the directions of the data transfers. (Proportions of match area to a PE's subimage are not necessarily to scale.)

tions which "begin" in the PE (i.e., for which the upper left corner of the match area overlaps a point of the PE's subimage). Each PE will therefore compute the correlation measure for $R'C'$ match positions. The computations will be performed simultaneously in all PEs. For match positions where the portion of the input image is not fully contained in a single PE (Fig. 3), the needed points will be transferred before the computations are performed. Such transfers will occur simultaneously for all PEs, so that at the same time that a pixel is being transferred, for example, from PE J+1 to PE J, the corresponding pixel is being transferred from PE J+2 to PE J+1, from PE J+3 to PE J+2, and so on.

Depending on the size relationships between r and R' or c and C', the transferred elements may come from PEs adjacent to PE J, or from several levels of adjacent PEs. If, for example, the match area dimension in one direction is large in comparison to the dimension of the portion of the input area stored in each PE, the matches will extend over several PE areas in that direction. PE J will transfer some y values a distance greater than one, and will receive some y values from a distance greater than one. Without loss of generality, in the subsequent discussions, it will be assumed that elements are needed only from adjacent PEs.

When computing the Σxy values, all PEs will use the same match area element simultaneously, so that element can be broadcast to all PEs from the control unit. Alternatively, if PE memory space is available, the match area, which is typically small, can be held in each PE's memory. The time to perform the broadcast from the control unit versus the memory fetch from the PE memory will be implementation dependent. In the space analyses that follow, it will be assumed that the match area values are broadcast from the control unit.

	Complete Sums	Partial Sums
# mult steps	$R'C'rc$	$R'C'rc$
# add steps	$R'C'(rc-1)$	$R'C'(rc-1)$
# transfer steps	$R'(c-1)+C'(r-1)$ $+(r-1)(c-1)$	$R'(c-1)+C'(r-1)$ $+(r-1)(c-1)$
space	$2R'C'+(2c-2)(r-1)+1$	$2R'C'$

Table 1: Complexity of complete sums and partial sums algorithms for computing Σxy terms.

Computation of all of the Σxy terms will be accomplished in the time required to compute the Σxy terms for the R' by C' subimage held in a single PE. These times are summarized in the first column of Table 1. Storage will be required for the PE's portion of the input image ($R'C'$ elements), for the computed Σxy values ($R'C'$ elements), and for the input image elements transferred to the PE in order to provide all of the data needed for the PE's match positions. The number of transferred elements is $(c-1)R' + (r-1)C' - (r-1)(c-1)$; however, it is not necessary to store all of these values at the same time. Consider the extra storage needed for non-local y values by a typical ("non-edge") PE J. The analysis is divided into two cases. It will show that at any point in time at most $(2c-2)(r-1)+1$ locations are required.

First, consider when the match area (upper left corner) is positioned in row i, $0 \leq i \leq R'-r$ (see Fig. 4). $(c-1)r$ locations are required for non-local y data, for the y data for columns 0 to $c-2$ of rows i to $r+i-1$ of PE J+1's subimage. The Σxy values can be calculated by moving the match area from position (0,0) to (0,1) to ... (0,C'-1), then from (1,0) to (1,1) to ... (1,C'-1), and finally from (R'-r,0) to (R'-r,1), to ... (R'-r,C'-1).

Next consider when the match area is positioned in row i, $R'-r < i < R'$ (see Fig. 4). In this case, at most $(2c-2)(r-1)+1$ locations are required for non-local y data. For these match po-

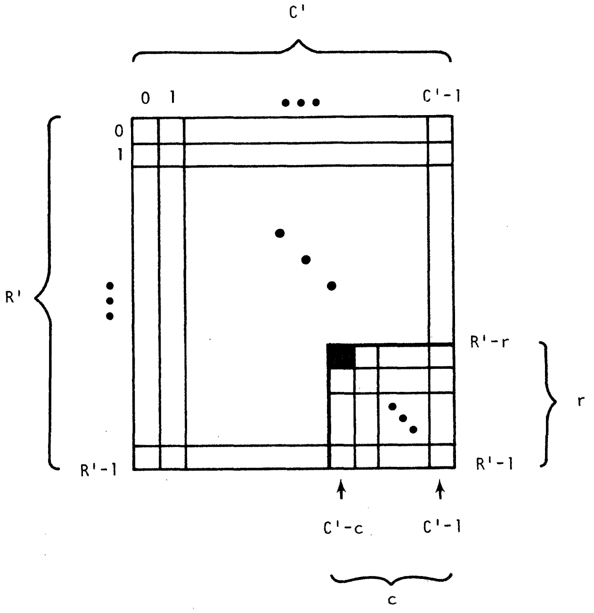


Fig. 4. Indexing in a PE's subimage.

sitions the match area will move along columns instead of rows, from $(R'-r+1,C'-1)$ to $(R'-r+2,C'-1)$ to ... $(R'-1,C'-1)$, then from $(R'-r+1,C'-2)$ to $(R'-r+2,C'-2)$ to ... $(R'-1,C'-2)$, ... finally from $(R'-r+1,0)$ to $(R'-r+2,0)$ to ... $(R'-1,0)$. For match positions (i,j), where $R'-r < i < R'$, and j is fixed at a value in the range $0 \leq j \leq C'-c$, the non-local y data needed are rows 0 to $r-(R'-i)-1$ of columns j to $j+c-1$ of PE J+NC's subimage. For match positions (i,j), where $R'-r < i < R'$, and j is fixed at a value in the range $C'-c < j < C'$, the non-local y data needed are rows i to $R'-1$ of columns 0 to $c-(C'-j)-1$ of PE J+1's subimage, rows 0 to $r-(R'-i)-1$ of columns j to $C'-1$ of PE J+NC's subimage, and rows 0 to $r-(R'-i)-1$ of columns 0 to $c-(C'-j)-1$ of PE J+NC+1's subimage. The maximum non-local y storage needed for this range of i and j is $(2c-2)(r-1)+1$.

For given c, r, C, R, and N, the number of arithmetic operations required for the algorithm is minimized by minimizing $Ip = R'C'$. By choosing $Ip = RC/N$, i.e., by dividing the input equally among the PEs, this minimum is attained. The number of transfer steps will be minimized by the values of C' and R' for which the expression

$$(r-1)C' + (c-1)R'$$

is minimized. Minimizing with respect to R' gives

$$R' = ((r-1)*Ip/(c-1))^{(1/2)}$$

subject to the constraints that R' and Ip/R' be integers. It will follow that

$$C' = ((c-1)*Ip/(r-1))^{(1/2)}.$$

In the special case where c = r, the image should be distributed such that

$$C' = R' = Ip^{(1/2)},$$

that is, each PE should contain a square subimage.

B. Σy and Σy^2 Computation

The complete sums algorithms to compute Σy and Σy^2 values will be based on the serial Σy and Σy^2 algorithms, with each PE operating on an $(R'+r-1)(C'+c-1)$ subimage.

Consider computing Σy and Σy^2 in a typical ("non-edge") PE J. A total of $(r-1)C'+(c-1)R'+rc-1$ y values must be transferred into the PE from adjacent PEs, as discussed in the previous subsection. The transfers are as shown in Fig. 3. However, it is not necessary to store all of these if data is transferred only when it is first needed. This is explained below in two cases. It will be shown that at most $(c-1)r$ locations will be required at any point in time.

When the match area is positioned in row i of the PE's subimage, $0 \leq i \leq R'-r$, $(c-1)r$ storage locations are required for non-local y data, for the y data for columns 0 to c-2 of rows i to r+i-1 of PE J+1's subimage.

When the match area is positioned in row i of the PE's subimage, $R'-r < i < R'$, at most $(c-1)(r-2)$ locations are required for non-local y data. Most y data can be incorporated into the current Σy being computed and the appropriate "colsum" vector location when it is transferred into a PE. The only y data that needs to be saved is that which will be needed for later "colsum" updates. Specifically, this is rows $R'-r+1$ to $R'-2$ of columns 0 to c-2 of PE J+1.

Using these data storage strategies, the Σy and Σy^2 values for each match position can be calculated as described in the serial algorithms (for a $(C'+c-1)(R'+r-1)$ image). The complexities for the Σy and Σy^2 computations are given in column one of Tables 2 and 3 respectively.

	Complete Sums	Partial Sums
# add steps	$4R'C'+3R'c+C'r$ $-R'+C'+rc$ $-r$	$4R'C'+R'c+3C'r$ $-3R'-5C'+rc$ $-3r-c+3$
# transfer steps	$R'(c-1)+C'(r-1)$ $+(r-1)(c-1)$	$R'(c-1)+C'(r-1)$ $+(r-1)(c-1)$
space	$2R'C'+(C'+c-1)$ $+(c-1)r$	$2R'C'+C'$

Table 2: Complexity of complete sums and partial sums algorithms for computing Σy terms.

	Complete Sums	Partial Sums
# mult steps	$(R'+r-1)(C'+c-1)$	$R'C'$
# add steps	$4R'C'+3R'c+C'r$ $-R'+C'+rc$ $-r$	$4R'C'+R'c+3C'r$ $-3R'-5C'+rc$ $-3r-c+3$
# transfer steps	$R'(c-1)+C'(r-1)$ $+(r-1)(c-1)$	$R'(c-1)+C'(r-1)$ $+(r-1)(c-1)$
space	$2R'C'+(C'+c-1)(r+1)$	$2R'C'+C'(r+1)$

Table 3: Complexity of complete sums and partial sums algorithms for computing Σy^2 terms.

C. RXY and SXY Computation

To compute RXY (or SXY) the previously described operations are interleaved so that the Σxy, Σy, Σy^2, and RXY (SXY) values for one match position are computed before the match area is moved to a new position. The maximum RXY (SXY) value and its match position coordinates are saved. The computation of RXY is described; the SXY computation is a subset of those operations.

Consider the computation performed in a typical ("non-edge") PE J. In order to combine the algorithms of subsections 4.A and 4.B, the Σxy algorithm must be slightly modified. The match area will move over the image in the way that was described in the Σy algorithm, that is, from position (0,0) to (0,1) to ... $(0,C'-1)$, then from (1,0) to (1,1) to ... $(1,C'-1)$, ... finally from $(R'-1,0)$ to $(R'-1,1)$ to ... $(R'-1,C'-1)$. The worst case for space is for $0 \leq i \leq R'-r$, when $r(C'+c-1)$ space is needed for y^2 values and $r(c-1)$ for y values (plus "colsums" and the original image). Less space is needed when $R'-r < i < R$ because space is not needed for non-local y^2 values.

Column one of Table 4 summarizes the total time, transfers, and space used. The time is a summation of that for computing Σxy, Σy, and Σy^2 for every match position. The transfers are for

	Complete Sums	Partial Sums
# mult steps	those for Σxy and Σy^2	those for Σxy and Σy^2
# add steps	those for Σxy, Σy, and Σy^2	those for Σxy, Σy, and Σy^2
# transfer steps	R'(c-1)+C'(r-1) +(r-1)(c-1)	3[R'(c-1)+C'(r-1) +(r-1)(c-1)]
space	R'C'+(C'+c-1)(r+2)+r(c-1)	R'C'+r(3C'+2c-2)

Table 4: Complexity of complete sums and partial sums algorithms for computing RXY. In addition to the above, each approach uses 2 subtractions, 3 multiplications, 2 divisions, and 1 square root operation for each of the R'C' match positions in order to combine terms. Both methods also require O(n) additional transfers for determining the maximum RXY value (and its coordinates in the input image) over all PEs.

the non-local y data needed. The space is for the PE's own subimage, the non-local y storage described above, and the extra storage used for intermediate results in calculating Σy and Σy^2.

Once each PE has found its own maximum RXY value, recursive doubling [19] can be used to find the overall maximum and its location. This will require O(n) additional inter-PE transfers.

5. PARTIAL SUMS APPROACH

A. Σxy Computation

The partial sums procedure for computing the Σxy values consists of three steps. The first step is the generation of partial sums by performing all parts of the calculation that can be done using the data within each PE. In the second step, the results of the partial sums generation are transferred so that each PE contains all of the partial sums needed to form the Σxy terms. In the last step, the final sums are developed within each PE by combining the appropriate partial sums. The details for this procedure follow. It is assumed that the match area elements are either broadcast from the control unit or stored in each PE's memory, as was discussed in subsection 4.A.

In the first step of the algorithm, each PE, independently of the others, computes the "partial sums" of match point-image point products that can be computed with its own data. This can be visualized by sliding the match area M over the image area in each PE, as shown in Fig. 5. At each match area-image area position from Fig. 5, a "partial sum" is generated. For each location where an image point and match point overlap in a given position, the product of the image and the match points is calculated; all the products for that match area-image area position are then summed. The partial sum terms generated by this procedure can be viewed as forming a (R'+r-1) by (C'+c-1) array called "psum." In Fig. 5, the element of "psum" into which the partial sum is stored is given for each of the example overlap positions. A match position will again be numbered by the input subimage coordinates (i,j) of the upper left corner of the match area. Since the match area may not be contained in the input image area however, the ranges of i and j differ from those in the serial and complete sums algorithms. If the upper left corner of the input subimage is considered to be position (0,0), -r < i < R' and -c < j < C'. The number of partial sums that must be computed in each PE is (r+R'-1)(c+C'-1). To develop these terms, every

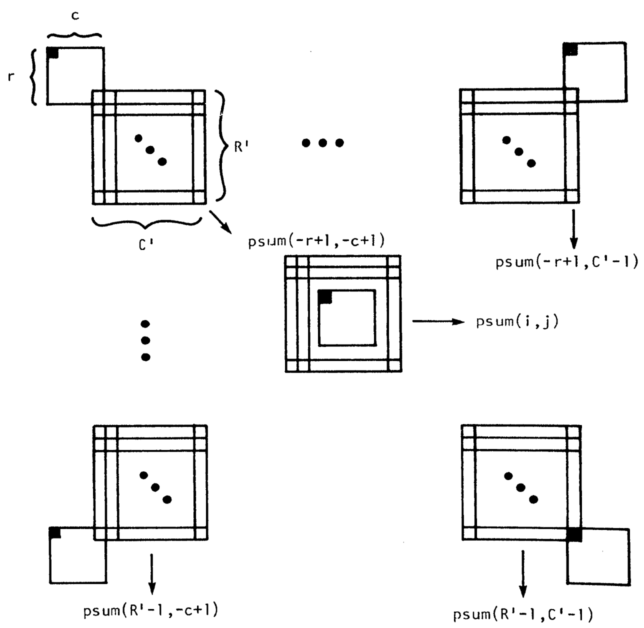


Fig. 5. Overlap positions in the partial sums approach, and the terms of the partial sums (psum) array calculated.

element of the match area M will be multiplied by every element of the input area in the PE. Therefore the number of multiplications required is rcR'C'. The number of additions required is equal to the number of multiplications minus the number of terms generated, or rcR'C' - (r+R'-1)(c+C'-1).

Once the partial sums have been computed independently in the PEs, it is necessary to combine the results from other PEs to build the complete sums. Using the criterion that the upper left corner element of the match area must be present in a partial sum for it to remain in a PE, the terms in the rightmost C' columns and bottommost R' rows of the "psum" array are kept in the PE, and are labeled "KEEP" in Fig. 6. Those elements "above" the kept area are transferred to PEs "above" this PE. Similarly, those elements "to the left" of the kept area are transferred to PEs "to the left," and the terms on the upper left are transferred to PEs diagonally above and to the left. As in the complete sums approach, the distance which elements will be transferred depends on the size relationships between r and R' and c and C'. The number of interprocessor transfers which will be required is equal to the total number of "psum" terms generated minus the number of terms kept (which is the number of image area points originally in each PE). Thus, the number of transfers required is (r+R'-1)(c+C'-1) - R'C'.

In the final step of this method, the partial sums transferred are combined with the partial sums that were kept to yield the final sums Σxy. The number of additions required to complete these calculations is equal to the number of partial sum terms that were transferred.

Rather than implementing the partial sums method as three separate steps, less space is required if the three steps for a given match position are executed in sequence. As soon as a

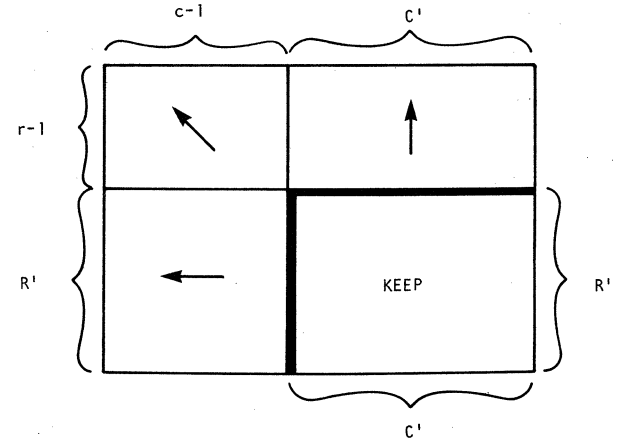


Fig. 6. Partition and direction of transfer of elements of partial sums array.

non-"kept" partial sum is computed, it can be transferred to its destination PE and saved in the memory location which will eventually hold the Σxy term of which it is a part. The execution time remains the same, and the only storage that is needed in each PE is two R'C' element arrays, one for the input image and one for the Σxy values.

B. Σy and Σy^2 Computation

The partial sums algorithms for computing the Σy and Σy^2 values are similar in strategy to the partial sums method for computing the Σxy values. Each PE computes Σy or Σy^2 terms for all match positions or portions of match positions for the R' by C' subimage which it contains. The partial sums Σy and Σy^2 algorithms are based on the serial Σy and Σy^2 algorithms. As in the serial algorithms, a C'-element vector "colsum" is used to save the column sums computed so far. After processing of row k, $-r < k < R'$,

$$\text{colsum}(j) = \begin{cases} \sum_{i=0}^{k+r-1} I(i,j) & -r < k \leq 0 \\ \sum_{i=k}^{k+r-1} I(i,j) & 0 < k \leq R'-r \\ \sum_{i=k}^{R'-1} I(i,j) & R'-r < k < R' \end{cases}$$

where $0 \leq j < C'$. Unlike the serial and complete sums algorithms, for each row, the leftmost sum consists of a single column sum, and for $-c+1 < j \leq 0$, the sum for position (i,j) is computed by adding colsum(j+c-1) to the sum for position (i,j-1). Similarly, for $C'-c < j < C'$, the sum for position (i,j) is obtained by subtracting colsum(j-1) from the sum for position (i,j-1). The sums (and colsums) for the topmost and bottommost c rows are computed in an analogous manner. In the "center" of each PE's subimage, the operations performed are identical to those in the serial and complete sums algorithms. The number of additions performed to generate the partial sums in each PE will be $4R'C' + 2C'r - 2R' - 4C' - 2r + 2$. As for the partial sums Σxy algorithm, the results which must be transferred are those in the non-"KEEP" area in Fig. 6. Each of these elements is added to a "kept" partial sum in the appropriate PE. The complexities of the partial sums Σy and Σy^2 algorithms are summarized in column two of Tables 2 and 3.

C. SXY and RXY Computation

As described for the complete sums method in subsection 4.C, for image correlation measures RXY or SXY, the Σxy, Σy, and Σy^2 computations will be interleaved so that all three are computed for a given match position before the match area is moved to a new position. The computation of RXY is described.

The modifications required for the RXY computation involve the storage for partial (incomplete) Σxy, Σy, and Σy^2 sums. In the algorithms described, a non-"kept" partial sum was transferred from the PE in which it was computed to its destination PE, and stored in the memory location for the Σxy (or Σy or Σy^2) of which it was a part. For the complete RXY computation, space is not needed for all of a PE's local Σxy, Σy, and Σy^2 values. Provisions must therefore be made for the incomplete sums. The procedure will be based on the Σy algorithm. It will be explained in terms of three cases (ranges of match positions). It will be shown that at most $2[(C'+c-1)(r-1)+c-1]$ locations are required to hold incomplete sums at any point in time.

Consider first a match position which is fully contained in the PE's subimage, i.e., position (i,j) where $0 \leq i \leq R'-r$ and $0 \leq j \leq C'-c$. The Σxy, Σy, and Σy^2 computations can be interleaved, and RXY for the position can be computed. For the same i range, when j exceeds C'-c, (i.e., $C'-c < j < C'$) two partial sums must be combined to produce the complete sum for each of Σxy, Σy, and Σy^2. After computing the Σxy partial sum for position (i,C'-c+k), $1 \leq k < c$, each PE can compute the partial sum for position (i,-c+k) and transfer its value to the left, where the total Σxy sum for position (i,C'-c+k) is then completed. By postponing computation of Σxy for position (i,-c+k) until it is needed, no extra space is required for row i's incomplete Σxy values. A comparable savings in space cannot be realized in the Σy and Σy^2 computations, since the computation of these partial results cannot be postponed (without doing additional summations). Except for the topmost and leftmost edges, the Σy (and Σy^2) sum for each match position is computed in terms of the sum for a previous match position, so (1) some previous results must be saved, and (2) the order in which the sums are computed cannot be altered or interrupted. The partial sum for position (i,-c+k), $1 \leq k < c$, must be computed and saved until it can be combined with the partial sum for position (i,C'-c+k). The same c-1 locations can be used for each row i in the range. Thus, c-1 locations are needed to save partial Σy results, and c-1 locations for partial Σy^2 results for match positions (i,j) where $0 \le i \le R'-r$ and $-c+1 \le j < 0$.

Similarly, for partial match positions along the top of the PE's subimage, where the row index is $-r+k$, $1 \le k < r$, computation of partial Σxy sums can be postponed until they are needed, but the partial Σy and Σy^2 sums must be computed and saved until the corresponding Σy and Σy^2 sums for row $R'-r+k$ have been calculated. Here, separate storage locations are needed for each row i, $-r < i < 0$, and column j, $-c < j < C'$. Therefore, for each of Σy and Σy^2, $(C'+c-1)(r-1)$ additional locations will be needed.

The partial sums complexity for computing RXY is summarized in Table 4. The time is a summation of that for computing Σxy, Σy, and Σy^2 for every match position. The transfers are for the non-"kept" partial sums. The space is for the PE's own subimage, the incomplete Σy and Σy^2 values which must be saved until they can be completed, and the intermediate results in calculating Σy and Σy^2.

As in the complete sums method, recursive doubling can be used to obtain the position of maximum correlation over all of the PEs.

6. CONCLUSIONS

Tables 1 through 4 contrast quantitatively the complete and partial sums approaches to the operations involved in image correlation. In order to more readily compare the two approaches, let $R' = C' = I'$ and $r = c = M'$. The results are shown in Table 5.

As can be seen from the table, for each of the individual Σxy, Σy, and Σy^2 algorithms, the complete sums approach requires more space and/or arithmetic operations than the partial sums approach. However, when these algorithms are interleaved to compute and locate the maximum RXY value, the complete sums method requires more arithmetic operations, but fewer inter-PE transfers and less space. Which method is faster will therefore depend on the relative time to perform arithmetic operations versus transfers. For example, if the time to perform a transfer equals the time to perform a multiplication, then the complete sums method will be faster. If inter-PE transfers can be overlapped with arithmetic operations, then the partial sums method will be faster. Thus, in order to determine which approach will be faster on a particular system, the exact timings for these operations must be considered.

The difference in the space required for the two approaches is not large. However, if the PE memories are small, or, for the RXY computation, if C' is large, the space difference may be a factor in selecting an algorithm.

Some basic differences resulting from the two algorithm strategies are evidenced in the RXY complexities. In the complete sums approach, two PEs hold and operate on some of the same image elements. As a result, redundant arithmetic operations are performed in the Σy and Σy^2 computations, i.e., the sum (or product) of the same two elements is sometimes performed in two PEs. These redundant operations are not performed in the partial sums algorithms. On the other hand, the partial sums method requires more transfers. Each non-local y value needed by a PE is transferred in only once in the complete sums approach and three times (as part of partial Σy, Σy^2, and Σxy terms) in the partial sums approach.

	Ac-Ap	Mc-Mp	Tp-Tc	Space Difference
Σxy	0	0	0	$Sc-Sp=$ $2(M')^2-4M'+3$
Σy	$8I'+3M'-3$	-	0	$Sc-Sp=$ $(M')^2-1$
Σy^2	$8I'+3M'-3$	$2I'M'-2I'$ $+(M')^2-2M'+1$	0	$Sc-Sp=$ $(M')^2-1$
RXY	$16I'+6M'-6$	$2I'M'-2I'$ $+(M')^2-2M'+1$	$2(2I'M'-2I'$ $+(M')^2-2M'+1)$	$Sp-Sc=2I'M'$ $-2I'-2M'+2$

Table 5: Comparison of the complete sums and partial sums approaches, using the following notation:
$R' = C' = I'$
$r = c = M'$
Ap = adds for partial sums approach
Ac = adds for complete sums approach
Tp = inter-PE transfers for partial sums approach
Tc = inter-PE transfers for complete sums approach
Mp = multiplies for partial sums approach
Mc = multiplies for complete sums approach
Sp = space for partial sums approach
Sc = space for complete sums approach

The SIMD machine model used assumed a multistage network which can perform each required data transfer in a single step. Consider instead an SIMD machine where the PEs are connected in a nearest neighbor pattern, i.e., PE i is connected to PE i+1, i-1, $i+N^{1/2}$, and $i-N^{1/2}$ (arithmetic mod N). Examples of such machines are the Illiac IV [4], DAP [7], CLIP4 [5], and MPP [3]. In analyzing the two algorithm approaches, the number of transfer steps must be increased. Assuming $NC = NR = N^{1/2}$, the nearest neighbor connection scheme requires 1 transfer step to do each of the PE i to PE i+1, i-1, i+NC, and i-NC transfers, and 2 transfer steps to do each of the PE i to PE i+NC+1, i+NC-1, i-NC+1, and i-NC-1 transfers. Furthermore, if the match area extends over more than two PEs, additional multiple data transfer steps will be needed. (Even though two transfers are required for some steps, typically each transfer in a nearest neighbor network will be faster than a transfer through a multistage network.) The results in Tables 1 to 5 can therefore be applied to nearest neighbor connected systems by modifying the transfer step counts as described. (The number of transfers for recursive doubling will also be increased.)

In the SIMD machine model in section 1, it was assumed that each processor was associated with a local memory to form a PE. Consider a different organization where the processors are separate from the memories, and the interconnection network is used to connect the processors to the memories. Inter-processor communications can be accomplished by writing into and reading from the shared memory. STARAN is an SIMD machine organized in this way [1,2]. Since all memory accesses go through the interconnection network, there are no explicit inter-processor data transfers (assuming a network such as one of those mentioned in section 1 were used). Thus, with such an organization, the partial sums approach is faster than the complete sums approach. (In the STARAN machine, the interconnection network is not flexible enough to allow the processors to access the appropriate memories in all cases (e.g., processor i to memory i+NC+1). In these cases, an additional pass through the network will be required to align the data.)

The SIMD algorithms presented demonstrate how SIMD parallelism can be used to reduce the execution time of computationally intensive image processing tasks. For the image correlation algorithms, the asymptotic complexity for arithmetic operations is reduced from O(RCrc) for the serial algorithm to O(RCrc/N) for the N-PE parallel algorithms. The overhead of inter-PE communications incurred has asymptotic complexity O(C'r+R'c+rc).

In summary, SIMD algorithms to perform the window-based operations needed for image correlation have been explored. Two fundamental algorithm strategies were presented, and their time-space-transfer complexities were compared. Through studies and analyses such as this, more can be learned about both the art of parallel programming and the ways in which parallelism can be exploited in image processing.

Acknowledgment: The authors thank George B. Adams III for his comments.

REFERENCES

[1] K. Batcher, "The flip network in STARAN," 1976 Int. Conf. Parallel Processing, Aug. 1976, pp. 65-71.

[2] K. Batcher, "The multidimensional access memory in STARAN," IEEE Trans. Comp., Vol. C-26, Feb. 1977, pp. 174-177.

[3] K. Batcher, "MPP--a massively parallel processor," 1979 Int. Conf. Parallel Processing, Aug. 1979, p. 249.

[4] W. Bouknight, et al., "The Illiac IV system," Proc. IEEE, Vol. 60, Apr. 1972, pp. 369-388.

[5] M. J. B. Duff, "CLIP4: a large scale integrated circuit array parallel processor," 3rd Int. Joint Conf. Pattern Recognition, 1976, pp. 729-732.

[6] T. Feng, "Data manipulating functions in parallel processors and their implementations," IEEE Trans. Comp., Vol. C-23, Mar. 1974, pp. 309-318.

[7] P. M. Flanders, "Efficient high speed computing with the distributed array processor," Symp. on High Speed Computer and Algorithm Organization, Apr. 1977, pp. 113-128.

[8] M. Flynn, "Very high-speed computing systems," Proc. IEEE, Vol. 54, Dec. 1966, pp. 1901-1909.

[9] A. J. Krygiel, "An implementation of the Hadamard transform on the STARAN associative array processor," 1976 Int. Conf. Parallel Processing Aug. 1976, p. 34.

[10] D. Lawrie, "Access and alignment of data in an array processor," IEEE Trans. Comp., Vol. C-24, Dec. 1975, pp. 1145-1155.

[11] M. Pease, "The indirect binary n-cube microprocessor array," IEEE Trans. Comp., Vol. C-26, May 1977, pp. 458-473.

[12] W. K. Pratt, "Correlation techniques of image registration," IEEE Trans. Aerospace Electronic Systems, Vol. AES-10, May 1974, pp. 353-358.

[13] A. Rosenfeld, M. Thurston, "Edge and curve detection for visual scene analysis," IEEE Trans. Comp., Vol. C-20, May 1971, pp. 562-569.

[14] S. Ruben, et al., "Application of a parallel processing computer in LACIE," 1976 Int. Conf. Parallel Processing, Aug. 1976, pp. 24-32.

[15] 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.

[16] H. J. Siegel, et al., "PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition," IEEE Trans. Comp., to appear.

[17] H. J. Siegel, S. D. Smith, "Study of multistage SIMD interconnection networks," 5th Symp. Comp. Arch., Apr. 1978, pp. 223-229.

[18] L. J. Siegel, H. J. Siegel, R. Safranek, M. Yoder, "SIMD algorithms to perform linear predictive coding for speech processing applications," 1980 Int. Conf. Parallel Processing, Aug. 1980, pp. 193-196.

[19] H. Stone, "Parallel computers," in Introduction to Computer Architecture, H. Stone, ed., S.R.A., Chicago, IL, 1975.

[20] K. J. Thurber, Large Scale Computer Architecture: Parallel and Associative Processors, Hayden Book Co., Rochelle Park, N.J., 1976.

PARALLEL COMPUTER ARCHITECTURES FOR IMAGE PROCESSING

Anthony P. Reeves
School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

Abstract -- Image processing problems frequently involve large structured arrays of data and a need for very rapid computation. Special parallel processing schemes have evolved over the last 20 years to deal with these problems. In this paper the nature of image processing tasks are outlined and the parallel computer architectures which have been developed for these tasks are reviewed. Most of these special architectures may be loosely classified as either SIMD or pipeline structures although some MIMD structures have been designed for high level image analysis.

In recent years several Multiple SIMD (MSIMD) schemes have been proposed as suitable architectures for image processing. The fundamental problems of developing an effective MSIMD system are discussed and a simple SIMD/MIMD computational model for comparison with such systems is proposed.

Introduction

Image processing frequently involves very large regular data structures and a need for very high speed computation. Through the brief history of digital image processing, special parallel processing architectures have been proposed and implemented, see [1] and [44].

In this paper we will consider architectures which deal with images in the most conventional format, namely a large two dimensional matrix of brightness values called pixels. This format is applicable to many applications. For example, in the industrial environment computer robot vision and part inspection may involve real-time video data as small as 256 x 256 pixels with only 6-bits of information for each pixel. Military applications involving FLIR and optical video imagery may involve similar sized data. At the large end of the scale, remotely sensed imagery from the LANDSAT satellite may involve images of 4000 x 4000 of 8 bit pixels and several spectral bands of information. Aerial photographs may also be digitized to 4000 x 4000 or larger. In the biomedical area there are many instances of imagery, mainly from microscope sides, at all levels of size, color, and resolution.

The nature of image processing problems may be divided into two characteristic classes: low level image processing and image analysis. In low level image processing, the output usually has a similar matrix size to the input. The processing may involve algorithms for restoration, noise removal, geometric correction or simple feature extraction such as edge detection, or feature enhancement. Such tasks are well suited to an SIMD computer structure and most special image processing hardware systems are of this form.

The image analysis involves classifying segments or features of the image into known classes. This may involve combining a set of segments or features to create a total composite object. The techniques involved here are usually termed pattern recognition or artificial intelligence. For these cases, the image is no longer considered as a large matrix of pixels. Segments of the image are represented by more convenient data structures such as a set of parametric measures or a labeled relation graphs. Classification algorithms frequently involve a set of sequential searches for pattern matching which may be conducted independently in parallel. Such tasks, which involve many independent operations on a common data base, are well suited to a MIMD parallel structure.

There has been much work at both extremes of image processing; many low level image processing algorithms have been developed as have many pattern recognition techniques. However, the interface between these two areas is less well understood. This grey area involves deciding which features are to be extracted by the low level image processing and in what format they should be presented to the image analysis section. A related problem is how the image analysis algorithm may interrogate the original low level data to obtain further information when necessary.

The areas of image processing are outlined in Fig. 1. SIMD processors are well suited to most low level algorithms and can also perform many simple feature extraction operations. In some cases they may be effectively utilized for some statistical classification techniques. MIMD processors are not very efficiently organized for low level image processing since much hardware is devoted to individual control units and reliable asynchronous data communication between processor units. There is also a problem with sharing near neighbor data between the processor units. Some feature extraction algorithms, especially those serial in nature, such as contour following, are well suited to the MIMD structure. Classification schemes especially those involving syntactic methods are very well suited to the MIMD structure

An initial problem in designing a complete parallel image processing system is to decide which basic machine architecture to use: SIMD, or MIMD.

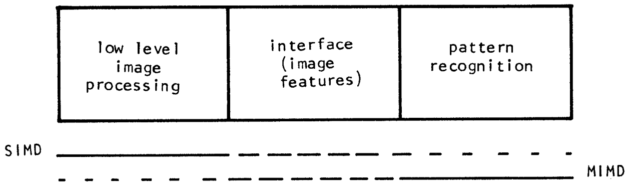


Fig. 1 The main stages of Image Processing.

0190-3918/81/0000/0199$00.75 © 1981 IEEE

Some researchers have proposed a combined MSIMD system as a possible solution. These schemes enable a group of independent SIMD processors to be assigned to a task. One problem in designing such systems is to ensure that the worst features of both systems, i.e., the expense and inefficiency of the MIMD data communication and control, and the inflexibility of the homogeneous SIMD structure are not both present in the combination. In this paper a simpler computational model is proposed which may be used as a benchmark for the efficiency of more complex designs.

Historically, a fundamental bottleneck in processing capability has been perceived with the low level image processing task. Many SIMD and pipeline architectures have been developed for low level image processing applications, whereas special architectures for high level image analysis have received less attention. Most of the low level architectures have the characteristic that a single operation is automatically applied to all elements of the image matrix in a fixed amount of time. These architectures may be divided into three types: (a) Parallel Binary Array Processor (b) pipeline processor and (c) special function units.

In the following sections of this paper the architectures which have been developed for low level image processing are reviewed with emphasis on more recent designs. Then the features of proposed MSIMD schemes for image processing applications will be discussed.

Parallel Binary Array Processors

Binary Array Processors [11] operate in the single instruction stream-multiple data stream (SIMD) mode with a matrix of identical processing elements (PE's). The whole image or a consecutive block of the image is distributed through the PE's and processed in parallel. All data paths within a PE are only one bit wide and each PE is connected to PE's adjacent to it.

The main features of the BAP scheme result from the bit-serial architecture of the PE's and near-neighbor interconnection scheme. The bit-serial architecture allows flexible data formats and makes the BAP very efficient with respect to memory and processing resource utilization. Many image processing algorithms require that data within local areas of each pixel is to be combined; the near neighbor interconnection scheme enables these algorithms to be efficiently implemented.

In 1959 Unger [6,7] proposed a parallel BAP machine with a matrix of PE's for image processing applications; many of Unger's ideas have been incorporated into later BAP designs. One of the first hardware BAP systems was the SOLOMON computer [8] which was built by Slotnick at Westinghouse. Another landmark parallel BAP was Illiac III [9] on which development started in 1963 by McCormick but which was never completed. Early parallel BAP development was hampered by the very high cost of the parallel hardware. Both SOLOMON and Illiac III were designed to have a 32 x 32 PE matrix size.

With the advent of large scale integration the construction of much larger BAP's has become feasible. Since the Illiac III design a sequence of BAPs called CLIP have been developed by Duff [10]. The most recent version, CLIP4, is an operational, LSI, 96 x 96 parallel BAP.

The features of current BAP designs will be discussed by describing two diverse architectures. These are: the BASE system which is being developed in a small prototype form at Purdue University [11,12] and the MPP [16].

The BASE PE is shown in Fig. 2. It consists of three main components: a Boolean processor which can implement any three input Boolean function, a 1-bit wide local memory and a Near Neighbor function processor (NNF). In general, the operands A, B and C would be held in 1-bit registers which could receive data from the local memory on a common bus line.

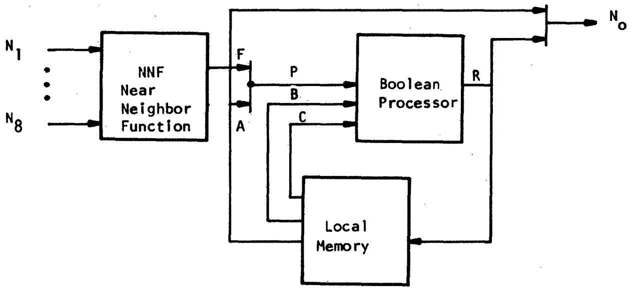


Fig. 2 The BASE PE Organization

The NNF receives data from the 8 near neighbor PE's as shown in Fig. 3 and realizes the following function

$$F = \bigvee_{i=1}^{8} (g_i \wedge N_i)$$

Where $\{g_1 \ldots g_8\}$ is an 8-bit control vector. A hexagonal near neighborhood may also be selected.

There are three fundamental instruction types for BAP's [11]: Boolean, simple near neighbor and recursive near neighbor. Boolean instructions are used for logical and arithmetic operations within the local memory; the NNF is not used. The BASE

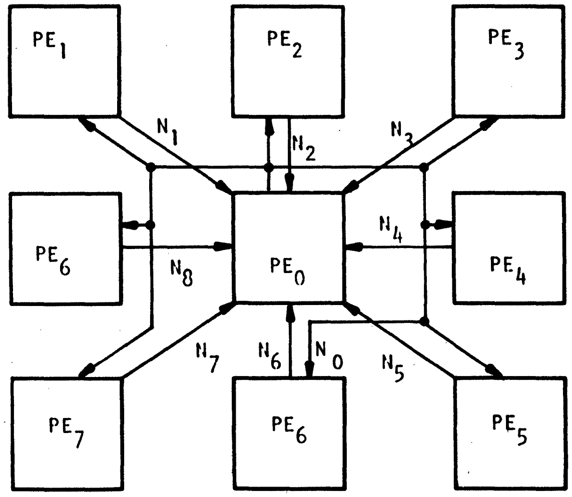


Fig. 3 Data interconnections between a BASE PE and its 8 adjacent PE's

Boolean processor can implement any of the 256 three input Boolean functions as specified by an 8-bit control vector. A 2-input Boolean function could be adequate for Boolean instructions; however, the 3-input function is much more efficient for arithmetic operations and is also necessary for recursive near neighbor instructions. Arithmetic may be achieved with conventional bit-serial algorithms or, in some cases, functions may be more efficiently implemented with a specially optimized instruction sequence [13].

Simple near neighbor instructions specify that one operand comes from a near neighbor PE or a selected subset of near neighbors. An ORed subset of near neighbors is useful in some binary image algorithms. For example, the perimeters of all objects in a binary image may be obtained with one of these instructions.

Recursive instructions are implemented on only a few BAP's. In this instruction the near neighbor output is taken from R rather than A in Fig. 1. A signal may propagate through a connected sequence of PE's in a single recursive instruction. This asynchronous operation may be several times faster than an equivalent sequence of simple near neighbor instructions depending upon the technology and detailed design of the processor. Recursive near neighbor instructions are useful for horizontal arithmetic, which treats the rows of the PE matrix as a set of data items, and some two dimensional binary topology operations.

A prototype BASE system, called BASE-8, is currently being developed at Purdue University [12]. It is constructed with Schottky TTL and involves 64 PE's arranged in an 8 x 8 matrix. The PE matrix can automatically scan a larger matrix size processing it in consecutive 8 x 8 blocks.

A general block diagram for a large scale BAP system is shown in Fig. 4. The data processing is achieved by the array of PE's which simultaneously process a submatrix of an image. The total image is processed as a sequence of these submatrices. Data is input to and output from the PE array via the I/O buffer memory which communicates the data to image peripherals and conventional computer bulk storage devices. With the current LSI systems each bit plane is input along one edge of the PE array one column on each clock cycle. Each row of the PE array acts as a shift register. When the complete bit plane has been input it is stored in the local memory in one clock cycle. When input data is in the form of a stream of pixels it must be converted to the bit-plane format. Instructions to the PE array are issued by a single high speed microprogrammed controller. The whole system synchronization is maintained by a conventional host computer which issues macro instructions to the controller. Some feature information may be extracted from the PE array by the global information extraction mechanism.

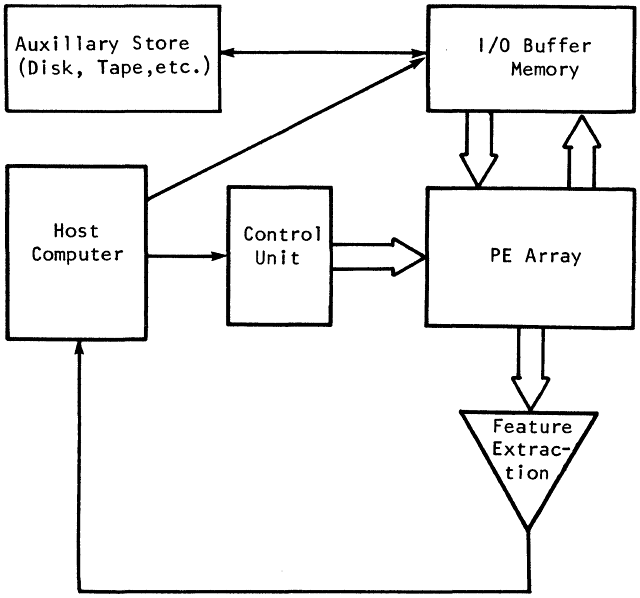


Fig. 4 Binary Array Processor System

The most usual mechanism is an OR function over all PE's which indicates if any PE has a one element. This is useful for terminating loops and for detecting the presence of objects. For applications which require more feature extraction, such as shape analysis, a more powerful scheme is desirable. One scheme is to count the number of bits set in the bit plane which is implemented on BASE-8 and could be efficiently implemented on LSI BAP systems [14].

The MPP was designed by NASA [15] with the primary function of analyzing LANDSAT satellite data. The system has been redesigned by Goodyear Aerospace [16] who are contracted to build a hardware MPP by mid 1982. The PE array consists of 16384 PE's organized in a 128 x 128 matrix. A special LSI CMOS/SOS chip has been designed which contains 8 PE's without their local memory in a 2 x 4 submatrix. In the initial version each PE will have 1024 bits of local memory; however, the design includes provisions for up to 16k bits per PE for when such memory chips become available. The clock cycle time for the array is 100 ns.

A simplified MPP PE is shown in Fig. 5. The emphasis with this design is fast arithmetic computation rather than binary near neighbor operations. The NN select unit enables a bit plane to be shifted in one of the four cardinal directions in one instruction. The Boolean processor implements all 16 possible Boolean functions between the P register and the value on the data bus; in this case P is an accumulator. For arithmetic operations a dynamically reconfigurable, variable length shift register with a maximum length of 30-bits and a full adder are available. This organization is faster than the BASE scheme especially where multi-

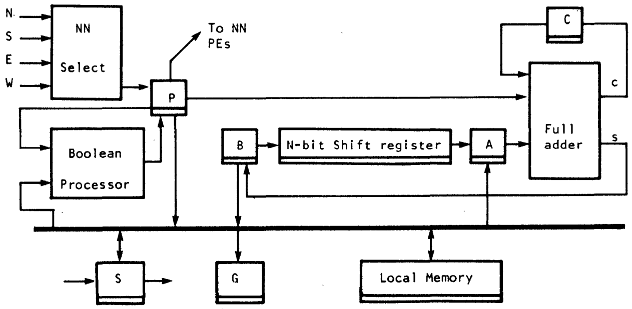


Fig. 5 Simplified MPP PE Organization

plication is involved; in this case the shift register is used for circulating the partial product.

The MPP is very fast for 8-bit pixel operations; it can execute 6.5 billion additions per second or 1.8 billion multiplications per second. For 32-bit floating point data the MPP can execute 430 million additions per second or 210 million multiplications per second. The integer addition is optimal for the given processor memory bandwidth, the multiplication could be made faster. It is anticipated that with VLSI technology faster multiplication and other processing functions will be included in PE designs and PE arrays will become larger [17,18]. A special programming language called Parallel Pascal has been developed for the MPP and other BAP's [19]. A compiler for this language is currently being developed and a translator is available which allows Parallel Pascal programs to be run on conventional Pascal systems.

Other parallel bit serial processors have also been used for image processing. The STARAN associative processor [41] has been used for LANDSAT image analysis [42]. It can perform similar operations to a BAP except that it can only process a row or a column of the image at a time in parallel. The distributed array processor (DAP) [40] is a BAP implemented with a 64 x64 PE matrix which was not designed specifically for image processing. A LSI chip has been developed for it which contains 4 PE's without local memory and is based on the uncommitted logic array (ULA) approach. Low level image processing tasks have been implemented on the DAP [39].

Pipeline Processors

The basic organization of a pipeline processor for image processing is shown in Fig. 6. Image data is passed in raster scan format from the buffer memory into a pipeline of processing stages. The function of each stage is specified by the host computer through the instruction bus. Once set up, a processing stage performs the same operation on every element of the data sequentially. There is an initial set-up delay while data is input to the pipeline before the first result appears at the last stage then a result is generated with each clock cycle. Therefore completely processing an n x n image requires the set-up time plus n^2 clock cycles.

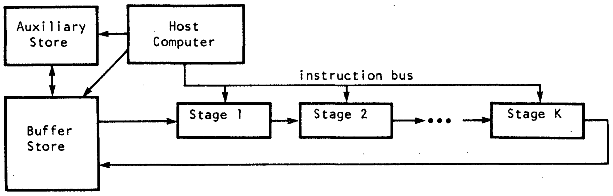


Fig. 6 Computer Organization for Pipelined Image Processing

One main advantage of this architecture is that no input data reformatting is necessary. In fact the input to the first stage could be taken directly from the output of a TV camera. Other advantages are the very simple data interconnections between processing stages and, since an instruction resides in a stage for many clock cycles, a high speed control unit is not required.

A pipeline stage can implement near neighbor processing functions by means of two shift registers as shown in the near neighbor processing unit (NNPU) in Fig. 7. Raster formatted data is input at register N_1 and is then shifted through the shift registers and near neighbor registers until it reaches N_5. To process an image with a row of n elements the shift registers are set to length n-3. Then an input data element will be at register P after n+1 clock cycles and its near neighbor values will be available in registers $N_1 \cdots N_8$. A small amount of additional logic is required to maintain the correct values of near neighbors at the edge of the image. A simple near neighbor function may be implemented by a near neighbor function unit connected to the nine registers as shown in Fig. 7. The set-up time for this NNPU is n+2 clock cycles.

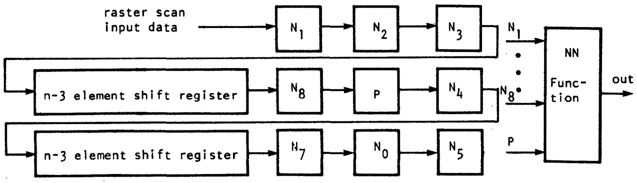


Fig. 7 Organization for a pipelined near neighbor processing unit (NNPU)

One of the most ambitious pipeline image processors which embodies these concepts is the cytocomputer developed by Sternberg at the Environmental Research Institute of Michigan [20]. Currently a prototype cytocomputer is operational which involves two pipelines: one contains 80 binary stages and the other contains 25 grey level stages. The data connections between the stages are 8 bits wide. The binary stage can compute a binary function on the near neighbors of one of the inputs and combine the result of this with the other 7 bits. All 2^9 near neighbor functions are implemented by means of a table-look-up memory which generates 8 outputs. A select unit selects 8 bits from the 8-bit input data and the 8-bit near neighbor function and these are mapped by means of a 256-word 8-bit table-look-up memory to the 8 outputs for the stage. For the grey level stages the architecture is similar to the NNPU; in this case all shift registers are 8-bits wide and the near neighbor function involves 8-bit arithmetic operations. The 8-outputs are also mapped through a 256-word table-look-up memory.

An LSI version of the cytocomputer is currently being developed. It will consist of up to 550 stages; each stage is capable of both binary and gray level functions. A special LSI CMOS/SOS chip is being developed which will contain one cytocomputer stage.

Eskenazi and Wilf have developed a simple pipelined processor system for real-time image analysis at the Jet Propulsion Laboratories [22]. The system is designed to produce partial object boundaries from raw image data and involves three different processing stages, each of which operates on a 3 x 3 local neighborhood.

The cytocomputer structure is an effective mechanism for very low level image processing (e.g., image filtering operations). Lougheed and McCubbrey have made a comparison between the cytocomputer and other parallel structures for some very low level image processing tasks [21]. However, it is not clear how such a structure can easily combine more than one image in an operation or perform functions such as geometric correction.

These limitations are due to the limited 8-bit linear interconnection scheme of the pipeline. It would be possible to have a more complex pipeline involving stages that could deal with more than one 8-bit input. However, the cost of implementing a more flexible interconnection scheme would be very high.

The FLIP system [23] consists of 16 special processors with two input ports and one output port (they do not contain a 3 x 3 local window function like the cytocomputer). These may be reconfigured into any interconnection arrangement by software. Interconnections are achieved by 16 interprocessor buses and 16 buses from the I/O control unit and buffer store. Each processor is capable of executing a simple sequential algorithm on each input pixel; it contains 256 words of program store and 50 bytes of data store. The processor intercommunication is achieved by asynchronous handshaking, so that the whole system will function correctly when the processors require different processing times for each pixel. The multiple input ports from the buffer memory enables a pixel and selected near neighbor values to be input to the processors simultaneously.

Special Function Units

The organization of the third architecture type, special function units, is illustrated in Fig. 8.

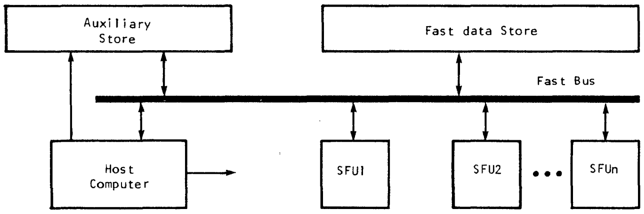


Fig. 8 Special Function Processor System

Each special function unit (SFU) is a direct hardware implementation of an image processing algorithm, or in some cases it may implement a set of related functions. SFU's may contain some local memory and program control. A system usually consists of a set of SFU's and a fast image memory connected together by a high speed bus. The image data is processed serially by a SFU, which may be pipelined, and image results are returned to the fast memory.

Early near neighbor processors were based on this scheme [2-5]. For some current medium speed SFU systems see also [26,37,43,44]. Current systems usually involve bit-parallel data and may involve SFU's which process several pixels simultaneously.

An important image processing operation is convolution with a small predefined matrix. Many image processing functions such as convolution and the DFT may be defined by an inner product operation. The inner product computer (IPC) [24-25] is a special function unit for rapidly computing the inner product operation. For two vectors A and B the inner product is defined by

$$R = \sum_{i=1}^{n} A_i B_i$$

A possible design for an IPC is shown in Fig. 9; it

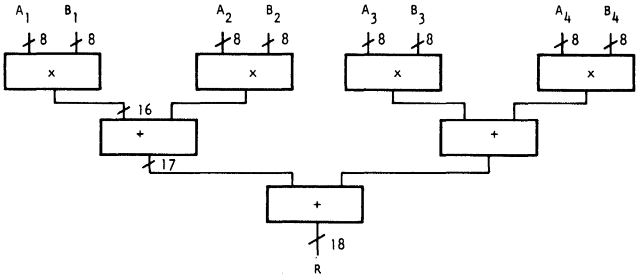


Fig. 9 An inner product computer design

involves n multipliers to compute the vector element products and a tree of n-1 adders to sum the results. In practice the design may be pipelined to achieve very high speed; moreover the logic is simple to design and lay out. The bandwidth of the system may be exactly matched for integer operands. In the example, 2 4-element 8-bit vectors result in an inner product of 18 bits.

A possible organization for an IPC system is shown in Fig. 10. Weights are stored in the

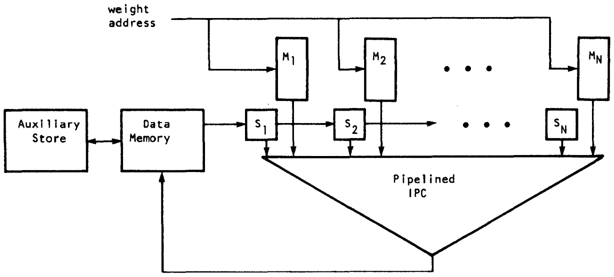


Fig. 10 An IPC computer organization for Image Processing

memories $M_1 \cdots M_n$ and data is shifted from the data store into the shift register $S_1 \cdots S_n$

To convolve on n-element kernel function with an image the n-kernel values are first loaded into the n-memories then the image data is shifted into the shift register S. After the set-up time for the IPC a new data item will be input and a result will be output with each clock cycle. If a two dimensional convolution is required then delay shift registers could be added to S as shown in Fig. 7. The set-up time would be increased but the following execution time would still be equal to the number of results.

General transform operations such as the Discrete Fourier Transform (DFT) may be achieved by loading the M memories with all the transform coefficients and the S registers with the data to be transformed. The addresses of the memories are stepped through sequentially and a result is generated with each clock pulse after the set-up time of the IPC. A complex DFT can be achieved on an IPC for real numbers with four inner product operations [24].

The parallel pattern processor (PPP) [26] involves an 8 input IPC with buffer memory for seven lines which can implement an 8 x 8 convolution function. The Dynamic Spatial Reconstructor [24] for very high speed tomographic reconstructions involves a very fast IPC based on sub-nanosecond ECL technology.

In some cases the SPU may contain several SIMD PE's which operate on different pixels [1,5,43] For example, the FIP processor in the PICAP II system [43] contains 4 programmable PE's which can execute a small convolution or a programmed sequence of near neighbor operations on 4 pixels simultaneously. Several rows of the image are rapidly accessible in a local buffer memory.

Initial research has been done on implementing image processing functions with charge coupled devices (CCD's). This technology processes unique features such as: low power delay product (0.1 pico Joules), very high packing density, and simple analog signal processing implementation. Moreover, since image sensors can be made with CCD's the possibility of including some low level processing on the same substrate as the sensor exists (smart sensors).

Several test chips have been developed for computing local functions such as edge detection, convolution filtering, median filtering and adaptive thresholding [30-32]. These test chips require data to be input in analog form from external CCD shift registers. Real-time processing of a video signal (7.5 MHz) is possible; however only about 6-bits of precision can be maintained with the analog processing.

For high speed parallel processing a planar (focal plane) processing concept has been proposed. In this scheme, data from a CCD matrix is shifted into a set of processors connected to one edge of the matrix. The outputs of the processors are connected to one edge of a result matrix into which the results are shifted. These proposals include a single substrate design for simple processing functions [31] and a multi-chip system for more complex processing [32].

CCDs have also been proposed for on-board Satellite classification of LANDSAT remotely sensed data [33]. This proposal is based on the concept of a CCD IPC device.

Characteristics of Low Level Architectures

In summary, some of the features of the current parallel architectures for image processing are as follows. For very high performance the BAP approach may be used; the parallelism may be increased until there are as many PE's as pixels without any fundamental problems. Once there are as many PE's as pixels, the PE's with bit-parallel rather than bit-serial operations may be considered for even higher performance. For very low level image processing the pipeline approach may offer a simpler solution than the BAP system. Both BAP and pipeline systems are very suitable for VLSI implementation since they involve a large number of identical modules. Furthermore, all chip interconnections are very short. Therefore, no scaling problems should be experienced when the number of processing units is increased. For medium speed applications the special function approach is possible. Higher speed may be achieved by using a wider high speed bus and redesigning the functional units to have several SIMD PE's. Some current systems would be difficult to extend in this way. Scaling problems may be experienced when the size of a system is expanded since the length of the bus may need to be extended and data interconnections may be more complex. In certain applications either very high speed logic or CCD technology may be appropriate.

Most current pipeline and special function units are based on 8 bit-wide data paths. This precision is adequate for representing most image data, but partial results may easily require more information. The BAP approach is not limited to a fixed pixel size and can easily deal with floating point data.

MIMD and MSIMD schemes

There has been much interest in recent years in developing multiple instruction stream--multiple data stream (MIMD) parallel processors. Such a computer may be constructed with a set of independent processors executing different programs, which can communicate with each other and share some memory resources. An organization for this type of computer is shown in Fig. 11. The processors usually have some local memory but also have access to a common shared memory. The interconnection network which connects the processors to the shared memory presents a significant design problem especially when a large number of processors are involved. A crossbar switch scheme would allow any processor to access any shared memory module. However, it is usually too costly to implement. A limited permutation network is usually more practical. However, significant delays in accessing data from the shared memory may occur due to the longer path length through the network and blocking. The availability of cheap microprocessors has made the construction of these computers feasible. However, fundamental research problems still exist as to the best methods to interconnect the processors and dynamically distribute the processing tasks between them.

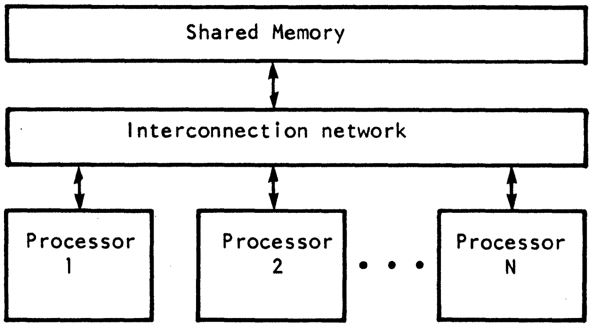


Fig. 11 A MIMD Processing Organization

The main advantage of the MIMD scheme over the

SIMD scheme occurs with high level image understanding tasks. For example, it would be possible to assign a set of processors to analyze a set of segmented objects where each processor deals with a single object. Alternatively several processors could be set to analyze a single object where each processor may realize a different analysis algorithm. Low level near neighbor algorithms are usually more easily implemented with the SIMD scheme.

There have been proposals for MIMD and MSIMD architectures which are intended for image processing applications [27,28,29,35,36]. Two microprocessor based MIMD designs are currently being researched at Purdue University. One scheme, the Partitionable SIMD/MIMD system (PASM) [27,28] involves 1024 microprocessors and 16 control units. The processors are interconnected with a limited, synchronous interconnection network. PASM may be dynamically reconfigured into a set of different size, independent SIMD systems to suit the requirements of a processing task. Up to 16 independent SIMD systems may be possible at one time, each controlled by one control unit. The number of processors allocated to a control unit may be dynamically varied; at one extreme a single control unit may control all 1024 processors in which case it behaves like a single SIMD system. There is no general shared memory in the PASM system, rather data is shared through interprocessor data connectors.

In the second scheme, the Purdue multi-mode multimicroprocessor [29] about 128 microprocessor are considered. Each microprocessor has its own control unit and local memory so it is a true MIMD system. In early designs an interprocessor mechanism for synchronizing a set of processors in an SIMD mode was proposed however this is not considered important in the current design. Block data transfers are made between the local processor memories and the shared memory; most processor data accesses are made to their local memory. With this scheme a simple processor interconnection network can be used and the delay in establishing a path from a processor to a shared memory module is offset by transferring more than one data word. In this sense, the processor local memories may be considered to be more like cache memories.

In both the above systems high speed low level image processing is achieved by distributing the task amongst as many available processing units as possible. The hardware utilization for these tasks is not as good as the special architectures previously discussed since only one control unit is necessary and multiple control units are a feature of an MIMD system.

The following simple SIMD/MIMD model is proposed which may offer better hardware utilization. This simple model is outlined in Fig. 12. A single SIMD processor is used for low level image processing and a separate MIMD processor is used for image analysis. The size of each processor would be initially configured to be adequate for the anticipated work load. In large applications, or when high reliability is necessary a loosely coupled network of several SIMD and MIMD processors could be used.

In the general scheme shown in Fig. 12, no specific architecture details are specified such as the nature of the interconnection network or method of I/O interfacing. The host of the MIMD system may be simply an I/O channel if there is a distributed operating system. The main advantage of the general structure is that the best features of both the SIMD and MIMD systems are easily attained.

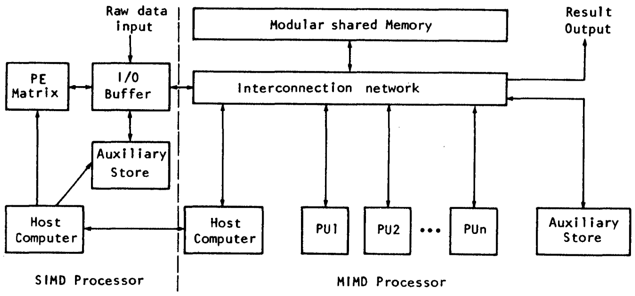


Fig. 12 SIMD/MIMD computational model

The data flow for low level image processing and image analysis are frequently quite different. In the proposed system the SIMD processor would be optimized for low level image processing and the MIMD system would be optimized for image analysis. The organization of the MIMD system would be much simpler than the MSIMD system since it would not have to deal with the SIMD data flow. SFU's may be added to the MIMD processors to assist with well defined image analysis tasks.

There are two possible cases when the MSIMD system might have an advantage over the proposed system: (a) if the ratio of low level image processing to image analysis was very unpredictable and (b) if the SIMD low level image processing required the processing of many, very small, sub images. The ratio of processing types in case (a) would have to have a very great variability before the extra expense of the MSIMD system would be warranted. For case (b), large SIMD systems may be partitioned by means of masks so that an identical algorithm may be executed simultaneously on a set of sub images. Therefore, a reduction in efficiency of the single SIMD system would only occur if the algorithm for the sub images were very different from each other. There would be very little reduction in efficiency if the low level image processing was realized with a pipeline structure.

Conclusions

Image Processing tasks may be divided into two areas: low level image processing and image analysis. Historically,the low level image processing problem has received the most attention and several viable parallel computer architectures have been developed for this task. Some of these architectures have been reviewed and their characteristics are summarized in section V. Current parallel architectures are very efficient for implementing current low level image processing algorithms and they are suitable for VLSI implementation.

several MSIMD architectures have been proposed in the literature for image processing. A SIMD/MIMD system has been proposed in this paper which is simpler to organize than the MSIMD scheme and will be more efficient for well defined image processing tasks.

References

1. K. Preston, Jr., M. J. B. Duff, S. Levialdi, P. E. Norgren and J-I, Toriwaki, "Basics of Cellular Logic with Some Applications in Medical Image Processing," Proceedings of the IEEE, Vol. 67, No. 5, May (1979) 826-856.
2. S. B. Gray, "Local Properties of Binary Images in Two Dimensions," IEEE Trans. on Computers, May (1971) 551-569.
3. M. J. E. Golay, "Hexagonal Parallel Pattern Transforms," IEEE Trans. on Computers, August (1969) 733-740.
4. K. Preston, "Feature Extraction by Golay Hexagonal Transforms," IEEE Trans. on Computers, September (1971) 1007-1014.
5. B. Kruse, "A Parallel Processing Machine," IEEE Trans. on Computers, December (1973) 1057-1087.
6. S. H. Unger, "A Computer Oriented Toward Spatial Problems," Proc. of the IRE, October (1958) 1744-1750.
7. S. H. Unger, "Pattern Detection and Recognition," Proc. of the IRE, (1959) 1737-1752.
8. D. L. Slotnick, W. C. Borck and R. C. McReynolds, "The SOLOMON Computer," Proceedings of the Fall Joint Computer Conference, (1962) 97-107.
9. B. H. McCormick, "The Illinois Pattern Recognition Computer ILLIAC III," IEEE Trans. on Computers, December (1963) 791-813.
10. M. J. B.Duff, "CLIP4: A Large Scale Integrated Circuit Array Parallel Processor," 3rd International Joint Conference on Pattern Recognition, (1976) 728-732.
11. A. P. Reeves, "A Systematically Designed Binary Array Processor," IEEE Trans. on Computers, April (1980) 278-287.
12. A. P. Reeves and R. Rindfuss, "The BASE 8 Binary Array Processor," Proceedings of the IEEE Conference on Pattern Recognition and Image processing, (1979) 250-255.
13. A. P. Reeves and J. D. Bruner, "Efficient Function Implementation for Bit-Serial Parallel Processors," IEEE Trans. on Computers Vol. C29, No. 9 (1980) 841-844.
14. A. P. Reeves, "On Efficient Global Feature Extraction Methods for Parallel Processors" Computer Graphics and Image Processing, Vol. 14 (1980) 159-169.
15. L. W. Fung, "A Massively Parallel Processing Computer," High-Speed Computer and Algorithm Organization, D. J. Kuck et. al. Ed. New York Academic, (1977) 203-204.
16. K. E. Batcher, "Design of a Massively Parallel Processor," IEEE Trans. on Computers Vol. C-29, No. 9 (1980) 836-840.
17. A. P. Reeves, "A VLSI Binary Array Processor Design," submitted to IEEE Trans. on Computers also in Purdue technical report TR-EE 80-32.
18. A. P. Reeves, "The Anatomy of VLSI Binary Array Processors," Workshop on New Computer Architectures and Image Processing, Ischia, Italy, 1980.
19. A. P. Reeves, J. D. Bruner and M. S. Poret, "The Programming Language Parallel Pascal," Proceedings of the 1980 International Conference on Parallel Processing, August 26-29 (1980) 5-6.
20. S. R. Sternberg, "Parallel Architectures for Image Processing," Proceedings of the 3rd International IEEE COMPSAC, Chicago, (1979) 712-717.
21. R. M. Lougheed and D. L. McCubbrey, "Tne Cytocomputer: A Practical Pipelined Image Processor," Proceedings of the 7th Annual International Symposium on computer Architecture, La Boule, France, May 6-8 (1980) 271-277.
22. R. Eskenazi, and J. M. Wilf, "Low level Processing for real-time Image Analysis," Proceedings of the COMPSAC 79 conference, Chicago, Illinois, November 6-8, (1979) 340-343.
23. K. Luetjen, P. Gemmar, and H. Ischen, "FLIP - A Flexible Multiprocessor System for Image Processing", Proceedings of the 5th International Conference on Pattern Recognition, (1980) 326-328.
24. E. E. Swartzlander, Jr., B. K. Gilbert and I. S. Reed, "Inner Product Computers," IEEE Trans. on Computers, Vol. C-27, No. 1, January (1978) 21-31.
25. J. V. Blankenbaker, "Comments on Inner Product Computers," IEEE Trans. on Computers, Vol. C-28, No. 12, December (1979) 944.
26. K-I Mori, M. Kidode, H. Shinoda and A. Asada, "Design of local pattern processor for image processing," National Computer Conference 1978 (1025-1031).
27. H. J. Siegel, F. Kemmerer, and M. Washburn, "Parallel Memory System for a Partitionable MIMD/SIMD Machine," Proceedings of the 1979 International Conference on Parallel Processing, August (1979) 212-221.
28. H. J. Siegel, P. T. Mueller, Jr. and H. E. Smalley Jr., "Control of a Partitionable Microprocessor System," Proceedings of the 1978 International Conference on Parallel Processing, August (1978) 9-17.
29. F. A. Briggs, K-S Fu, K. Hwang and J. Patel, "PM^4--A reconfigurable multiprocessor system for pattern recognition and image processing," AFIPS Conference Proceedings, Vol. 48, 255-265.
30. G. R. Nudd, "Image Understanding architectures," National Computer Conference (1980), 377-390.
31. G. R. Nudd, F. A. Nygaard, G. D. Thurmond and S. D. Fouse, "A CCD Image Processor for Smart Sensor Applications," Proceedings of Photographic and Instrumentation Engineers Symposium, San Diego, August (1978).
32. T. J. Willett and G. Tisdale, "Hardware Implementation of a smart Sensor: A preview." ARPA Image Understanding Workshop, May (1978) 1-8.
33. W. E. Snyder, C. Husson and H. F. Benz, "Satellite Pattern Classification using charge Transfer Devices," Pattern Recognition and Image Processing Conference (1979), 246-249.
34. G. H. Granlund, "GOP: A Fast and Flexible Processor for Image Analysis", Proceedings of the 5th International Conference on Pattern Recognition, (1980), 489-492.
35. K. S. Fu, "Special Computer Architectures for Pattern Recognition and Image Processing - An Overview", National Computer Conference, Vol. 47, (1978), 1003-1013.
36. P. H. Swain, H.J. Siegel, and J. El-Achkar, "Multiprocessor Implementation of Image Pattern Recognition - A General Approach", Proceedings of the 5th International Conference on Pattern Recognition", (1980), 309-317.
37. B. Kruse, "Experience with a Picture Processor in Pattern Recognition Processing", National Computer Conference, Vol. 47, (1978), 1015-1024.
38. J. Keng and K.S. Fu, "A Special Computer Architecture for Image Processing", Proceedings of the 1978 Pattern Recognition and Image Processing Conference, Chicago, (1978), 287-290.
39. P. Marks, "Low-Level Vision Using an Array Processor", Computer Graphics and Image Processing," (1980), Vol. 14, 281-292.
40. S. F. Readdaway, "The DAP Approach", Infotech State of the Art Report on Supercomputers, Vol. 2, (1979), 836-840.
41. K. E. Batcher, "STARAN Parallel Processor System Hardware", Proceedings of the National Computing Conference, (1974), 405-410.
42. J. L. Potter, "The STARAN Architecture and its Application to Image Processing and Pattern Recognition Algorithms", Proceedings of the National Computer Conference, Vol. 47, (1978), 1041-1047.
43. B. Kruse, B. Gudmundsson, and D. Antonsson, "FIP - The PICAP II Filter Processor", Proceedings of the 5th International Conference on Pattern Recognition, (1980), 484-487.
44. D. Sherdell, "A Low Level Architecture for a Real Time Computer Vision System", Proceedings of the 5th International Conference on Pattern Recognition, (1980), 290-295.

SIGNAL PROCESSING WITH SYSTOLIC ARRAYS(a,b)

R. W. Priester
Systems and Measurements Division
Research Triangle Institute
Research Triangle Park, NC 27709

H.J. Whitehouse
Naval Ocean Systems Center
Catalina Boulevard
San Diego, CA 92152

K. Bromley
Naval Ocean Systems Center
Catalina Boulevard
San Diego, CA 92152

J. B. Clary
Systems and Measurements Division
Research Triangle Institute
Research Triangle Park, NC 27709

Abstract -- This paper discusses the application of systolic array processors to signal processing problems that are amenable to a matrix formulation. Systolic arrays are formed by providing nearest-neighbor interconnections between a large number of elemental processors to form either a one - or two-dimensional array. With the possible exception of boundary elements each processing element performs identical computations in synchronism with other elements in the array. A number of important problems for which systolic arrays hold potential are mentioned and the systolic array processor definition, in a number of its forms, is reviewed. When applied to strongly band-limited matrices, systolic array processors can be characterized as highly efficient from the standpoint of both hardware utilization and algorithm time. However, as the bandwidth becomes large this high performance is degraded. In an effort to overcome performance degradation, this paper introduces and evaluates a data transformation which, when applied to an n x n dense matrix, results in an improved banded structure with attendant hardware savings. An interesting feature of this transform is its invariance properties with respect to the ordering of output time sequences and algorithm execution time. Another interesting aspect is its relation to the classical Gauss-Seidel's method of iteration.

It is shown that systolic array processors possess some efficient testability features which can be exploited concurrently. These are briefly summarized.

1.0 Introduction

This paper discusses the application of systolic array architectures to signal processing problems.

Introduced by Kung [1], systolic array architectures provide the capability for realizing a number of important matrix operations. In addition to achieving a high computation rate by means of pipelining and concurrent computation, the architecture is a good candidate for implementation with VLSI (very large scale integration) technology. If the matrices processed are characterized by a narrow bandwidth, excellent hardware utilization efficiency can be achieved. However, in those cases where the matrix bandwidth becomes appreciable, for instance in the case of square densely-populated matrices, hardware utilization efficiency is degraded significantly. This paper addresses the problem of using systolic arrays to process matrices whose structure is less constrained. A simple but effective data transform which can in some instances significantly improve hardware utilization efficiency is introduced and developed.

The paper is organized as follows. Section 2.0 presents a brief and general discussion of several problem areas where the systolic array architecture is of interest. Section 3.0 outlines the main features of the systolic array architecture and only summarizes the extensive treatment given in [1,2]; this section is included only for purposes of completeness of presentation. The PRT (partial row translation) data transform is introduced and developed in detail in Section 4.0. Section 4.0 also quantitatively compares the efficiency of the original systolic array processor with that which results from applying the PRT transform. These results provide a means for deciding when PRT is advantageous. Matrix inversion is the topic of Section 5.0 while Section 6.0 briefly outlines an efficient technique that is useful for testing some systolic array matrix processors.

2.0 Matrix Operations in Signal Processing Applications

Matrix operations represent a significant portion of the computational burden encountered in many signal processing applications. Adaptive filtering, data compression, beamforming, and cross-ambiguity calculation represent problem areas where stable matrix analysis techniques are of current interest. In terms of resources required for system

[a] The work reported in this paper was sponsored by the Naval Ocean Systems Center, San Diego, CA, under contract N66001-80-C-0118 with the Research Triangle Institute.

[b] This paper was presented at the Tactical Airborne Distributed Computing and Networks Conference (AGARD), held in Roros, Norway, 22-26 June, 1981.

0190-3918/81/0000/0207$00.75 © 1981 IEEE

implementation, these problems can be classified as memory intensive and computation intensive. Construction of systems capable of providing the computations required for analysis of the above problems must provide for such operations as matrix multiplication, inversion, addition and various decompositions.

For example, in least squares approximation problems, one might encounter matrix multiplication, matrix inversion, and/or singular value decomposition. The computational approach used in a particular instance depends upon the numerical stability properties of the problem at hand. For instance, if the order of a particular problem is sufficiently small, the Gauss normal equations might be solved by performing a straightforward matrix inversion. However, in the solution of ill-conditioned systems commonly encountered in large-scale problems, achieving a meaningful solution might require application of singular value decomposition computations.

In [3] Speiser and Whitehouse discussed the signal processing problems mentioned above and considered the applicability of competing architectures such as transversal filters, array processors, bus-organized multiprocessors and systolic array architectures. Of these, the most promising architecture is that of the systolic array which has the potential to support real-time implementation of the algorithms required in order to address those problem areas mentioned in this section.

3.0 The Systolic Array Architecture

In the interest of a self-contained presentation, the systolic array architecture will be outlined and illustrated in this section. A thorough, comprehensive treatment can be found in [1] or in [2]. The systolic array architecture is founded almost exclusively upon a single computational element--the inner product step processor--which implements the relation

$$y^{k+1} = a_{k+1} \bullet x_{k+1} + y^{k}; \quad k = 0\ , 1, 2, \ldots, n-1. \qquad (1)$$

Systolic array processors are constructed by appropriately interconnecting a group of inner product step processors. In the systolic array architecture, only nearest-neighbor processor communication is permitted. For purposes of data communication and computation, each inner product step processor is equipped with three data registers: R_y (for y), R_a (for a_k) and R_x (for x_k). Each register has two connections - one for input, the other for output. Kung [1] defined two types of inner product step processors which are illustrated in Fig. 1. These elemental processors can be connected in a number of ways which provide the capability to perform various matrix operations such as matrix multiplication, L-U decomposition of symmetric positive-definite matrices, and the solution of triangular linear systems of equations.

A basic unit of time measure for both types of processors shown in Fig. 1 is defined as follows: (a) the processor loads inputs y^k, x_k and a_k, into R_y, R_x, and R_a respectively, (b) y^{k+1} is computed according to (1), and (c) y^{k+1}, x_k, and a_k are output.

As an example, a systolic array matrix-vector processor will be configured to form the product

$$y = Ax \qquad (2)$$

using a linearly connected group of Type 1 processors. The relations which must be implemented are as follows

$$y_i^{k+1} = a_{i,\ k+1} \bullet x_{k+1} + y_i^{k}; \quad k = 0, 1, 2, \ldots, n-1$$

$$y_i^{0} = 0 \qquad (3)$$

$$y_i = y_i^{w}; \qquad i = 1, 2, \ldots, n.$$

Fig. 2 illustrates the systolic array of processors, the element data arrangements and flow required to evaluate (2) for the case where A is an n x n matrix with bandwidth w = p + q - 1 = 4. Definition of p and q are as follows:

$$p = \max(j-i+1),\ a_{ij} \neq 0 \text{ for } j \geq i$$
$$q = \max(i-j+1),\ a_{ij} \neq 0 \text{ for } j \leq i.$$

The y_i enter the array from the right as zero and accumulate so as to form the inner product of the ith row of A with vector x which moves to the right after being input from the left. As the x and y vectors move through the array in the manner noted, A is shifted downward such that elements along the main diagonal pass through P_2. In general elements of A above and parallel to the main diagonal pass through processors to the left of P_2. Similarly elements of A below and parallel to the main diagonal pass through processors to the right of P_2. A detailed example illustrating the operation of this systolic array matrix-vector processor will be presented in Section 4.0.

Generalization of the linearly-connected systolic array to a two-dimensional orthogonally-connected structure enables the evaluation of matrix-matrix products. A systolic array for evaluating

$$C = AB \qquad (4)$$

where all matrices are n x n is shown in Fig. 3. Matrix A is input to the systolic array in exactly the same way as described earlier for the matrix-vector processor while columns of B are input, with appropriate spatial shift to allow for A's time delay, into successive rows of the array. If B contains a large number of columns this implementation can be inefficient even for stronly banded matrices. Kung [1] overcame this problem by devising the hexagonal-connected systolic array which is based upon the type 2 processor of Fig. 1. An example of this processor is presented in Fig. 3(b) for the case (4) when A, B and C are strongly banded. Note the direction of flow orientation of A, B and C. Entries in C are accumulated as it is shifted upward from the bottom of the array, where the c_{ij} enter with zero value.

Using the array structures presented above, Kung [1] was able to realize two additional important matrix operations. Due to space limitations, these only will be mentioned here. A triangle equation solver can be constructed using a linearly connected array of inner product step processors; however, it is necessary to introduce a new processor capable of division. The resulting processor solves a nonsingular triangular system of linear equations by back-substitution. Similarly, by adding special elements on the upper portion of the periphery of the hexagonal array (Fig. 3b), Kung [1] showed that one can obtain the following matrix decomposition

$$A = LU$$

where A is a symmetric, positive definite matrix,
L is lower triangular having 1s on the main diagonal, and
U is upper triangular.

Therefore, this processor, when coupled with the triangle equation solver, can be used to solve a fairly general class of simultaneous equations.

Table 1 summarizes the hardware requirements and algorithm execution time steps for the family of systolic array processors defined by Kung. When considered from the standpoint of hardware uniformity, a surprising degree of capability is realized by the systolic array architecture. For the case of strongly banded matrix structures, this architecture is efficient in terms of both the quantity of hardware used and in hardware utilization efficiency. However, if square dense matrices or matrices of more general structure are considered, hardware utilization efficiency can be degraded considerably. This problem is addressed in the remainder of this paper where methods for improving implementation efficiency are introduced and studied.

4.0 Definition and Development of the PRT Transform

In this section the PRT (partial row translation) transform will be defined and some of the benefits available from its application in connection with systolic arrays will be presented. It will be shown to improve hardware utilization efficiency and in addition provide a hardware savings in the case of square dense matrices.

Definition of the PRT Transform

Consider the matrix-vector multiplication problem stated in (2) with A constrained to be n x n and densely populated. Express A as a strictly subdiagonal part, A_L (i.e. with no diagonal elements) juxtaposed with A_U, the upper triangular part of A which contains the main diagonal elements of A. This may be expressed as follows

$$A = \begin{bmatrix} \ddots & A_U \\ A_L & \ddots \end{bmatrix} \qquad (5)$$

Applying the PRT transform to (5) provides

$$A_{PRT} = \begin{bmatrix} \ddots & A_U & \ddots & 0 \\ 0 & \ddots & A_L & \ddots \end{bmatrix} \qquad (6)$$

That is, A_{PRT} is obtained from A simply be translating (i-1) elements in row i to the right n positions within the row for i = 2, 3, ..., n. In the resulting $n \bullet (2n - 1)$ array, all elements not specified by A_U and the displaced A_L are set to zero. Now, applying the PRT transform to (2) yields, the equivalent expression

$$y = A_{PRT}\, x_{PRT} = A_{PRT} \begin{bmatrix} x \\ \hline x_p \end{bmatrix} \qquad (7)$$

where $x_p = (x_1, x_2, \ldots, x_{n-1})$. It is noted that the PRT converts a square array into a nonsquare array with enhanced banded structure. The transform necessitates augmenting x with a partial copy, x_p. A detailed example where A is $4 \bullet 4$ is detailed in Fig. 4. Four processors are used and the required number of time steps is eleven. These quantities compare favorably with Kung's systolic array which would use seven processors and also eleven time steps. For n large, it follows that the PRT transform saves about n/2 inner product step processors with no increase in execution time. If the original systolic array were designed such that immediately upon processing element a_{nn}, the values of y contained in the array could be unloaded, a time advantage would result for this processor configuration. The corresponding PRT based array, while saving about one-half the number of processors, would incur only about a 50% increase in execution time.

The PRT transform readily extends to the problem of evaluating the product of two square matrices as expressed in (4). It can be shown that the resulting systolic array for this problem is identical to that of Fig. 3a. The only difference occurs in the way A and B are input to the array. The PRT is applied to A which saves about $n^2/2$ processors and the columns of B, input on the left side of the array are partially repeated as prescribed in (7). Due to the large number of connections which would be required to immediately unload this two dimensional array, the PRT configured processor will evaluate the matrix-matrix product without any time penalty compared with the original systolic array.

Although they will not be discussed here, the PRT transform can be advantageously applied to some problems where nonsquare matrices are encountered.

Quantitative Assessment of the PRT Transform

The remainder of this section will be devoted to a quantitative comparison of the performance of the sytolic array processor proposed by Kung [1] (hereafter called original and denoted in certain instances by the subscript orig) with that of the PRT based structure (henceforth called alternate and denoted by subscript alt). The comparisons to be made will be based upon the following three figures of merit:

(a) Processor utilization efficiency η_{orig} and η_{alt}.
(b) Space-Time product $(ST)_{orig}$ and $(ST)_{alt}$ where
S = number of inner product step processors
T = number of algorithm time steps.
(c) Overall figure of merit $F = \eta/(ST)$, $Q = F_{alt}/F_{orig}$.

In the comparisons which follow, no penalty or cost is assigned to implementing the PRT transform. Also it is assumed that n is large.

First consider the matrix-vector problem which is shown for both processor configurations in Fig. 5. Adjacent to each processor configuration expressions for η, S, and T are given. η is defined as the ratio of active area to the total area as shown in the figure. Simply stated it is an approximate measure of the proportion of algorithm time for which computations are performed. Only square matrices are considered here with bandwidth $w = p + q - 1$. Note also that the comparisons made here assume processor initialization as illustrated.

Fig. 6 presents plots of as a function of the normalized bandwidth parameters $y = p/n$ and $x = q/n$. This figure is drawn under the assumption that the array of the original configuration may be unloaded immediately after element a_{nn} has been processed. Alternately, Fig. 7 presents the same information except that immediate unloading of the original configuration is not allowed. The results show that the capability to immediately unload the array is important when $x, y \longrightarrow 1.0$. Note that the original configuration provides excellent efficiency for x and y both small, that is, for strongly banded matrices; however, as $x, y \longrightarrow 1.0$ the alternate form is superior.

Now consider a comparison on the basis of (ST) product. Solving the relation $(ST)_{orig} = (ST)_{alt}$ provides the result plotted in Fig. 8. When the pair (x,y) lie above the curve, the alternate configuration provides a smaller (ST) product.

Generally it will be desirable to maximize the quantity $F = \eta/(ST)$ for a given problem. Therefore, Fig. 9 shows a plot of $Q = F_{alt}/F_{orig}$ versus y with x a parameter. Given x and y for a particular problem these results clearly indicate the preferred processor configuration.

Attention is now directed to the matrix multiplication problem where it is required to evaluate C = AB when both A and B are n x n dense matrices. For the sake of simplicity, the general case of banded matrices will not be treated in this comparison. Three systolic array configurations will be considered.

(a) A PRT-based orthogonally-connected processor
(b) The orthogonally-connected processor shown in Fig. 3(a).
(c) The hex-connected processor presented in Fig. 3(b).

The quantities of interest for comparing these three configurations (subsequently referred to as configuration (a), (b) and (c)) are tabulated in Table 2. (Note in Table 2 that the double subscript on Q is interpreted to mean $Q_{ab} = F_a/F_b$ where a and b refer to the configurations listed above). From these results the PRT-based systolic array is seen to offer significant performance advantages with respect to configurations (b) and (c) under the conditions specified.

5.0 Applications of Systolic Arrays to Matrix Inversion

This section will consider both explicit and implicit methods for solving a given consistent set of linear equations. By explicit it is meant that the inverse matrix is made available to the user while implicit is used to imply that only the solution vector is determined and made available.

The hexagonally connected systolic array mentioned earlier can be used to explicitly invert a given symmetric, positive-definite matrix. The approach is discussed by Speiser and Whitehouse [3] and can be summarized as follows. First the L-U decomposition of the given matrix is formed using the hex-connected systolic array. Then using n appropriately interconnected triangle equation solvers, L^{-1} can be computed. In this step the input to the array of triangle equation solvers, i.e. the known input vectors taken collectively, forms the identity matrix. U^{-1} is computed in a similar manner, and finally the inverse matrix is obtained by taking the matrix product $U^{-1}L^{-1}$. All of these steps can be implemented using systolic arrays.

Implicit matrix inversion can be performed in several ways, the most direct consisting of L-U decomposition followed by two executions using a triangle equation solver. That is, given

$Ax = b$:	A and b known
$LUx= b$:	LU decomposition step
$Ly = b$:	solve for y using triangle equation solver
$Ux = y$:	solve for x using triangle equation solver

This method, while it does not explicitly provide A^{-1} is generally more accurate than the explicit method which computes $x = A^{-1}b = U^{-1} L^{-1}b$ [4]. Other implicit techniques such as Jacobi's method, Gauss- Seidel's method and the successive overrelaxation (SOR) method [5] can be realized with systolic arrays. Implementation of Gauss-Seidel's method is interesting because it is closely related to the PRT transform. Consider the equation $Ax = b$. Factoring A into the form $A = D(L + I + U)$ where L and U are strictly lower and upper triangular matrices respectively (i.e., their main diagonal elements are zero) and D is a diagonal matrix $D = \text{diag}\ (a_{ii})$, $a_{ii} \neq 0$, $i = 1, 2, \ldots, n$. Jacobi's method of iteration can be written in terms of these definitions as follows

$$x_i^{k+1} = (-L_i x^k - U_i x^k) + b_i/a_{ii},$$
$$i = 1, 2, \ldots, n \tag{8}$$

where L_i and U_i denote the ith rows of L and U respectively. Implementation of (8) using either the original or alternate forms for systolic array matrix-vector multiplication is straightforward, only requiring insertion of zeros along the main diagonal and evaluation of the terms b_i/a_{ii} outside the array as an auxiliary computation. The equations defining Gauss-Seidel's method are as follows [5]

$$x_i^{k+1} = (-L_i x^{k+1} - U_i x^k) + b_i/a_{ii}.$$
$$i = 1, 2, \ldots, n. \tag{9}$$

Here the notation is identical to that in (8) except that in the term $L_i x^{k+1}$, x^{k+1} represents only a partially filled vector $(x_1^{k+1}, x_2^{k+1}, \ldots, x_{i-1}^{k+1}, 0, \ldots)$ which is "built up" as the computation proceeds. Gauss-Seidel's iteration can be implemented in systolic array form by using the PRT transform. This is illustrated in Fig. 10 which shows that the diagonal elements have been omitted and the terms b_i/a_{ii} are evaluated outside the array. Assuming that the computation is started with an initial estimate x^k, it can be observed from Fig. 10 that x_1^{k+1} will be output and available for processing by the strictly subdiagonal elements L. (For a detailed example of this property see Fig. 4 and note that in the present case $x_1^{k+1} = y_1$, is output at time step 5. Note also that this value of y_1 is required in time step 6 for processing by a_{21}, which in the present case is L_2). Since U always processes a backdated estimate, it can be seen that the PRT transform, or some equivalent method, must be applied in order to realize Gauss-Seidel's method using systolic arrays. That is, unless the elements of L can be moved to the input side of the array where the x_i^{k+1} are input, the pipelining effect of the array prohibits implementing Gauss-Seidel's method. Therefore, the original form of the systolic array cannot, without modification, be used to implement Gauss-Seidel's iterative method.

Note from Fig. 10 that Gauss-Seidel's implementation can provide extremely efficient utilization of processor capability. Processor utilization efficiency, starting at 83%, monotically increases toward 100% as the number of iterations increase. Although not discussed earlier when matrix-vector processors were considered, a form similar to that shown in Fig. 10 can be obtained for the problem $y = Ax$ where A is $n \times m$ with $n \geq m$. For this case, input vector x is simply repeated the required number of times while the PRT transform is applied to successive $m \times m$ partitions of A.

The SOR method of solution by iteration is very similar to Gauss-Seidel's method, the most important distinction being that the systolic array in this case computes the residual error which is then weighed by a relaxation parameter appropriately chosen to accelerate convergence [5].

6.0 Concurrent Testing of Systolic Array Processors

Utilization of any functional device in realizing important system features ultimately leads to questions regarding reliability and maintainability properties. In this section interesting methods for externally testing systolic arrays for proper operation will be considered. It is not practical to consider reliability features here; therefore, only issues related to maintainability, namely testability, will be considered. Only external methods for testing will be explored.

Consider the systolic array for performing a matrix-vector product originally proposed by Kung [1]. Given the way in which the matrix rows pass through the processor array, a rather simple external test for proper operation of the array would be to augment the given matrix by adding two check rows--one at the top and another at the bottom. This is illustrated in Fig. 11 where the two additional rows must be identical in order to facilitate the check. Note from Fig. 11 that if no $x_i = 0$ and no augmentation element is zero, each processor will be checked in the process of performing the matrix-vector product. The test is very simple since it requires only that y_1 be compared for equality with y_{n+2}.

Two additional processors are required to realize this test. It is interesting to examine the cost required to implement this check in terms of added hardware and algorithm execution time. Let S represent the hardware required to realize a processor in the array and t denote the time interval required for each shift in passing the matrix through the processor. For an $n \times n$ dense matrix and using the product S (computation time) as a measure of resources used, then the efficiency is given by:

$$\eta = \frac{(S \bullet 2nt) \text{ without test}}{[S(2n+2)t] \text{ with test}}$$
$$\eta \simeq 1 - 2/n$$

For n large, it follows that this is a very efficient test in terms of required resources.

With respect to test effectiveness, however, questions follow with regard to fault coverage. If x is known to be dense and the augmentation does not use zero elements, the test will be good for detecting hard failures. However, transient failures represent a problem for this approach.

The test method just described can be applied to matrix-matrix processors, although comparison of more quantities must be made. It also follows that this approach is applicable to the PRT transform. Note for this case from Fig. 11, however, that for about n time steps no checks on the computation are performed. This can be overcome by additional augmentations, appropriately interspersed, in the original matrix.

7.0 Conclusion

Systolic arrays represent a potentially important means for implementing computations involving large-scale matrices. The realization of a general matrix-oriented computing capability that is founded upon a few standard modules using VLSI technology is appealing. However, as emphasized in [2, Sec. 8.2], minimization of wiring requirements (communication costs) is a central problem in this technology. The PRT transform introduced in this paper can significantly reduce these costs for some problems. Of particular importance is the fact that these savings can be realized in some cases without increasing algorithm time.

It has been shown that for n x n banded matrices the PRT-based systolic array and that originally proposed by Kung [1] are complimentary in the sense that when one is efficient, the other form tends toward lower efficiency. The PRT transform does not alter the original systolic array hardware definition. The time-ordered outputs are invariant under this transform - the only changes appearing in the order of accumulation of intermediate values before they are output at the array port(s).

Solution of linear, simultaneous equations by iteration methods using systolic arrays results in an interesting interpretation of the PRT transform. The PRT or some equivalent transform appears necessary in order to apply systolic arrays to Gauss-Seidel's method or to the SOR method.

A simple, efficient - though somewhat limited - testing technique was introduced for performing external concurrent tests on systolic arrays. This topic, as well as the others considered in this paper, is worthy of further study.

References

[1] H. T. Kung and C. E. Leiserson, "Systolic Arrays for (VLSI)," Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA., April, 1978, (Last Revised December 1978).

[2] C. Mead and L. Conway, Introduction to VLSI Systems, Addison-Wesley, 1980. Section 3 of Chapter 8 closely follows [1], although the terminology systolic array is not used.

[3] J. M. Speiser and H. J. Whitehouse, "Architectures for Real-Time Matrix Operations," GOMAC (Government Microcircuit Applications Conference), 1980 Digest of Papers, pp. 21-26.

[4] A. H. Sameh, "Numerical Parallel Algorithms -- A Survey," in High Speed Computer and Algorithm Organization, eds. Kuck, Lawrie, and Sameh, Academic Press, N.Y., 1977.

[5] G. Dahlquist and A. Bjorck, Numerical Methods, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1974.

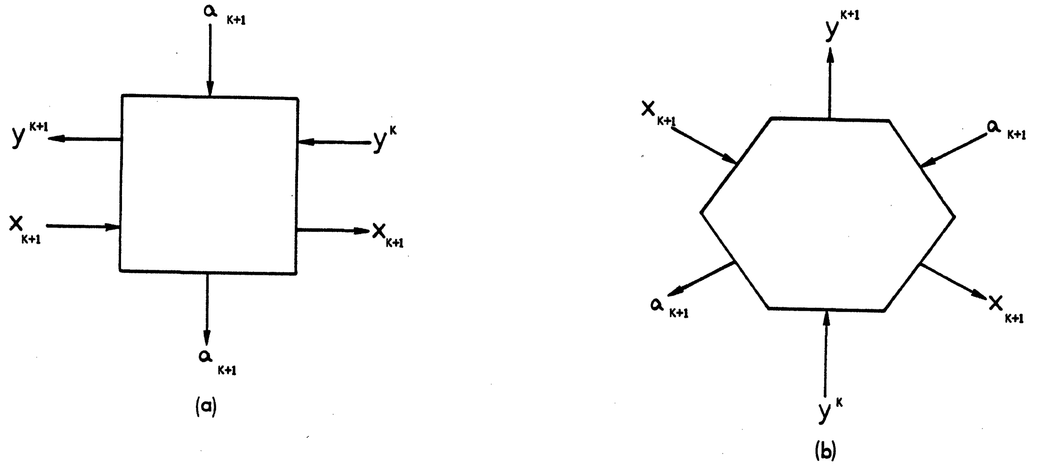


Fig. 1. Two Types of Inner Product Step Processors: (a) Type 1, (b) Type 2.

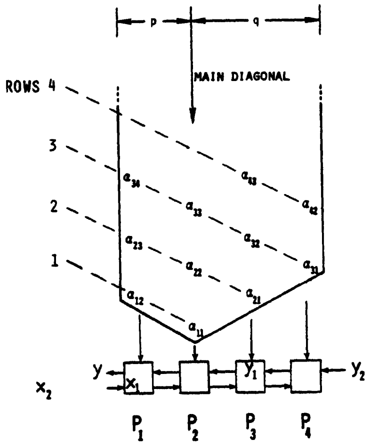
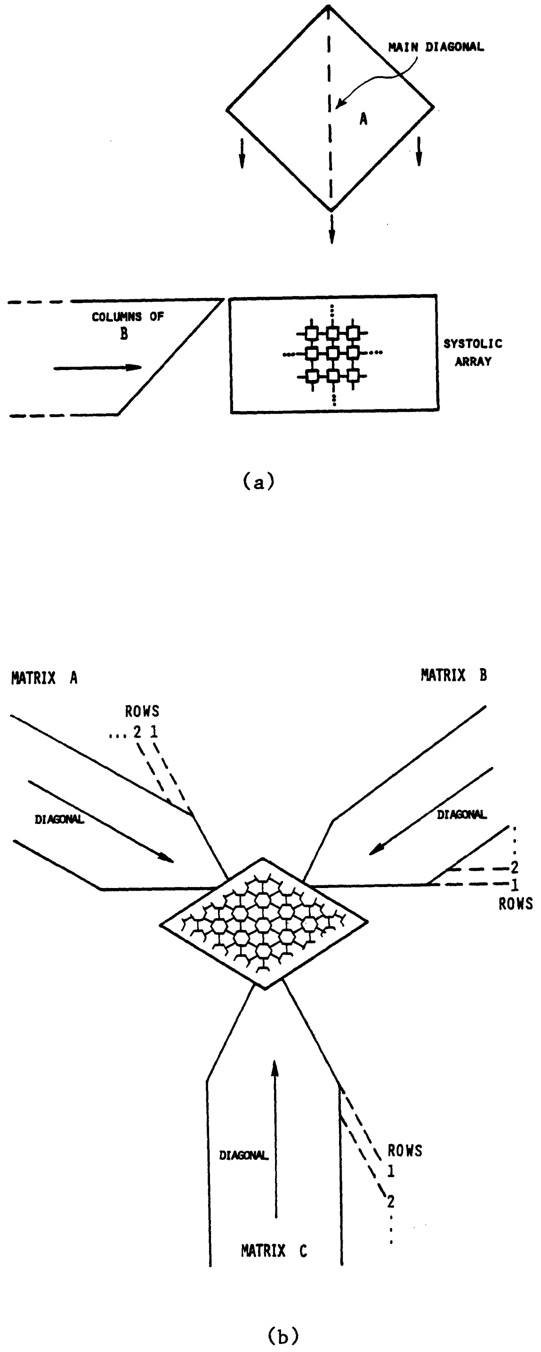


Fig. 2. Systolic Array Processor Configured to Form Matrix-Vector Product.



(a)

(b)

Fig. 3. Systolic Arrays to Compute Matrix Product C = AB: (a) Orthogonally Connected, (b) Hexagonally Connected.

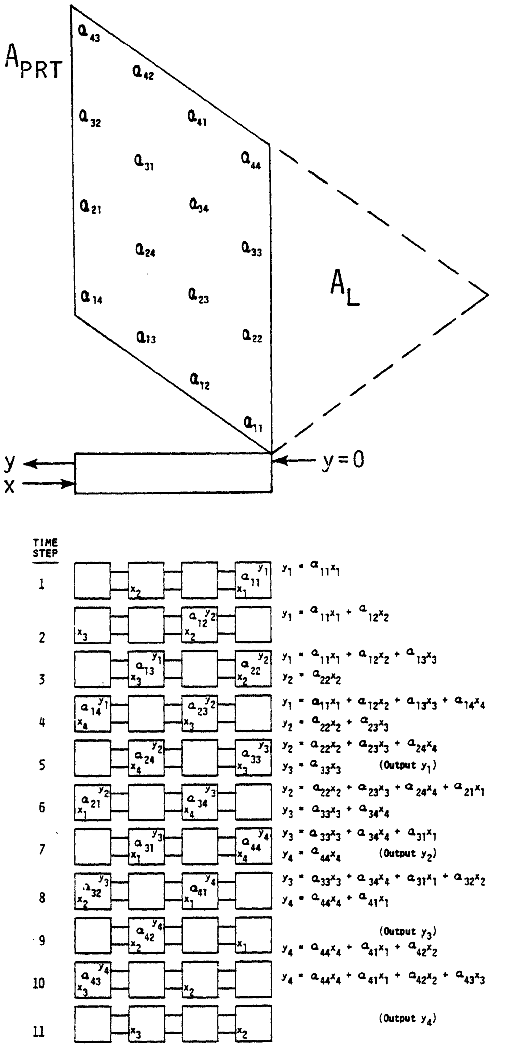
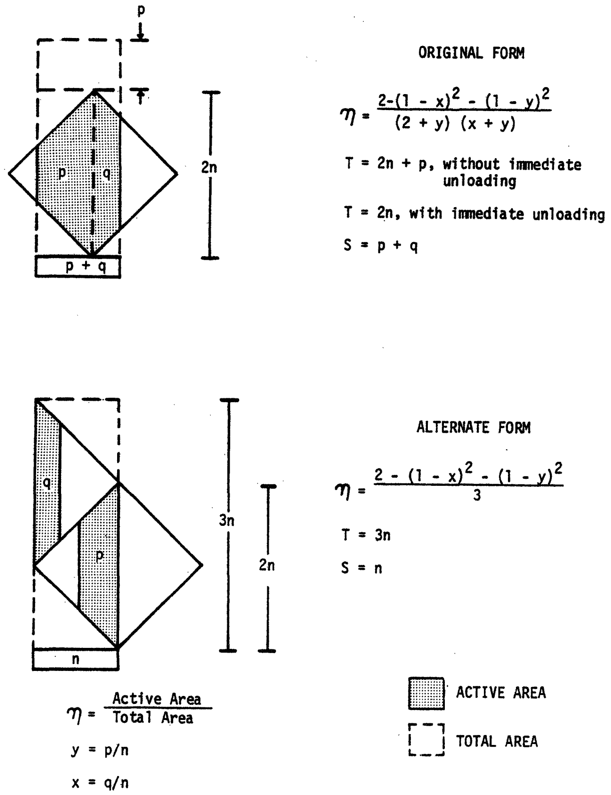

A_{PRT}
a_{43}
a_{42}
a_{32}
a_{41}
a_{31}
a_{44}
a_{21}
a_{34}
a_{24}
a_{33}
a_{14}
a_{23}
A_L
a_{13}
a_{22}
a_{12}
a_{11}
y
x
y=0
TIME STEP
1
2
3
4
5
6
7
8
9
10
11
$y_1 = a_{11}x_1$
$y_1 = a_{11}x_1 + a_{12}x_2$
$y_1 = a_{11}x_1 + a_{12}x_2 + a_{13}x_3$
$y_2 = a_{22}x_2$
$y_1 = a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4$
$y_2 = a_{22}x_2 + a_{23}x_3$
$y_2 = a_{22}x_2 + a_{23}x_3 + a_{24}x_4$
$y_3 = a_{33}x_3$ (Output y_1)
$y_2 = a_{22}x_2 + a_{23}x_3 + a_{24}x_4 + a_{21}x_1$
$y_3 = a_{33}x_3 + a_{34}x_4$
$y_3 = a_{33}x_3 + a_{34}x_4 + a_{31}x_1$
$y_4 = a_{44}x_4$ (Output y_2)
$y_3 = a_{33}x_3 + a_{34}x_4 + a_{31}x_1 + a_{32}x_2$
$y_4 = a_{44}x_4 + a_{41}x_1$
(Output y_3)
$y_4 = a_{44}x_4 + a_{41}x_1 + a_{42}x_2$
$y_4 = a_{44}x_4 + a_{41}x_1 + a_{42}x_2 + a_{43}x_3$
(Output y_4)

Fig. 4. Detailed Example of PRT Transform and Linearly Connected Systolic Array to Compute y = Ax.



ORIGINAL FORM

$$\eta = \frac{2-(1-x)^2-(1-y)^2}{(2+y)(x+y)}$$

T = 2n + p, without immediate unloading

T = 2n, with immediate unloading

S = p + q

ALTERNATE FORM

$$\eta = \frac{2-(1-x)^2-(1-y)^2}{3}$$

T = 3n

S = n

$$\eta = \frac{\text{Active Area}}{\text{Total Area}}$$

y = p/n

x = q/n

Fig. 5. Performance Comparison of Original with Alternate Systolic Array for Matrix-Vector Problem with Square Banded Matrix.

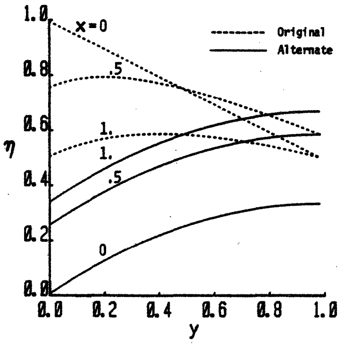


Fig. 6. Processor Utilization Efficiency Assuming Immediate Unloading.

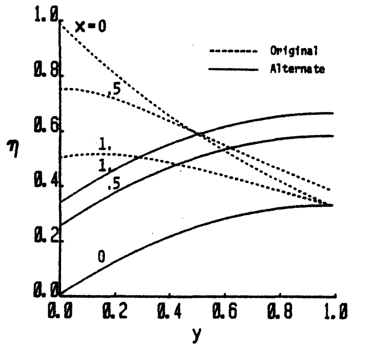


Fig. 7. Processor Utilization Efficiency Without Immediate Unloading.

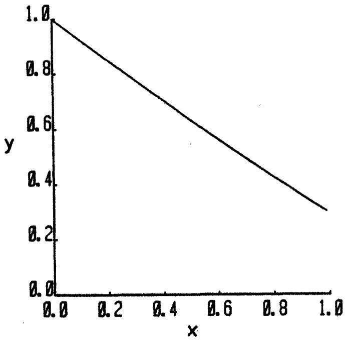


Fig. 8. Values of x,y which Satisfy $(ST)_{orig} = (ST)_{alt}$ Without Immediate Unloading.

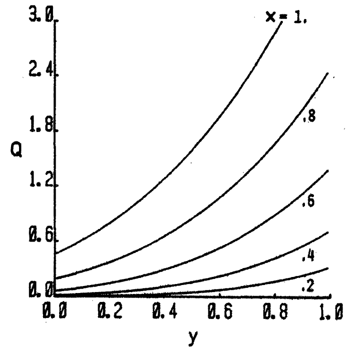


Fig. 9. Figure of Merit $Q = F_{alt}/F_{orig}$ Without Immediate Unloading.

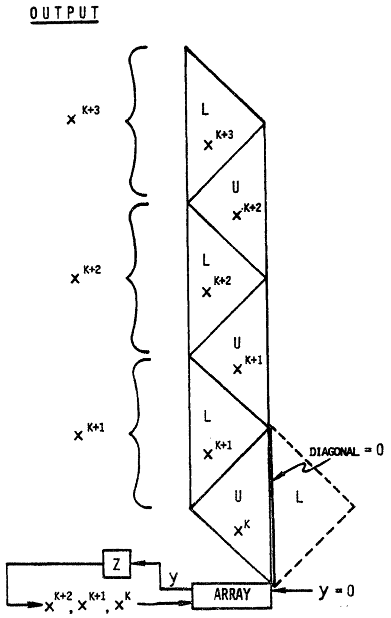
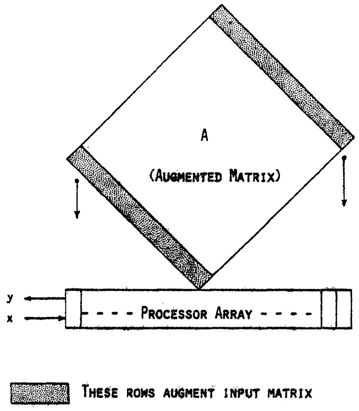


Fig. 10. Three Iterations of Gauss-Seidel's Method with Initial Estimate x^k (External Computations Performed in Block Z).



Test Implementation: Compare y_1 with y_{n+2} for equality

Fig. 11. Concurrent Testing of Systolic Array Matrix-Vector Processor by Augmentation Method.

Table 1. Summary of Systolic Array Hardware and Algorithm Execution Time Requirements for Some Matrix Problems.

Systolic Array Configuration	Problem Solved	No. of Processors Required	Algorithm Time
Linearly Connected Array	Matrix-Vector Mult.	w	2n+w
Linearly Connected Array	Sol. of Triangular System	w	2n+w
Orthogonally Connected Array	Matrix-Matrix Mult.	nM	3n+M
Hexagonally Connected Array	Matrix-Matrix Mult.	$w_A w_B$	3n+M
Modified Hexagonally Connected Array	L-U Decomposition A = LU	pq	3n+m

Note: (a) Matrices are assumed n x n with bandwidths $w = p+q-1$. Subscripted w denotes bandwidth of indicated matrix.

(b) Matrix-Matrix Multiplication either $C = AB$ or $C' = B'A'$.

(c) $M = \min(w_A, w_B)$, $m = \min(p,q)$.

Table 2. Comparison of Systolic Array Configurations for Matrix-Matrix Multiplication (all matrices n x n).

Quantity of Interest	Configuration (a)	(b)	(c)
T	5n	5n	4n
S	n^2	$2n^2$	$4n^2$
η	2/3	1/2	1/8

$Q_{ab} \simeq 2.7$

$Q_{ac} \simeq 17$

Note: Tables 1 and 2 do not reflect processor inactivity (see [2, Sec. 8.3]). If these effects are considered and the lower bound on S is used for configuration (c) (see [2, p.305]) one obtains: $Q_{ab} \simeq 2.7$ and $Q_{ac} \simeq 8$.

PARALLEL PROCESSING OF THE KALMAN FILTER

Angus Andrews
Science Center
Rockwell International Corporation
Thousand Oaks, California 91360

Abstract – This paper presents a pipelined mechanization for a multiprocessor system to implement the Kalman filter equations for adding process noise and updating the estimates after observations. These parallel algorithms use the UDL decomposition of the covariance matrix of estimation uncertainty. Parallelism can be used within the pipeline to further increase processing speed. The fastest method for the n-state filter requires O(log n) add-times per scalar update. Serial methods require $O(n^2)$ multiply-add-times per update.

Introduction

The Kalman filter is an optimal linear estimator that was introduced by R.E. Kalman in 1960 [1]. It provides a real-time mechanization for estimating the n-dimensional state vector x of a discrete-time linear gaussian system

$$x_k = \Phi_k x_{k-1} + G_k u_k \qquad (1)$$

given m-dimensional observation vectors

$$z_k = H_k x_k + v_k \qquad (2)$$

and the covariance matrices Q_k, R_k of the gaussian processes $\{u_k\}$, $\{v_k\}$. The estimate $\hat{x}$ is updated by the following well known formulas:

$$\hat{x}_k := \hat{x}_k + K_k(z_k - H_k\hat{x}_k) \qquad (3)$$

$$K_k := P_k H_k^T (H_k P_k H_k^T + R_k)^{-1} \qquad (4)$$

$$P_k := (I - K_k H_k) P_k \qquad (5)$$

$$\hat{x}_{k+1} := \Phi_{k+1}\hat{x}_k \qquad (6)$$

$$P_{k+1} := \Phi_{k+1} P_k \Phi_{k+1}^T + G_k Q_k G_k^T \qquad (7)$$

where P_k is the covariance of estimation uncertainty. Update of the estimate following an observation is mechanized by Eqs. (3-5), and propagation of the estimate in time is mechanized by Eqs. (6-7). Although these equations describe the theoretically optimum linear estimator, they are not necessarily well suited to numerical implementation in finite precision. Schlee [2] and others have observed numerical instability of these equations, and much of the subsequent work by Joseph [3], Schmidt [4], and others has been directed toward more accurate and efficient algorithms for mechanizing these equations.

Potter [5] introduced the idea of using a *square root* of the covariance matrix in the algorithmic implementation. This is a matrix

$$S = P^{\frac{1}{2}} \quad \text{such that} \quad P = SS^T . \qquad (8)$$

The advances in "square-root" filtering up to 1971 have been summarized by Kaminski, *et al.* [6]. Subsequently, Agee and Turner [7], Carlson [8], and Bierman [9] have introduced strictly algorithmic approaches to square-root filtering. This paper shows how a variation of the Bierman algorithm can be implemented in a pipeline architecture, and how the same architecture can be used for adding the "process noise" covariance Q_k of Eq. (7), using the Agee-Turner algorithm.

Observation Update

The Kalman filter is a recursive algorithm for updating the estimate. Equations (3-7) must be implemented at each recursion step. Agee and Turner introduced the idea of using a recursive algorithm for implementing Eqs. (3-7) as well. They also introduced the idea of using a UDL (or LDU) decomposition of the covariance matrix in place of the square root decomposition. This is a decomposition of the sort

$$P = UDU^T \qquad (9)$$

where D is a diagonal matrix and U is an upper triangular matrix with 1's along its main diagonal. This factorization does not require taking scalar square roots. Bierman derived a recursive algorithm for implementing Eqs. (3-5) in terms of U and D, rather than P. It assumes that the covariance matrix R of measurement uncertainty is a diagonal matrix. This form can always be obtained from the UDL decomposition of R. If

$$R = \tau\Delta\tau^T \qquad (10)$$

is such a decomposition, then for $T = \tau^{-1}$ the alternate observation $z := Tz$ with alternate observation sensitivity matrix $H := TH$ has a diagonal covariance matrix of observation uncertainty $R := \Delta$. When R is diagonal, each component of z can be processed serially as an independent scalar observation. It is this feature that allows the update equations to be pipelined.

The following is a variation of the Bierman algorithm. It performs the update mechanization on U, D, and x, given z, H, and R.

0190-3918/81/0000/0216$00.75 © 1981 IEEE

```
for j:=1 step 1 until m do
begin
   y:=z(j);                                  (11)
   w:=R(j,j);
   for k:=1 step 1 until n do
   begin
      s:=H(j,k);
      y:=y-s*x(k);                           (12)
      for i:=1 step 1 until k-1 do
         s:=s+U(i,k)*H(j,i);                 (13)
      d:=s*D(k,k);                           (14)
      K'(k,j):=d                             (15)
      a:=w;
      w:=w+s*d;                              (16)
      c:=-s/a;                               (17)
      D(k,k):=D(k,k)*a/w;                    (18)
      for i:=1 step 1 until k-1 do
      begin
         u:=U(i,k);
         U(i,k):=u+c*K'(i,j);                (19)
         K'(i,j):=K'(i,j)+u*d                (20)
      end
   end;
   y:=w/y;                                   (21)
   for i:=1 step 1 until n do
      x(i):=x(i)+y*K'(i,j)                   (22)
end;
```

The reader is referred to Bierman's book for a proof of the validity of the algorithm. Bierman's algorithm performs Eqs. (12-13) in separate loops. The modification nests the do-loops so that all computations sweep from left to right across the columns of U and D. These computations involve only one column of U at a time, and this feature allows one to pipeline the process. While the p^{th} column of U is being updated with information from the q^{th} rows of R, H, and z, the $(p-1)^{th}$ column of U can be updated with information from the $(q-1)^{th}$ rows of R, H, and z. Simultaneously, the $(p-2)^{th}$ column of U can be updated with information from the $(q-2)^{th}$ rows of R, H, and z, and so forth. This data flow is illustrated by Figure 1, which shows how the different data from R, H, z, U, D, and x comes together for arithmetic operations in the above algorithm.

Figure 1 is not meant to imply a particular multiprocessor architecture, but merely to illustrate the relative data flow as viewed from the matrix U. The diagonal of the matrix D is shown replacing the main diagonal of U. During the arithmetic processing, the transposed rows of H flow through successive columns of U from left to right, along with the partially computed columns of the unweighted Kalman gain matrix K'. The associated scaling coefficients w are computed in the flow down the diagonal. By adjoining the estimated state $\hat{x}$ and scalar observation z on the right of the U-D matrix, one can perform the following equivalent form of Eq. (3) in the last column:

$$x := x + K'D_w(z - Hx) \qquad (23)$$

where

$$D_w = \begin{bmatrix} w_1 & 0 & \cdot & \cdot & \cdot & 0 \\ 0 & w_2 & \cdot & \cdot & \cdot & 0 \\ \cdot & \cdot & \cdot & & & \cdot \\ \cdot & \cdot & & \cdot & & \cdot \\ \cdot & \cdot & & & \cdot & \cdot \\ 0 & 0 & \cdot & \cdot & \cdot & w_m \end{bmatrix} \qquad (24)$$

and w_1, w_2, $\cdots$, w_m is the order of arrival of weighting coefficients in the last (x) column.

The data flow in Figure 1 suggests a pipeline mechanization in which the data in the columns of U, D, and x remain within the pipeline and the rows of R, H, and z flow through the pipeline. This arrangement seems a practical one, because U, D, and x are the "permanent" variables of the Kalman filter and R, H, and z are only temporary data. Also, U and D are usually considered "nuisance" parameters in that they are necessary for the processing but otherwise of no interest. Therefore, they can remain within the pipeline without need of access from outside. The estimate x is available at the end of the pipeline.

In this pipeline mechanization, the essential pipeline processor element is associated with a column of the U-D matrix or x. The arithmetic processes involving a column of Figure 1 must be completed before the results are available for the next column. The order of execution of arithmetic computations involving a row of H and a column of the U-D matrix or x is given in Table 1. This mechanization could be implemented by using one processor for each square in Figure 1, much like the "systolic" processor arrays of H.T. Kung [10]. The required data flow between processors in each column is shown by the vertical arrows in Figure 1. Such a mechanization would not make efficient use of the processors, however. It would require n process-times to implement Eq. (12), and n process-times for the w-coefficient in Eq. (22) to make its way to x(1). Therefore, it would require O(n) processors to perform O(n) arithmetic operations in O(n) time, and the average processor utilization factor would be $O(n^{-1})$.

The arithmetic processor utilization can be improved to O(1/log n) and the update execution time shortened to O(log n) by using an array processor to form the dot products of Eqs. (11-13). All multiplies can be done simultaneously, and n binary adds can be performed in $\log_2 n$ add-times. All other arithmetic operations do not depend upon n. The fastest serial methods for the observation update require $O(n^2)$ multiplies and adds, for either the square-root [9] or conventional [3] methods.

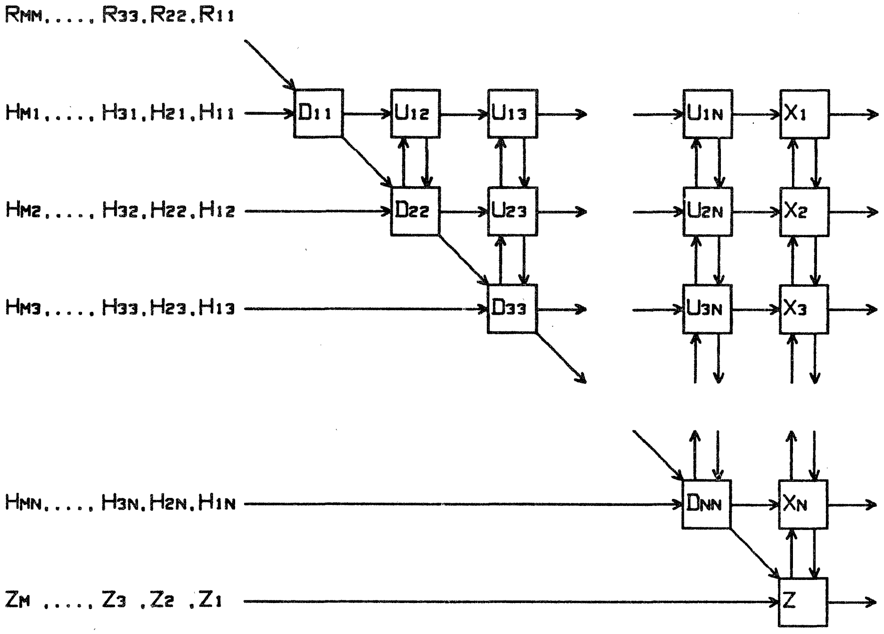

RMM, ... , R33, R22, R11
HM1, ... , H31, H21, H11
HM2, ... , H32, H22, H12
HM3, ... , H33, H23, H13
HMN, ... , H3N, H2N, H1N
ZM, ... , Z3, Z2, Z1
D11
U12
U13
U1N
X1
D22
U23
U2N
X2
D33
U3N
X3
DNN
XN
Z

Figure 1. Relative Data Flow in Observation Update

Table 1. Execution Order

Order of Execution	Equation Numbers	
	Columns 1 – n	Column (n+1)
1	13	11-12
2	14-17	21
3	18-20	22

Filling the pipeline requires O(log j) arithmetic operations for computations in the jth column. Filling n columns then requires

$$O\left(\sum_{j=1}^{n} \log j\right) = O(n \log n) \qquad (25)$$

arithmetic operation times. Therefore, when one adds the time required for filling and clearing the pipeline, processing m independent scalar observations requires O((n+m)log n) arithmetic operation times.

Adding Process Noise

The variate u_k of Eq. (1) is called "process noise." The Kalman filter uses the covariance matrix Q of process noise in Eq. (7). Following an argument similar to that used for R, one can assume that Q is a diagonal matrix. (If it were not, then the factorization of Eq. (10) would lead to an equivalent formulation with $Q := \Delta$ diagonal and $G := GT$.) In that case,

$$GQG^T = \sum_{j=1}^{p} q_j g_j g_j^T \qquad (26)$$

where q_j is the j^{th} diagonal element of Q, g_j is the j^{th} column of G, and p is the column dimension of G. Therefore, it suffices to be able to perform the operation

$$P := P + qgg^T \qquad (27)$$

in terms of U and D. This is done by the following algorithm, due to Agee and Turner [7]:

```
for j:=1 step 1 until p do
begin
   e:=Q(j,j);
   for k:=n step -1 until 1 do
   begin
      d:=D(k,k);
      D(k,k):=d+e*G(k,j)↑2;                     (28)
      for i:=k-1 step -1 until 1 do
      begin
         G(i,j):=G(i,j)
            -G(k,j)*U(i,k);                     (29)
         U(i,k):=U(i,k)
            +e*G(i,j)*G(k,j)/D(k,k)             (30)
      end;
      e:=e*d/D(k,k)                             (31)
   end
end;
```

The data flow for this algorithm is illustrated by Figure 2, which shows how the data from U, D, G, and Q come together for arithmetic operations. The arithmetic operations of Eqs. (29-30) are to

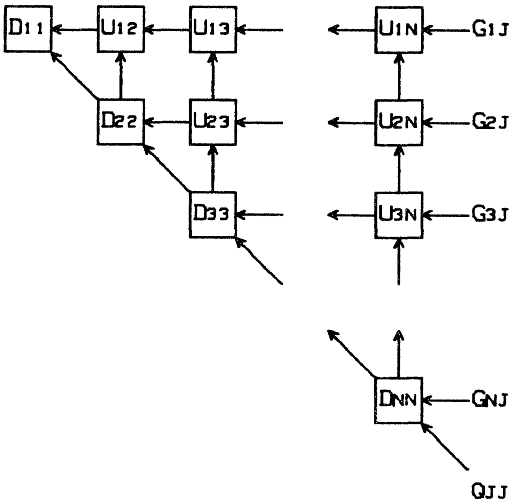


Figure 2. Relative Data Flow for Adding Process Noise

be performed in the off-diagonal rectangles, and those of Eqs. (28,31) in the diagonal rectangles. This process allows the same type of pipeline structure as the update, but with the direction of data flow reversed.

In this case, there are O(n) arithmetic operations required in the longest column of Figure 2, but they can be performed almost in parallel. The exception is that the factors G(k,j) and e*G(k,j)/D(k,k) must be computed in the diagonal squares before processing can start in the off-diagonal squares. Assuming that these data can be broadcast to the off-diagonal squares, the processing in the longest column could be completed in O(1) arithmetic operation times. Therefore, the minimum time required to add process noise covariance and clear the pipeline is O(n+p) operation times.

Remarks

One can show that it requires $O(n^2/\log n)$ arithmetic processor elements (APE) to attain the ultimate observation update speed in a multiprocessor system. Updating the j^{th} column of the U and D matrices requires O(j) adds and multiplies. Therefore, $O(n^2)$ adds and multiplies are required throughout the pipeline for updating the n columns, and performing these in O(log n) operation times (the ultimate speed) requires $O(n^2/\log n)$ APEs.

The time update is probably the greatest computational problem for general Kalman filter. The state dynamical equations (6-7) contain terms for deterministic (Φ) and nondeterministic (Q) dynamics. This paper has considered only the latter. The several serial solution methods which have been proposed for the triangular square root time update are mostly borrowed from square root least squares methods, such as Cholesky decomposition [11], modified Gram-Schmidt orthogonalization [4,12], and Householder [13] and Givens [14] transformations. Unfortunately, these presuppose formation of the product ΦU, which requires $O(n^3)$ arithmetic operations. The square root algorithms of Morf and Kailath [15] satisfy Eqs. (4,5,7) simultaneously, and only presuppose the products HL, ΦL for

$$L_k = (P_k - P_{k-1})^{\frac{1}{2}} . \qquad (32)$$

It is often the case that L_k can be maintained with column dimension $\alpha \ll n$, which would reduce the number of arithmetic operations to $O((m+n)n\alpha)$ prior to triangularization. (These methods require that Φ, H, R, G, and Q be constant, however.) Therefore, the fastest known parallel update with $O(n^2/\log n)$ APEs would be expected to require at least $O(\alpha \log n)$ arithmetic operation times for the time *and* observation update.

References

[1] Kalman, R.E., "A New Approach to Linear Filtering and Prediction Problems," Journal of Basic Engineering, Vol. 82D, March 1960, pp. 35-45.

[2] Schlee, F., Standish, C., and Toda, N., "Divergence in the Kalman Filter," AIAA Journal, Vol. 5, No. 6, June 1967, pp. 1114-1120.

[3] Joseph, P.D., "Space Control Systems — Attitude, Rendezvous and Docking," Lecture Notes, UCLA Engineering Extension, Los Angeles, California, 1964.

[4] Schmidt, S.F., "Computational Techniques in Kalman Filtering," NATO AGARDograph No. 139, February 1970, pp. 65-86.

[5] Battin, R.H., *Astronautical Guidance*, McGraw-Hill, 1964, pp. 338-340.

[6] Kaminski, P.G., Bryson, A.E., and Schmidt, S.F., "Discrete Square-Root Filtering: A Survey of Current Techniques," IEEE Transactions on Automatic Control, Vol. AC-16, No. 6, December 1971, pp. 727-735.

[7] Agee, W.S. and Turner, R.H., "Triangular Decomposition of a Positive Definite Matrix Plus a Symmetric Dyad with Applications to Kalman Filtering," U.S. Army, White Sands Missile Test Range, Technical Report No. 38, October 1972.

[8] Carlson, N.A., "Fast Triangular Formulation of the Square Root Filter," AIAA Journal, Vol. 11, No. 5, September 1973, pp. 1259-1265.

[9] Bierman, G.J., *Factorization Methods for Discrete Sequential Estimation,* Academic Press, New York, 1977.

[10] Mead, M. and Conway, L., *Introduction to VLSI Systems,* Addison-Wesley, 1980.

[11] Benoit, "Note sur une methode de resolution des equations normales provenant de l'application de la methode des moindres carres a un systeme d'equations lineares en nombre inferieur a celui des inconnues," Bulletin Geodesique, Toulouse, 1923, pp. 67-75.

[12] Björck, A., "Solving Least Squares Problems by Gram-Schmidt Orthogonalization," *BIT,* Vol. 17, 1967, pp. 1-21.

[13] Businger, P. and Golub, G.H., "Linear Least Squares Solution by Householder Transformations," Mathematics of Computation, Vol. 20, 1966, pp. 5-12.

[14] Gentleman, W.M., "Least Squares Computations by Givens Transformations Without Square Roots," Journal of the Institute for Mathematics and Its Applications, Vol. 12, 1973, pp. 329-336.

[15] Morf, M. and Kailath, T., "Square-Root Algorithms for Least-Squares Estimation," IEEE Transactions on Automatic Control, Vol. AC-20, No. 4, August 1975, pp. 487-497.

ON THE REARRANGEABILITY OF A (2log N-1) STAGE PERMUTATION NETWORK*

Kyungsook Yoon Lee
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

Abstract

Both the rearrangeability proof and the control algorithm are well known for the Benes network which is intrinsically symmetric. However, there has been little progress for the case of nonsymmetric networks of similar hardware requirement.
We provide a proof on the rearrangeability of a (2log N-1) stage network. Our proof does not depend on the symmetry of the network and can be applied to nonsymmetric as well as symmetric networks. We develop a global approach, one advantage of which is that it leads naturally to the idea of the rearrangeability proof and a control algorithm. For ease of understanding and presentation, the reduced $\Omega_N\Omega_N^{-1}$ is chosen to show the proof method. The Ω_N^{-1}-passable permutations are first characterized and the bit control algorithm emerges as the 'natural' control algorithm for such permutations. By a simple reinterpretation into Ω_N of the Ω_N^{-1}-passable condition, unique control algorithm for the reduced Ω_N to transform an arbitrary permutation into an Ω_N^{-1}-passable permutation is obtained.

The hardware requirement of the reduced $\Omega_N\Omega_N^{-1}$ is (Nlog N-N+1) switches which is the lower bound for rearrangeable networks. Though our algorithm has the same time complexity of O(Nlog N) for single control and O(2N) for multiple control as the looping algorithm of the Benes network, it is simpler because calculations of inverse mappings are not required and it is easier to understand because the switches are set stage by stage.

1. Introduction

An interconnection network is rearrangeable if its permitted states realize every assignment of input points to output points [Bene64]. Our primary concern here is $N = 2^n$ input/output interconnection network used as a permutation network.

The Benes binary network (we will simply call it the Benes network hereafter), a member of Clos' three-stage networks [Clos53], is a rearrangeable permutation network with a well-defined control algorithm [OpTs71], and it requires near-optimum hardware [Waks68], [Joel68]. While the Benes network was proven to be a rearrangeable network long ago [Bene64], very little has been known about the rearrangeability of other multistage networks of the same hardware complexity. The topological difference--symmetry in the Benes network and nonsymmetry in the multistage networks, symmetry meaning that the left half and the right half of a network are the mirror images of each other--may well be the main reason. The Slepian-Duguid theorem [Bene65], [Bene75] on which the rearrangeability of the Benes network is founded, inherently applies only to symmetric networks.

In this paper, we provide a new method to understand and prove the rearrangeability or the universality [Sieg77] of (2log N-1) stage networks. In this method the first (log N-1) stages (FH) and the last log N stages (LH) of the network are controlled by two different control algorithms. All the permutations realizable on LH are characterized in terms of residue classes regarding the input permutation as on ordered set. The LH is controlled by the usual destination tag method [Lawr76], [Pate79]. To transform an arbitrary permutation into a LH-passable permutation, FH is controlled by a proper residue class partitioning. Though we shall use a symmetric network $\Omega_N\Omega_N^{-1}$ in the proof for ease of understanding and presentation, it is emphasized that the proof does not depend on the symmetry of the network.

Definitions and notations are given in Section 2. The Ω_N^{-1}-passable condition is derived in Section 3. In Section 4, the omega network Ω_N [Lawr76] with its last switching stage removed (we will call it the reduced Ω_N), is shown to be able to transform any permutation into an Ω_N^{-1}-passable permutation. The reduced $\Omega_N\Omega_N^{-1}$ control algorithm and more hardware redundancy are discussed in Section 5 and finally, in Section 6, the conclusion and the extension are described.

2. Notations and Definitions

Def. A Complete Residue System modulo m, CRS (mod m), is a set of m intergers which contains exactly one representative of each residue class mod m.

Def. A Complete Residue Partition, CRP is a partition of a CRS(mod 2m) into two CRS(mod m).

For a given CRS(mod 2m) there are $2^{(m-1)}$ different CRP's.

*This work was supported in part by the National Science Foundation under Grant No. US NSF MCS80-01561, and in part by the Univ. of California under Grant No. US DOE SBC UCAL 5498609.

We consider a N input and N output ($N = 2^n$) interconnection network consisting of finite number of switching stages and fixed connections. In this paper, we study only the case where each switching stage is N/2 of (2×2) switching elements. We can represent a n-stage network ν as

$$\nu = C^0E^0C^1E^1 \ldots C^{(n-1)}E^{(n-1)}C^n \quad ,$$

where C denotes a fixed connection and E denotes a switching stage and the superscript i specifies the i-th stage.

In particular, the inverse omega network [Lawr76] Ω_N^{-1} can be represented as

$$\Omega_N^{-1} = E^0 U E^1 U \ldots E^{(n-1)} U$$

where U is an unshuffle.

We shall use A^i to denote an ordered set of N input numbers, $(a_0^i, \ldots, a_{(N-1)}^i)$, which is the input permutation of the switching stage E^i. An Ω_8^{-1} is shown in Fig. 1. Throughout the paper we will be interested in the CRS properties of various partitions of A^i.

Def.

$A_{0,k}^i = \{a_k^i\}, \quad 0 \leq k < 2^n$

$A_{j,k}^i = A_{(j-1),2k}^i \cup A_{(j-1),(2+1)}^i, \; 0 \leq k < 2^{(n-j)}, \; 1 < j \leq n$

(See Fig. 2 for the pictorial representation of these partitions for n=3.) Thus $\{A_{j,k}^i | 0 \leq k < 2^{(n-j)})\}$ forms a $2^{(n-j)}$-partition of $A^i = \{a_k^i | 0 \leq k < 2^n\}$. By the definition,

$$A_{j,k}^i = \bigcup_{\ell} \{A_{(j-s),(2^sk+\ell)}^i | 0 \leq \ell < 2^s\} \quad , \; 0 \leq s \leq j. \qquad (1)$$

In particular,

$$A_{(n-i),k}^i = \bigcup_{\ell} \{A_{1,\ell}^i | 2^{(n-i+1)}k \leq \ell < 2^{(n-i+1)}(k+1)\}. \qquad (2)$$

By $A = \{x,y\} \equiv \{p,q\} \pmod m$, we mean that A consists of two elements x and y, congruent to p and q (mod m) respectively.

The relation between A^i and $A^{(i+1)}$ is decided by the effects of E and U. By $A^{(i+1)} = A^iEU$, we mean that $A^{(i+1)}$ is obtained from A^i permuted by E and then by U. If $A^{(i+1)} = A^i E U = (A^iE)U$, then

$$\{a_{2k}^i\, E,\; a_{2k+1}^i\, E\} = \{a_{2k}^i,\; a_{2k+1}^i\} \; ,$$

$$(a_k^{(i+1)},\; a_{(k+2^{(n-1)})}^{(i+1)}) = (a_{2k}^i\, E,\; a_{2k+1}^i\, E) \; ,$$

and so $\{a_k^{(i+1)}, a_{k+2^{(n-1)}}^{(i+1)}\} = \{a_{2k}^i, a_{2k+1}^i\}, 0 \leq k < 2^{(n-1)}$ (3)

An example of this relation for $n = 3$ is given in Fig. 3.

Thus, if $A^{(i+1)} = A^i E U$, $0 \leq i < n$,

then

$$A_{j,k}^i = A_{(j-1),k}^{(i+1)} \cup A_{(j-1),(k+2^{(n-j)})}^{(i+1)} \; , \qquad (4)$$

$$1 \leq j \leq n \; , \qquad 0 \leq k < 2^{(n-j)} \quad .$$

(Fig. 4 shows the relation for $n = 3$.) So,

$$A_{j,k}^i = \bigcup_{\ell} \{A_{(j-s),(k+\ell \cdot 2^{(n-j)})}^{(i+s)} | 0 \leq \ell < 2^s\} \; , \qquad (5)$$

$$1 \leq j \leq n \; , \quad 0 \leq k < 2^{(n-j)} \; , \quad 0 \leq s \leq j \quad .$$

In particular,

$$A_{j,k}^i \subset A_{(i+j),k}^0 \quad . \qquad (6)$$

Def. An input sequence $A^0 = (a_0^0, a_1^0, \ldots a_{(N-1)}^0)$ is an Ω_N^{-1}-passable permutation if $a_k^n = k$, $0 \leq k \leq N$, when $A^n = A^0\Omega_N^{-1}$.

The Ω_N^{-1}-control algorithm is as follows: each switch setting in the i-th stage is controlled by the i-th bit of the upper input in binary representation. If the control bit is 0, then the switch is set straight. If the control bit is 1, then the switch is set cross (Fig. 5). Refer to Fig. 6 for an example of Ω_8^{-1}-control algorithm for the destination permutation (7,2,4,1,3,6,0,5).

3. Ω_N^{-1}-Passability

In this section, we shall prove Theorem 1 on Ω_N^{-1}-passability through Lemma 1 and Lemma 2. Lemma 1 describes a characteristic of the Ω_N^{-1}-control algorithm. The Ω_N^{-1}-control algorithm works by sorting bits--if certain input conditions are satisfied, the lower i bits of the input numners to the i-th switching stage E^i are in ascending order, and moreover, the (i+1)-th bits of two numbers that share a switch are distinct. An example of Lemma 1 is shown in Fig. 6 for n=3.

Lemma 1

If $A_{j,k}^0$ is a CRS(mod 2^j), $1 \leq j < n$, $0 \leq k < 2^{(n-j)}$ and $A^{(i+1)} = A^iE^iU$, $0 \leq i < n$, where E^i is controlled by the Ω_N^{-1}-control algorithm, then $A_{1,k}^i \equiv \{p, p+2^i\}$ (mod $2^{(i+1)}$) for any $A_{1,k}^i \subset A_{(n-i),p}^i$, $0 \leq p < 2^i$, $1 \leq i < n$.

Proof By induction of i.

Suppose that $A_{1,\ell}^i \equiv \{q, q+2^i\}$(mod $2^{(i+1)}$) for any $A_{1,\ell}^i \subset A_{(n-i),q}^i$. Let $A_{1,k}^{(i+1)} \subset A_{(n-(i+1)),p}^{(i+1)}$. Then $2^{(n-(i+1)-1)} \cdot p \leq k < 2^{(n-(i+1)-1)} \cdot (p+1)$, $0 \leq p < 2^{(i+1)}$.

Case 1 $0 \leq k < 2^{(n-2)}$:

$2^{(n-(i+1)-1)} \cdot p < 2^{(n-2)}$, and so $p < 2^i$.

$A_{1,k}^{(i+1)} = \{a_{2(2k)}^{i} \cdot E^{i},\ a_{2(2k+1)}^{i} \cdot E^{i}\}$

$2^{(n-i-1)} \cdot p \le 2k,\ (2k+1) < 2^{(n-i-1)} \cdot (p+1)$

Therefore, $A_{1,2k}^{i},\ A_{1,(2k+1)}^{i} \subset A_{(n-i),p}^{i}$ and so $A_{1,2k}^{i},\ A_{1,(2k+1)}^{i} \equiv \{p, p+2^{i}\} (\mathrm{mod}\ 2^{(i+1)})$.

Thus, $a_{2(2k)}^{i} \cdot E^{i} \equiv a_{2(2k+1)}^{i} \cdot E^{i} \equiv p\ (\mathrm{mod}\ 2^{(i+1)})$ (since $p<2^{i}$). However, they must be distinct, $(\mathrm{mod}\ 2^{(i+2)})$, since $A_{1,k}^{(i+1)} \subset (A_{1,2k}^{i} \cup A_{1,(2k+1)}^{i}) = A_{2,k}^{i} \subset A_{(i+2),k}^{0}$ by (6) and $A_{(i+2),k}^{0}$ is a CRS$(\mathrm{mod}\ 2^{(i+2)})$. Hence,

$A_{1,k}^{(i+1)} \equiv \{p, p+2^{(i\_1)}\} (\mathrm{mod}\ 2^{(i+2)})$.

<u>Case 2</u> $2^{(n-2)} \le k < 2^{(n-1)}$

$2^{(n-2)} < 2^{n-(i+1)-1} \cdot (p+1)$, and so $2^{i} \le p < 2^{(i+1)}$.

$A_{1,k}^{(i+1)} = \{a_{2(2k')+1}^{i} \cdot E^{i},\ a_{2(2k'+1)+1}^{i} \cdot E^{i}\}$,

where $2k' = 2k - 2^{(n-1)}$.

$2^{(n-i-1)} \cdot (p-2^{i}) \le 2k',\ (2k'+1) < 2^{(n-i-1)} \cdot (p-2^{i}+1)$

$A_{1,2k'}^{i},\ A_{1,(2k'+1)}^{i} \subset A_{(n-1),(p-2^{i})}^{i}$

Therefore, $A_{1,2k'}^{i},\ A_{1,(2k'+1)}^{i} \equiv \{p-2^{i}, p\}$ $(\mathrm{mod}\ 2^{(i+1)})$.

Thus, $a_{2(2k')+i}^{i} \cdot E^{i} \equiv a_{2(2k'+1)+1}^{i} \cdot E^{i} \equiv p (\mathrm{mod}\ 2^{(i+1)})$, (since $p-2^{i}<2^{i}$ and $2^{i} \le p < 2^{(i+1)}$). Again $A_{1,k}^{(i+1)} \cup A_{1,(2k'+1)}^{i}) = A_{2,k'}^{i} \subset A_{(i+2),k'}^{0}$, and as $A_{(i+2),k'}^{0}$ is a CRS$(\mathrm{mod}\ 2^{(i+2)})$,

$A_{1,k}^{(i+1)} \equiv \{p, p+2^{(i+1)}\} (\mathrm{mod}\ 2^{(i+2)})$.

Since it can be shown in the same way as above that $A_{1,k}^{1} \equiv \{p, p+2\}\ (\mathrm{mod}\ 2^{2})$ for any $A_{1,k}^{1} \subset A_{(n-1),p}^{1}$, $0 \le p < 2$ as the induction basis, the lemma is proved.

Q.E.D.

Lemma 2 states that for any input sequence that is Ω^{-1}-passable, the lower i bits of the input numbers are in ascending order and the (i+1)-th bits of two numbers that share a switch are distinct for all the intermediate stages i. This indicates together with Lemma 1 that the Ω_N^{-1}-control algorithm the 'natural' control algorithm for Ω_N^{-1}-passable permutations.

<u>Lemma 2</u>

If $a_k^n = k$, $0 \le k < 2^n$ and $A^{i} = A^{(i+1)}E^{(i-1)}U$, then regardless of a particular switch seting of E^{i}, $A_{1,k}^{i} \equiv \{p, p+2^{i}\} (\mathrm{mod}\ 2^{(i+1)})$ for any $A_{1,k}^{i} \subset A_{(n-i),p}^{i}$, $0 \le p < 2^{i}$ and $1 \le i < n$.

<u>Proof</u> By induction of i.

Note that $A_{1,k}^{(n-1)} = \{a_k^n,\ a_{k+2^{(n-1)}}^{n}\} = \{k, k+2^{(n-1)}\} \equiv \{k, k+2^{(n-1)}\}\ (\mathrm{mod}\ 2^{n})$.

Suppose that $A_{1,\ell}^{(i+1)} \equiv \{q, q+2^{(i+1)}\}$ $(\mathrm{mod}\ 2^{(i+2)})$ for any $A_{1,\ell}^{(i+1)} \subset A_{n-(i+1),q}^{(i+1)}$. Let $A_{1,k}^{i} \subset A_{(n-i),p}^{i}$. Then $2^{(n-i-1)} \cdot p \le k < 2^{(n-i-1)} \cdot (p+1)$, $0 \le p < 2^{i}$. Now $A_{1,k}^{i} = \{a_{2k}^{i}, a_{(2k+1)}^{i}\} = \{a_k^{(i+1}, a_{k+2^{(n-1)}}^{(i+1)}\}$, $a_k^{(i+1)} \in A_{1,\ell}^{(i+1)}$ and $a_{k+2^{(n-1)}}^{(i+1)} \in A_{1,\ell+2^{(n-2)}}^{(i+1)}$ where ℓ is given by $k \in \{2\ell, 2\ell+1\}$. Since $2^{(n-(i+1)-1)} \cdot p \le \ell < 2^{(n-(i+1)-1)} \cdot (p+1)$, $2^{(n-(i+1)-1)} \cdot (p+2^{i}) \le \ell + 2^{(n-2)} < 2^{(n-(i+1)-1)} \cdot (p+2^{i}+1)$, therefore $A_{1,\ell}^{(i+1)} \subset A_{(n-(i+1),p)}^{(i+1)}$, $A_{1,\ell+2^{(n-2)}}^{(i+1)} \subset A_{n-(i+1),(p+2^{i})}^{(i+1)}$.

Hence, by the induction hypothesis,

$A_{1,\ell}^{(i+1)} \equiv \{p, p+2^{(i+1)}\} (\mathrm{mod}\ 2^{(i+2)})$ and

$A_{1,(\ell+2^{(n-2)})}^{(i+1)} \equiv \{p+2^{i}, p+2^{i}+2^{(i+1)}\} (\mathrm{mod}\ 2^{(i+2)})$,

and so $a_k^{(i+1)} \equiv p (\mathrm{mod}\ 2^{(i+1)})$ and $a_{(k+2^{(n-1)})}^{(i+1)} \equiv p+2^{i} (\mathrm{mod}\ 2^{(i+1)})$.

Thus, $A_{1,k}^{i} \equiv \{p, p+2^{i}\} (\mathrm{mod}\ 2^{(i+1)})$.

Q.E.D.

<u>Theorem 1</u> Ω_N^{-1}-passability:

An input sequence $A^{0} = (a_0^0, a_1^0, \ldots, a_{N-1}^0)$, is Ω_N^{-1}-passable iff $A_{j,k}^{0}$ is a CRS$(\mathrm{mod}\ 2^{j})$, $1 \le j < n$ and $0 \le k < 2^{(n-j)}$.

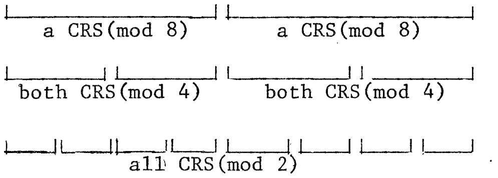
<u>Proof</u> <u>If</u>: We use the Ω_N^{-1}-control algorithm for switch settings. Then by Lemma 1, $A_{1,k}^{(n-1)} = \{a_{2k}^{(n-1)}, a_{(2k+1)}^{(n-1)}\} = \{k, k+2^{(n-1)}\}$, $0 \le k < 2^{(n-1)}$. Hence, $(a_k^n, a_{k+2^{(n-1)}}^{n}) = (a_{2k}^{(n-1)} \cdot E^{(n-1)},\ a_{(2k+1)}^{(n-1)} \cdot E^{(n-1)}) = (k, k+2^{(n-1)})$, $0 \le k < 2^{(n-1)}$. Thus $a_k^n = k$, $0 \le k < 2^n$ and the output sequence is Ω_N^{-1}-passable. <u>Only if</u>: By Lemma 2, $A_{1,k}^{i} = \{p, p+2^{i}\} (\mathrm{mod}\ 2^{(i+1)})$ for any $A_{1,k}^{i} \subset A_{(n-i),p}^{i}$, $0 \le p < 2^{i}$ and $1 \le i < n-2$. Consider $A_{1,k}^{0} = \{a_k^1.\ a_{k+2^{(n-1)}}^{1}\}$, and $a_k^1 \in A_{1,\ell}^{1}$, $a_{k+2^{(n-1)}}^{1} \in A_{1,(\ell+2^{(n-2)})}^{1}$ with ℓ given by $k \in \{2\ell, 2\ell+1\}$. Then $0 \le \ell < 2^{(n-2)}$ (since $k<2^{(n-1)}$) and $A_{1,\ell}^{1} \subset A_{(n-1),0}^{1}$, $A_{1,\ell}^{1} \equiv \{0,2\} (\mathrm{mod}\ 2^{2})$, ıd $A_{1,(\ell+2^{(n-2)})}^{1} \subset A_{(n-1),1}^{1}$, $A_{1,(\ell+2^{(n-2)})}^{1} \equiv \{1, 1+2\} (\mathrm{mod}\ 2^{2})$.

Thus $a^1_k \equiv 0 (\mathrm{mod}\ 2)$, $a^1_{(k+2^{(n-1)})} \equiv 1 (\mathrm{mod}\ 2)$ and $A^0_{1,k}$ is a CRS(mod 2). For $j>1$, $A^0_{j,k} = \bigcup_{\ell}\{A^{(j-1)}_{1,p(\ell)} \mid 0 \le \ell < 2^{(j-1)}$, $p(\ell) = k+\ell \cdot 2^{(n-j)}$, $0 \le k < 2^{(n-j)}$. Since $2^{(n-(j-1)-1)} \cdot \ell \le p(\ell) < 2^{(n-(j-1)-1)} \cdot (\ell+1)$, $A^{(j-1)}_{1,p(\ell)} \subset A^{(j-1)}_{n-(j-1),\ell}$ and $A^{(j-1)}_{1,p(\ell)} \equiv \{\ell, \ell+2^{(j-1)}\} (\mathrm{mod}\ 2^j)$. It follows that $A^0_{j,k}$ for $j>1$ is a CRS(mod 2^j).

Q.E.D.

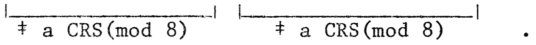
As an example of Theorem 1, consider a permutation, (3,4,5,6,7,8,9,10,11,12,13,14,15,0,1,2), which is an uniform shift of distance 3. It is Ω^{-1}_{16}-passable because

(3,4,5,6,7,8,9,10, 11,12,13,14,15,0,1,2)



The switch settings for the same permutation for n=3 is shown in Fig. 7. As another example of Theorem 1, a shuffle permutation (0,8,1,9,2,10,3,11,4,12,5,13,6,14,7,15) is not Ω^{-1}_{16}-passable because

(0,8,1,9,2,10,3,11, 4,12,5,13,6,14,7,15)



Usually network passability has been defined in terms of the bit relations between the source tags and the destination tags--a tag meaning the binary representation of a number [Lawr76], [Peas77], [YeLa80]. In Theorem 1 the Ω^{-1}-passable condition for an input permutation is given in terms of CRS, which is an easier tool to deal with. We can see easily that certain permutations, identity and uniform shift, are Ω^{-1}-passable while certain permutations, shuffle, unshuffle and bit-reversal, are not Ω^{-1}-passable. Clearly, identity permutation satisfies the CRS properties of Theorem 1 and the uniform shift preserves the CRS properties; while shuffle, unshuffle and bit-reversal violate the CRS properties inherently. As another example of the application of Theorem 1, the percentage of Ω^{-1}_N-passable permutations among N! permutations can be easily calculated by Theorem 1 and the use of combinatorics.

Def. The Ω^{-1}_N-passable condition for an input permutation $A^0 = (a^0_0, a^0_1, \ldots a^0_{N-1})$ is that $A^0_{j,k}$ is a CRS(mod 2^j), for $1 \le j \le n$ and $0 \le k < 2^j$.

The Ω^{-1}_N-passable condition which is defined over the original input sequence A^0, can be transformed into a condition for intermediate input sequences A^i in the following way.

Let $B^i_k = \{b^i_{k,\ell} \mid 0 \le \ell < 2^i\} \subset A^i$,

$b^i_{k,\ell} = a^i_{(k+\ell \cdot 2^{(n-i)})}$, $0 \le k < 2^{(n-i)}$, $1 \le i \le n$.

Thus B^i_k's, in addition to $A^i_{i,k}$'s, provide another $2^{(n-i)}$-partition of A^i. However, in contrast to $A^i_{i,k}$, elements of B^i_k are scattered all over A^i (see Fig. 8).

Lemma 3

The Ω^{-1}_N-passable condition is equivalent to the condition that each B^i_k is a CRS(mod 2^i), $0 \le k < 2^{(n-i)}$, $1 \le i \le n$.

Proof By (5), $A^0_{i,k} = \bigcup_{\ell} \{A^i_{0,(k+\ell \cdot 2^{(n-1)})} \mid 0<\ell<2^i\} = B^i_k$.

Q.E.D.

4. Rearrangeability of the Reduced $\Omega_N \Omega^{-1}_N$

We show that any input permutation can be transformed into an Ω^{-1}-passable permutation via the reduced Ω_N network. We only reverse the direction of input and output in Ω^{-1}_N, and consider it as Ω_N with input A^n and output A^0. This enables us to use all the relations derived in the previous section without any change. Now B^i_k's are natural partitions for Ω_N as can be seen in the relations

$$(b^i_{2k,\ell}, b^i_{(2k+1),\ell})E = (b^{(i+1)}_{k,\ell}, b^{(i+1)}_{k,(\ell+2^i)}), \quad 0 \le \ell < 2^i,$$
$$B^i_{2k} \cup B^i_{(2k+1)} = B^{(i+1)}_k, \quad 0 \le k < 2^{n-(i+1)}, \quad 0 \le i < n , \tag{8}$$

which follows from (7) and the unshuffle effects. If we consider the inverse of E as an Ω_N control, then (8) leads us to the needed control for the reduced Ω_N in virtue of Theorem 1 and Lemma 3. Let σ denote a shuffle permutation [Ston71].

Def. An exchange permutation E^i_R is defined by: if $A^i = (A^{(i+1)}\sigma)E^i_R$ and if every $B^{(i+1)}_k$ is a CRS(mod $2^{(i+1)}$), then every B^i_k is a CRS(mod 2^i).

Now $\{E^i_R \mid 0<i<n\}$ is the one and only control we have been looking for by the following theorem.

Theorem 2

$\{E^i_R \mid 0<i<n\}$ is the one and only kind of control to transform an arbitrary permutation, $A^n = (a^n_0, a^n_1, \ldots, a^n_{N-1})$ into an Ω^{-1}_N-passable permutation $A^0 = (a^0_0, a^0_1, \ldots, a^0_{N-1})$ via the reduced Ω_N.

Proof

Since $B_0^n = A^n$ is always a CRS(mod 2^n), by the definition of E_R^i, it follows that each B_k^i, $0\le k<2^{(n-i)}$, $1\le i\le n$ is a CRS(mod 2^i) when $A^i = A^{(i+1)} \sigma E_R^i$. Hence, by Lemma 3, $A^0 = (a_0^0, a_1^0, \ldots, a_{N-1}^0)$ is an Ω_N^{-1}-passable permutation. Conversely, any such control must be E_R^i at stage i by Theorem 1, Lemma 3 and (8). The fact that $0<i<n$, not $0\le i<n$, in Theorem 2, means that Ω_N^{-1}-passable condition is satisfied already for A^1. Thus, the switches of the last stage of Ω_N are redundant leaving us with the reduced Ω_N.

Q.E.D.

We now show that E_R^i is always realizable by a set of switches. The relation (8) states that E_R^i should be able to partition $B_{2k}^i \cup B_{2k+1}^i$ when it is a CRS(mod $2^{(i+1)}$) into two CRS(mod 2^i), B_{2k}^i and B_{2k+1}^i by partitioning proper $(b_{2k,\ell}^i, b_{2k+1,\ell}^i)$ at each switch element. Thus the following Lemma ensures the realization of E_R^i.

Lemma 4

Consider (a_k, b_k), $0\le k<2^{(i-1)}$, $\{a_k, b_k \mid 0\le k< 2^{(i-1)}\}$ = a CRS(mod 2^i). Then there is a permutation E such that $\{a_k E, b_k E\} = \{a_k, b_k\}$ for each k (i.e., E is an exchange permutation) and such that $\{a_k E \mid 0\le k<2^{(i-1)}\}$ is a CRS(mod $2^{(i-1)}$).

Proof

By construction. Let $a_k \equiv p_k (\mathrm{mod}\ 2^{(i-1)})$, $b_k \equiv q_k (\mathrm{mod}\ 2^{(i-1)})$, $0\le k<2^{(i-1)}$. Every number between 0 and $(2^{(i-1)}-1)$ occurs twice among p_k's and q_k's. Rearrange the pairs into groups in such a way that, in each group with more than one pair, any two adjacent pairs have a number in common and the first and the last pairs have a number in common. In such a group, if the common number of the 1st and the 2nd pairs occurs in the same position of eacg pair, then exchange positions of the two numbers of the seconf pair. By repeating this process between the second and the third pairs and so on, we obtain (p_k', q_k')'s, such that $(p_k', q_k') = \{p_k', q_k'\}$, $0\le k<2^{(i-1)}$ and all p_k''s are distinct.

$$\text{Set } (a_k \cdot E, b_k \cdot E) = \begin{cases} (a_k, b_k) & \text{if } p_k' = p_k\ , \\ (b_k, a_k) & \text{otherwise.} \end{cases}$$

Then by construction,

$$a_k \cdot E = p_k' (\mathrm{mod}\ 2^{(i-1)}), \quad 0\le k<2^{(i-1)}$$

and $\{a_k \cdot E \mid 0\le k<2^{(i-1)}\}$ is a CRS(mod $2^{(i-1)}$).

Q.E.D.

Note that in the exchange processes in the above proof one pair of numbers does not need an exchange, indicating the redundancy of a switch in each CRP of E_R^i. An example of Lemma 4 is given in Fig. 9 and an example showing the transformation of a non-Ω_8^{-1}-passable permutation into an Ω_8^{-1}-passable permutation is shown in Fig. 10.

In conclusion, the concatenation of the reduced Ω_N and Ω_N^{-1}, called the reduced $\Omega_N \Omega_N^{-1}$ is a (2log N-1) stage rearrangeable network.

Theorem 3

The reduced $\Omega_N \Omega_N^{-1}$ is a rearrangeable network.

Proof

By Theorem 1 and Theorem 2.

Q.E.D.

5. The Reduced $\Omega_N \Omega_N^{-1}$-Control Algorithm and Redundancy

In this section, we summarize the reduced $\Omega_N \Omega_N^{-1}$-control algorithm which has been described in the previous sections. We shall call a binary tree of CRP's a CRPT. A CRPT is a full binary tree of (n-1) levels whose root node is a CRP partitioning a CRS(mod 2^n) and the two sons of a node are two CRP's partitioning two CRS's produced by the parent node. Observe that $E_R^{(i-1)}$ in the previous section corresponds to the i-th level of a CRPT.

The reduced $\Omega_N \Omega_N^{-1}$-control algorithm.

1. The reduced Ω_N is controlled by a CRPT.
2. Ω_N^{-1} is controlled by the Ω_N^{-1}-control algorithm (Sec. 2).

An example of the reduced $\Omega_{16} \Omega_{16}^{-1}$-control algorithm is shown in Fig. 11 for the bit reversal permutation.

We discussed earlier, in the proof of Theorem 2, the redundancy of the last switching stage of Ω_N by which we obtained the reduced Ω_N. Further redundancy in switches was mentioned after Lemma 4.

The number of the CRP's needed for the reduced $\Omega_N \Omega_N^{-1}$-control algorithm is,

No. of CRP's for the reduced $\Omega_N \Omega_N^{-1}$ =

$$1+2+\ldots+2^{(n-2)} = 2^{(n-1)}-1 = (N/2-1)\ .$$

Thus the number of necessary switches in the reduced $\Omega_N \Omega_N^{-1}$ is,

No. of switches in the reduced $\Omega_N \Omega_N^{-1}$ =

$$(2n-1)\cdot N/2 - (N/2-1) = (N\log N-N+1)\ .$$

This is exactly the same as the number of switches required for the reduced Benes network [Waks68], [Joel68] which is a lower bound for rearrangeable networks.

The control by a CRPT requires O(Nlog N) time steps, and the Ω_N^{-1}-control requires O(log N) time steps. Therefore, the time complexity of the reduced $\Omega_N\Omega_N^{-1}$-control algorithm is O(Nlog N). When more hardware is available for control, the CRP's on the same level of a CRPT can be done in parallel reducing the CRPT control time and thus the overall control time to $O(N+N/2+\ldots+2)=O(2N)$. These control time complexities are in the same order of magnitude as those of the looping algorithms for the Benes network [OpTs71]. But our algorithm is simpler as inverse mappings are not needed, and it is easier to understand as the switches are set stage by stage.

A characteristic of a CRPT control is that any fixed connection at the beginning of a rearrangeable network is redundant. Therefore, the first shuffle of the reduced $\Omega_N\Omega_N^{-1}$ can be removed and the resultant network is still rearrangeable.

6. Conclusion and Extension

Employing a new global approach a proof of the rearrangeability of the reduced $\Omega_N\Omega_N^{-1}$ was given. The proof method can be used for nonsymmetric as well as symmetric networks. We have succeeded in constructing a rearrangeable network with the same hardware requirement as the reduced Benes network. A control algorithm with the same control time complexity as that of Opferman and Tsao-Wu's looping algorithm was described. Our control algorithm is simpler because calculations of the inverse mappings are not required, and it is easier to understand because the switches are set stage by stage.

We have considered only the $N=2^n$ case with (2×2) switches in this paper. An immediate generalization may be to the $N=p^n$ case with $(P\times P)$ switches for $p>2$ [Lee]. Another extension is being investigated on the proof of the rearrangeability for the networks resulting from concatenation of two delta networks [Pate79] based on the CRPT related equivalence relation [Lee].

We could have used the known network equivalence relations to obtain many rearrangeable networks with the minimum hardware requirement [Lee]. Some networks surely will be nonsymmetric (for ex. $B\Omega^{-1}$). One of the networks that can be obtained in this way is the reduced $\Omega\rho\Omega$. Parker showed that $\Omega\rho\Omega$ is rearrangeable [Park80] without giving a control algorithm. Now we have a control algorithm for the (2log N-1) stage reduced $\Omega\rho\Omega$ where the reduced Ω is controlled by a modified CRPT and the second Ω is controlled by a single bit control algorithm.

Even the Benes network can be thought of as another (2log N-1) multistage network, and if we use a similar approach, we get a new Benes network control algorithm [Lee]. Thus, the notion of symmetry and recursiveness which differentiated the Benes network from other multistage networks disappears and they can be treated as a family of rearrangeable permutations networks with the minimum hardware requirement.

Now the bottleneck of the control algorithm lies in the set partitioning problem. If we can find a fast set partitioning algorithm, the network control time can be greatly reduced. Also, the non-uniqueness of the CRP might be useful for fault-tolerant network designs.

Acknowledgment

The author would like to express sincere gratitude to Professor David J. Kuck for his encouragement and guidance.

References

[Bene64] V. E. Benes, "Permutation Groups, Complexes, and Rearrangeable Connecting Networks," Bell System Tech. Journal, Vol. 43, No. 4, pp. 1619-1640, July 1964.

[Bene65] V. E. Benes, Mathematical Theory of Connecting Networks and Telephone Traffic, Academic Press, NY, 1965.

[Bene75] V. E. Benes, "Proving the Rearrangeability of Connecting Networks by Group Calculations," Bell System Tech. Journal, Vol. 54, No. 2, pp. 421-434, Feb. 1975.

[Clos53] C. Clos, "A Study of Non-Blocking Switching Networks," Bell System Tech. Journal, Vo. 32, No. 2, pp. 406-424, March 1953.

[Joel68] A. E. Joel, "On Permutation Switching Networks," Bell System Tech. Journal, Vol. 47, No. 5, pp. 813-822, May-June 1968.

[Lawr76] D. H. Lawrie, "Access and Alignment of Data in an Array Processor," IEEE Trans. on Computers, Vol. C-25, No. 12, pp. 1145-1155, Dec. 1976.

[Lee] K. Y. Lee, Ph.D. Thesis, in preparation.

[OpTs71] D. C. Opferman and N. T. Tsao-Wu, "On a Class of Rearrangeable Switching Networks, Part I: Control Algorithm," Bell System Tech. Journal, Vol. 50, No. 5, pp. 1579-1600, May-June 1971.

[Park80] D. S. Parker, "Notes on Shuffle/Exchange Type Switching Networks," IEEE Trans. on Computers, Vol. C-29, No. 3, pp. 213-222, March 1980.

[Pate79] J. H. Patel, "Processor-Memory Interconnections for Multiprocessors," Prof. 6th Annual Symp. on Computer

Architecture, New York, NY, pp. 168-177, April 1979.

[Peas77] M. C. Pease, "The Indirect Binary n-Cube Microprocessor Array," IEEE Trans. on Computers, Vol. C-26, No. 5, pp. 458-473, May 1977.

[Sieg77] H. J. Siegel, "The Universality of Various Types of SIMD Machine Interconnection Networks," Proc. 4th Annual Symp. on Computer Architecture, pp. 70-79, March 1977.

[Ston71] H. S. Stone, "Parallel Processing with the Perfect Shuffle," IEEE Trans. on Computers, Vol. C-20, pp. 153-161, Feb. 1971.

[Waks68] A. Waksman, "A Permutation Network," Journal of the ACM, Vol. 15, No. 1, pp. 159-163, Jan. 1968.

[WuFe79] C. L. Wu and T-y. Feng, "The Reverse-Exchange Interconnection Network," Proc. of the Int'l. Conf. on Parallel Processing, pp. 160-174, 1979.

[YeLa80] P-C. Yew and D. H. Lawrie, "An Easily Controlled Network for Frequently Used Permutations," Proc. of the Workshop on Interconnection Networks, Lafayette, IN, pp. 72-73, April 1980.

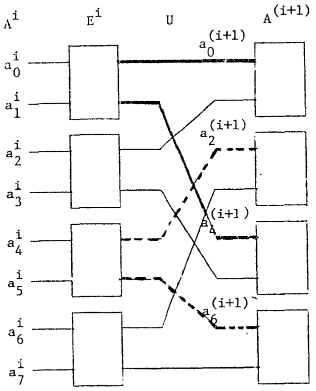


Fig. 3 . Effect of an Unshuffle $\{a_0^{(i+1)}, a_4^{(i+1)}\} = \{a_0^i, a_1^i\}$, $\{a_2^{(i+1)}, a_6^{(i+1)}\} = \{a_4^i, a_5^i\}$, ...

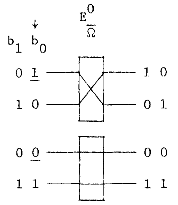


Fig. 5 . The bit b_i of the upper input number to a switch controls the switch setting for the i-th stage E^i. The control bits are underlined.

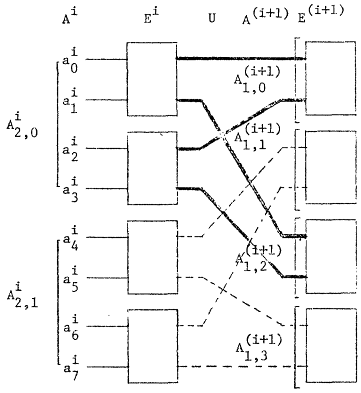


Fig. 4 . Example of Formulae (4)

$$A^i_{2,0} = A^{(i+1)}_{1,0} \cup A^{(i+1)}_{1,2}$$

$$A^i_{2,1} = A^{(i+1)}_{1,1} \cup A^{(i+1)}_{1,3}$$

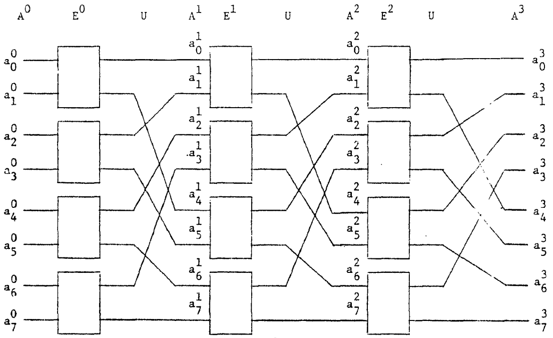


Fig. 1 . An Ω_8^{-1} Network

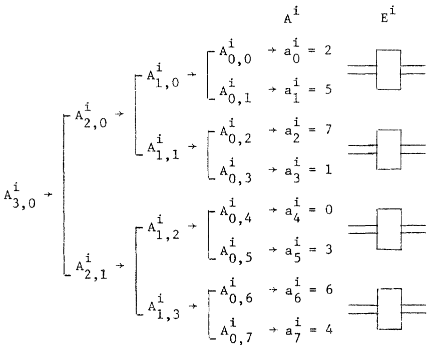

A^i
E^i
$A^i_{3,0}$
$A^i_{2,0}$
$A^i_{2,1}$
$A^i_{1,0}$
$A^i_{1,1}$
$A^i_{1,2}$
$A^i_{1,3}$
$A^i_{0,0} \rightarrow a_0^i = 2$
$A^i_{0,1} \rightarrow a_1^i = 5$
$A^i_{0,2} \rightarrow a_2^i = 7$
$A^i_{0,3} \rightarrow a_3^i = 1$
$A^i_{0,4} \rightarrow a_4^i = 0$
$A^i_{0,5} \rightarrow a_5^i = 3$
$A^i_{0,6} \rightarrow a_6^i = 6$
$A^i_{0,7} \rightarrow a_7^i = 4$

Fig. 2 . Partitions of A^i, $A^i_{j,k}$ for n=3, $0\le j\le n$, $0\le k<2^{(n-j)}$

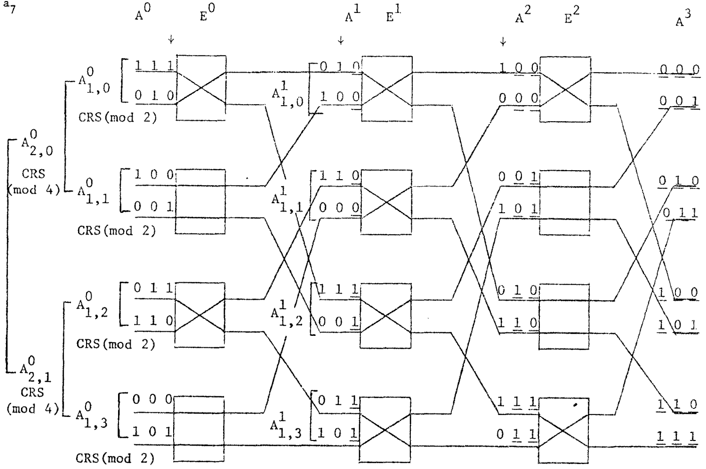


Fig. 6 . An Example of Lemma 1 for n=3. The lower i bits of A^i are sorted in the ascending order by the Ω_N^{-1}-control algorithm. (→ indicates the control bit.)

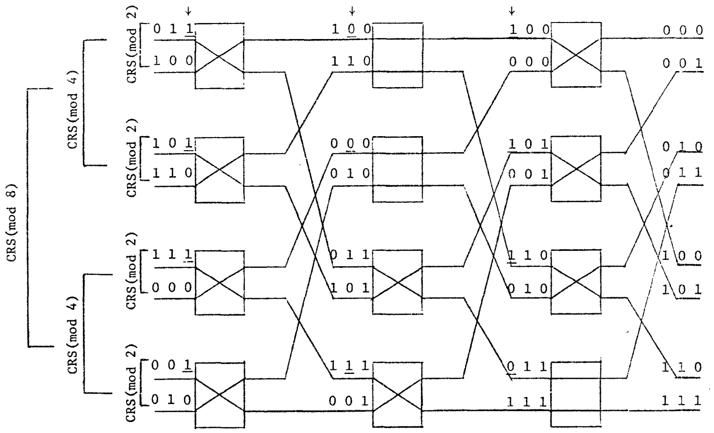


Fig. 7 . An Example of Theorem 1 for n=8. An uniform shift of distance 3, (3,4,5,6,7,0,1,2), is Ω_8^{-1}-passable. (Control bits are underlined.)

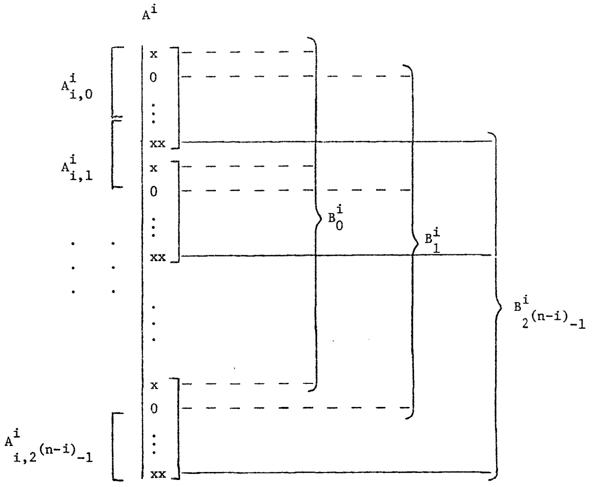


Fig. 8 . Two different ways of a $2^{(n-i)}$-partitioning of A^i.

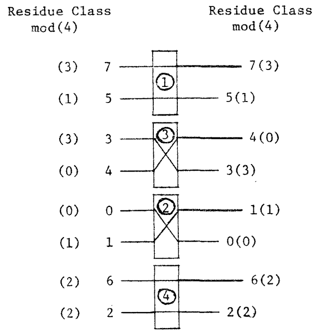


Fig. 9 . An Example of Lemma 4: a CRP can be always realized on a set of switches. (Circled numbers show the order of the switch settings. Input CRS(mod 8) = {7,5,3,4,0,1,6,2} is partitioned to two CRS(mod 4), {7,4,1,6} and {5,3,0,2}.)

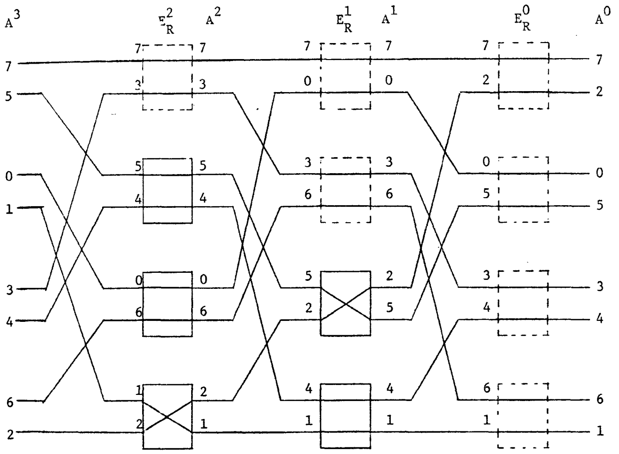


Fig. 10. An Example of Theorem 2: an arbitrary permutation can be transformed to meet the Ω_N^{-1}-passable condition via the reduced Ω_N. $A^3 = (7,5,0,1,3,4,6,2)$ is not Ω_8^{-1}-passable but $A^0 = (7,2,0,5,3,4,6,1)$ is. (Redundant switches are drawn in dotted lines.)

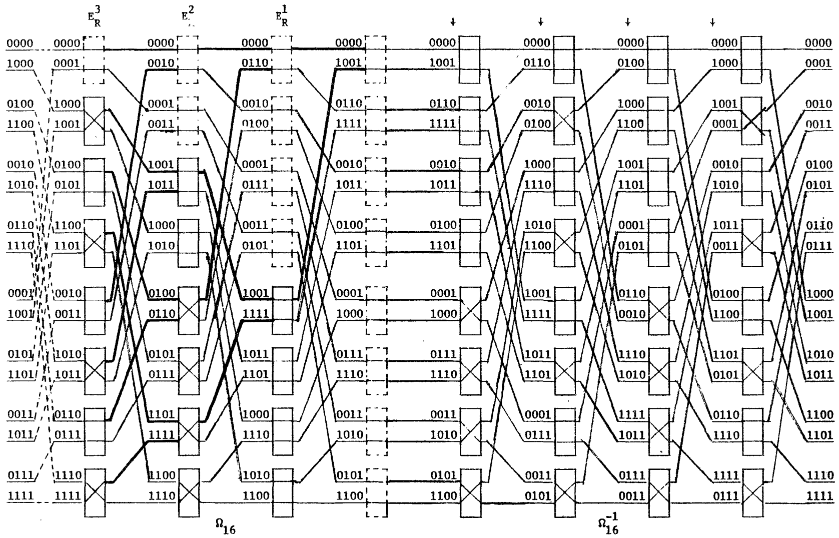


Fig. 11. Bit Reversal on $\Omega_{16}\Omega_{16}^{-1}$. The reduced $\Omega_{16}\Omega_{16}^{-1}$ is drawn in solid lines. (→ indicates control bits and heavy lines show a CRS partitioning.)

PERFORMANCE AND IMPLEMENTATION OF 4x4 SWITCHING NODES IN AN INTERCONNECTION NETWORK FOR PASM

Robert J. McMillen, George B. Adams III, and Howard Jay Siegel
School of Electrical Engineering, Purdue University
West Lafayette, IN 47907

Abstract

Design issues for the multistage Generalized Cube network are discussed in this paper. An analysis of the merits of 2-input/2-output interchange boxes versus 4-input/4-output crossbars for interconnection network implementation is made. The cost and performance of each network for the two switching node alternatives are examined. Discussion of the suitability of each approach for VLSI implementation is included. It is shown that in a packet switching environment, 4x4 crossbars outperform, and are less expensive to implement than the four interchange boxes they replace.

I. INTRODUCTION

The choice of interconnection network is a central issue in the design of large-scale, multimicroprocessor-based distributed and parallel systems. The Ballistic Missile Defense (BMD) Agency is designing a test bed for evaluating such systems as they may apply to BMD tasks [8]. PASM is a multimicroprocessor system being designed at Purdue University for a variety of image processing and pattern recognition problems [16]. In both cases a highly flexible network is needed for communication among processors and memories.

The Generalized Cube network has a cube-type topology and is constructed from 2-input/2-output crossbars or interchange boxes [17]. A more general form of interchange box is an a-input/a-output (a x a) switching node. A relative of the Generalized Cube network can be constructed from a x a switching nodes using cube-type connections between stages. Many papers in the literature discuss using larger than 2x2 interchange boxes for implementing multistage cube-type networks [2, 7, 10, 11, 12, 18]. In the following, design options for 4x4 switching nodes are considered. The performances of two designs are evaluated and their implementation in discrete logic (e.g., TTL) and VLSI is considered. It will be shown that a 4x4 crossbar performs better and costs less than four 2x2 crossbars in a packet switching environment.

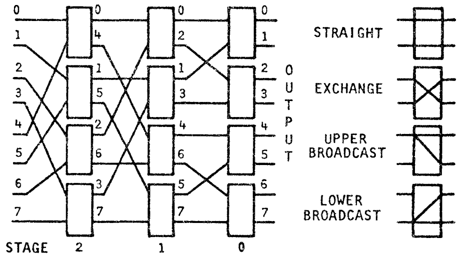
The logical structure of the Generalized Cube network is defined in Section II to provide a framework for discussing modifications. In Section III, the performance of two network implementations are compared. Implementation considerations are presented in Section IV. For further details of all this material see [14].

II. DEFINITIONS

A partitionable SIMD/MIMD system is a parallel processing system which can be structured as one or more independent SIMD and/or MIMD machines [4] of varying sizes. PASM is a partitionable SIMD/MIMD system for image processing and pattern recognition [16]. The BMD testbed should have the flexibility to perform as a partitionable SIMD/MIMD machine. The cube network described here can function efficiently in such an environment.

The Generalized Cube network (Fig. 1) is a multistage cube-type network topology which was introduced in [17]. It has been shown that this topology is equivalent to that used by the omega [7], indirect binary n-cube [11], STARAN [1], and SW-banyan (F=S=2) [6] networks [17, 20]. An N input/output Generalized Cube topology has $m = \log_2 N$ stages, where each stage consists of a set of N lines connected to N/2 interchange boxes. Each interchange box is a 2-input/2-output device. The labels of the input/output (I/O) lines entering the upper and lower inputs of an interchange box are used as the labels for the upper and lower outputs, respectively. The labels are the integers from 0 to N-1. Each interchange box can be set to one of four states as shown in Fig. 1. The connections in this network are based on the cube interconnection functions [13]. Stage i of the generalized cube topology pairs I/O lines that differ only in the i-th bit position.

The name cube network will be used to refer to the network consisting of the Generalized Cube topology and four-state interchange boxes. Each interchange box will be controlled independently through the use of routing tags [7, 15].



(a) (b)

Figure 1(a): Generalized Cube topology for N=8.
(b): Four states of an interchange box.

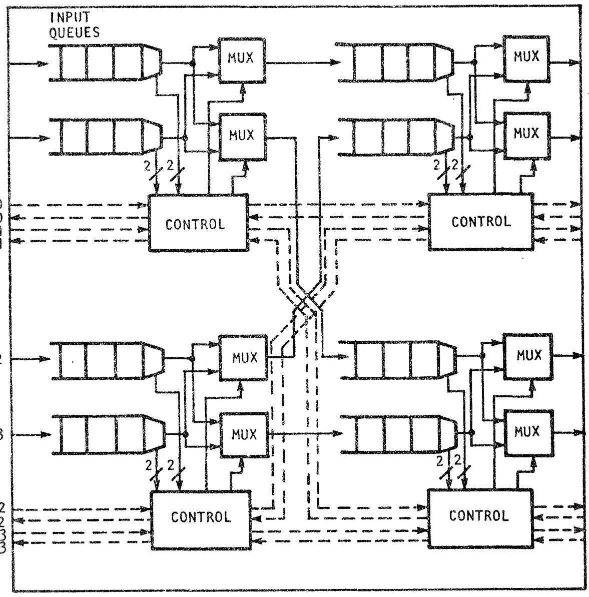
This work was supported by the Ballistic Missile Defense Agency under grant number DASG60-80-C-0022 and the Air Force Office of Scientific Research, Air Force Systems Command, USAF, under AFOSR-78-3581. The United States Government is authorized to reproduce and distribute reprints for Government purposes nonwithstanding any copyright notation hereon. The views, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other official documentation.

0190-3918/81/0000/0229$00.75 © 1981 IEEE

It is assumed that processors and memories are paired to form *processing elements (PE's)*. The network is configured such that PE i is connected to input i and output i, $0 \leq i < N$. The *packet switching* mode, in which packets move from stage to stage in the network as paths between stages become available, is assumed. They do not require that their entire path be established prior to entering the network. A *packet* consists of a routing tag and a number of data items. Packet switching in multistage networks has been discussed in [3, 19].

The primary goal here is to investigate the cost-effectiveness of constructing multistage cube networks from 4x4 crossbars versus 2x2 crossbars (interchange boxes). Since a single 2x2 interchange box is not functionally comparable to a 4x4 crossbar (i.e., it can only handle two items at a time instead of four), the 4x4 crossbar is compared with a 4x4 composition of four 2x2 interchange boxes. This configuration is called a *composite node* and is shown in Fig. 2. A network constructed from properly connected (to be specified later) composite nodes is identical to a cube network constructed from 2x2 interchange boxes. The external connections of the crossbar (Fig. 3) are identical to those of the composite node, so it can be directly substituted for a 4x4 composite node.

Many options for the implementation of 2x2 interchange boxes were discussed in [9]. To avoid repetition, one of the configurations discussed in that paper will be assumed here. It is assumed that packet switching is implemented and that an entire packet is transferred between adjacent stages during one network clock cycle. Furthermore, the size of each input queue in a switching node is assumed to be an integral multiple of the packet size. The packet size is thus not restricted to be any particular number of words.



R-REQUEST G-GRANT

Figure 2: A 4x4 composite node constructed from four 2x2 interchange boxes.

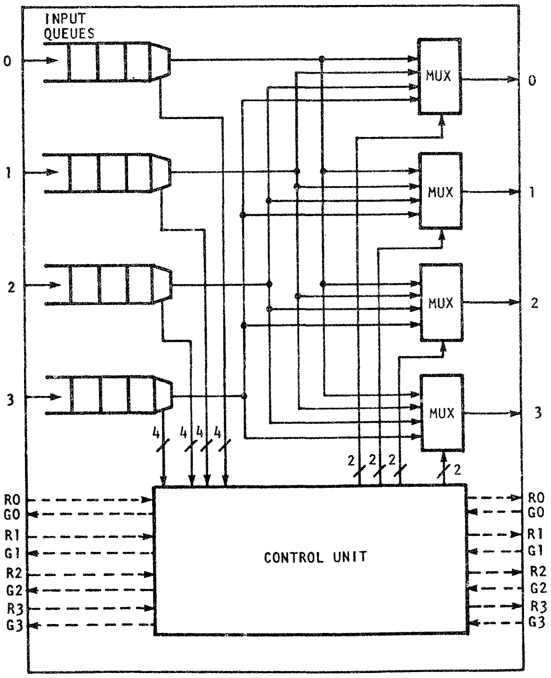
III. PERFORMANCE ANALYSIS

The 4x4 crossbar node and composite node will be compared in their performance at both a local and global level. On a local level blocking within a node is examined. On the global level, the permuting ability of two networks constructed from the respective 4x4 switching nodes is compared.

Consider the local level. Let *level 1* of a composite node be the two interchange boxes connected to the inputs of the node and *level 2* be the two interchange boxes connected to the outputs. The composite node can perform 16 permutation connections (each box either straight or exchange) and the crossbar node can perform all 4! possible permutation connections.

For those permutations where there is no conflict in either node, the messages traverse the composite node in twice the time required by those in the crossbar node due to the two levels of interchange boxes. When conflicts occur in the crossbar node, the delay due to waiting diminishes the speedup achieved.

Consider situations where there are conflicts in a switch node. For this analysis it is assumed that the destination of any message is a uniformly distributed random variable. Also, it is assumed that each message has only one destination (i.e., no broadcasting). Both the composite node and the 4x4 crossbar node have four inputs and four outputs so there are $4^4=256$ distinct patterns in which messages may need to be routed through the boxes. Since the destinations are assumed to be random and uniformly distributed, the distinct data patterns of routing are all equally likely. Assuming four simultaneous inputs is somewhat of a worst case, since in MIMD mode this would be con-



R-REQUEST G-GRANT

Figure 3: A 4x4 crossbar node.

sidered heavy loading and in SIMD mode destinations are not random but structured and chosen to avoid conflicts. The node is assumed initially empty.

Consider the 4x4 crossbar node. Let r be the maximum number of messages desiring any given output of the 4x4 crossbar node. The total time required for all four messages to pass through the node is r. $P(r=1) = 24/256$, $P(r=2) = 180/256$, $P(r=3) = 48/256$, and $P(r=4) = 4/256$. The expected time to pass all four messages through the crossbar node is given by:

$$\sum_{i=1}^{4} i \cdot P(r=i) = 2.125 \text{ network clock cycles.}$$

That is, given that four messages arrive at an empty crossbar node simultaneously, on the average it will take 2.125 network clock cycles for the node to empty.

Now consider the composite node. The following notation will be used in the ensuing equations, where i=1 or 2:

$P(iU) = P(\text{no conflict level } i\text{, upper box}) = 1/2;$
$P(iL) = P(\text{no conflict level } i\text{, lower box}) = 1/2;$
$P(iX) = 1/2$, where X = U or L; and
$P(i) = P(\text{no conflict in level } i) = 1/4.$

Now consider the probabilities of different amounts of time, t, to pass four input messages through the composite node. The minimum time possible is 2 network clock cycles because there are two levels.

$P(t=2) = P(1U) \cdot P(1L) \cdot P(2U) \cdot P(2L) = 1/16.$

For a total time of 3 network clock cycles there are 5 cases to consider. First assume no conflicts occur in level 1.

$P(t=3\text{, case } 1) = P(1) \cdot (1-P(2)) = 3/16.$

Next, assume exactly one level 1 interchange box has a conflict. $P(t=3\text{, case } 2) = [(1-P(1U)) \cdot P(1L)+P(1U) \cdot (1-P(1L))] \cdot P(2X) = 1/4.$

For case 3, there is one conflict at each level, but the maximum delay is 3 cycles.

$P(t=3\text{, case } 3) = [(1-P(1U)) \cdot P(1L)+P(1U) \cdot (1-P(1L))] \cdot (1-P(2X)) \cdot (1/2) \cdot P(2X) = 1/16.$

The first factor is the probability that exactly one box at level 1 has a conflict. The next factor is the probability that the first message from the level 1 box which had a conflict, call this message M, also has a conflict at level 2. The (1/2) is the probability that M will be chosen to pass through the level 2 box first. The last factor is the probability that the two delayed messages do not conflict.

Case 4 assumes that there is a conflict in both level 1 boxes and that both level 2 boxes receive messages (this happens half the time there are two conflicts in level 1).

$P(t=3\text{, case } 4) = (1/2) \cdot (1-P(1U)) \cdot (1-P(1L)) = 1/8.$

Finally, assume conflict in both level 1 boxes but only one level 2 box receives messages and there is no conflict for either pair that passes through: $P(t=3)\text{, case } 5) = (1/2) \cdot (1-P(1U)) \cdot (1-P(1L)) \cdot P(2X) \cdot P(2X) = 1/32.$

The probability that all messages pass through the composite node in 3 network clock cycles is

$P(t=3) = 3/16 + 1/4 + 1/16 + 1/8 + 1/32 = 21/32.$

For a time of 4, there are four cases to consider. The first case is where there is one conflict at each level. There are two ways to obtain a time of 4 from this situation: (1) the delayed message enters a non-empty queue in level 2 and (2) the delayed message enters an empty queue but conflicts with the other remaining message:

$P(t=4\text{, case } 1) = [(1-P(1U)) \cdot P(1L)+P(1U) \cdot (1-P(1L))] \cdot [(1/2) \cdot (1-P(2X))+(1/2) \cdot (1-P(2X)) \cdot (1-P(2X))]=3/16.$

Now assume conflict in both level 1 boxes and that only one level 2 box receives messages (this happens half the time there are two conflicts in level 1). Given this occurs, there are three ways (cases 2, 3, and 4) a time of 4 occurs. In case 2, the first two messages reaching the box in level 2 conflict, but there are no subsequent conflicts:

$P(t=4\text{, case } 2) = (1/2) \cdot (1-P(1U)) \cdot (1-P(1L)) \cdot (1-P(2X)) \cdot P(2X) = 1/32.$

In case 3, the first pair of messages do not conflict but the second pair do:

$P(t=4\text{, case } 3) = (1/2) \cdot (1-P(1U)) \cdot (1-P(1L)) \cdot P(2X) \cdot (1-P(2X)) = 1/32.$

In case 4, the first and second pair of messages conflict. When the second pair conflicts, one queue will contain two messages. For a time of 4 the queue with two items must be selected to resolve the second conflict and a third conflict must not occur.

$P(t=4\text{,case } 4) = (1/2) \cdot (1-P(1U)) \cdot (1-P(1L)) \cdot (1-P(2X)) \cdot (1-P(2X)) \cdot (1/2) \cdot P(2X) = 1/128.$

The probability of a time of 4 is:

$P(t=4) = 3/16 + 1/32 + 1/32 + 1/128 = 33/128.$

The time of 5 happens when either of the two conditions of case 4 for a time of 4 are not met.

$P(t=5) = (1/2) \cdot (1-P(1U)) \cdot (1-P(1L)) \cdot (1-P(2X)) \cdot [(1/2)(1-P(2X))+(1/2)(1-P(2X))(1-P(2X))] = 3/128.$

The expected time for all four messages to pass through the composite node is:

$$\sum_{i=2}^{5} i \cdot P(t=i) = 3.242 \text{ network clock cycles.}$$

This time is 53% longer than the 2.125 network clock cycles expected with the crossbar node.

Consider the global level. To construct a network from m/2 stages of N/4 4x4 switching nodes, assume all connection lines in the network are labeled in base 4 and that the stages are numbered (m/2)-1,•••,1,0 (from input to output). At stage i, the four input lines to a node are those that differ only in the i-th position of their base 4 representation. The line with a 0 in the i-th position connects to the top input, 2 to the next input, 1 to the next input, and 3 to the bottom input. The output lines of the 4x4 switching nodes have the same labels as the input lines, but in increasing order, i.e., the top output line label has a 0 in the i-th position, next 1, next 2, and the bottom 3. When composite nodes are used, making connections in the above manner creates a cube network. When crossbars nodes are used, a network is created whose capabilities are a superset of those of the cube network.

A composite node network consists of Nm/2 interchange boxes, allowing $2^{Nm/2}$ permutations. Assuming m is even, a 4x4 crossbar node network consists of Nm/8 nodes, permitting $(4!)^{Nm/8}$ permutations. If m is odd and one stage is constructed by 4x4 crossbar nodes limited to act as a 2x2 nodes, then $2^{N/2}(4!)^{N(m-1)/8}$ permutations are possible.

IV. IMPLEMENTATION

To control the network, the destination tags defined in [7] are used. Let the destination address D be represented in binary as $d_{m-1} \cdots d_1 d_0$. A switching node in stage i examines bits d_{2i+1} and d_{2i}. For the composite node, the first level interchange boxes examine only bit d_{2i+1} and the second level interchange boxes examine only bit d_{2i}. If the bit examined is 0, the upper output link of the interchange box is selected and if the bit is 1, the lower link is selected. For the crossbar node, both bits are examined simultaneously. Together they are considered a single base four digit which corresponds to one of the outputs labeled 0 through 3.

To add a broadcast capability, an m-bit broadcast mask is appended [15]. Let the mask B be represented in binary as $b_{m-1} \cdots b_1 b_0$. A switching node in stage i now examines b_{2i+1}, b_{2i}, d_{2i+1} and d_{2i}. For the composite node, first level interchange boxes examine bits with index 2i+1 and second level boxes examine bits with index 2i. If the broadcast mask bit is 0, the destination tag bit is interpreted as before. If the mask bit is 1, the destination bit is ignored and both output links of the interchange box are selected. For the crossbar node the four bits are all examined simultaneously. They are interpreted so as to establish the same connections as those that would be obtained in the composite node. Five kinds of broadcasts are defined for either type of 4x4 switching node.

Hardware Without Broadcast Capability

For simplicity, designs for the composite node and the crossbar node initially will be developed assuming no broadcast capability. Then, those portions of the designs affected by inclusion of a broadcast capability will be modified and compared.

In the following analysis, hardware complexity is measured in terms of logic gate count and chip count. The gate counts are used as a first approximation to compare VLSI implementations. Designs using this technology must also consider wiring complexity [5]. The chip counts are used to compare discrete logic (e.g. TTL) implementations, assuming standard gate-per-chip packaging.

Examining Figs. 2 and 3, the first difference noted is that the crossbar node requires half as many queues as the composite node. Depending on the actual queue size, a considerable savings in logic may be realized in the implementation of the crossbar node. To compare multiplexer requirements, typical implementations of 2-to-1 and 4-to-1 multiplexers were examined [14]. Eight 2-to-1 multiplexers require 20% more gates (regardless of path width) than four 4-to-1 multiplexers. The chip counts are equal. Since the number of external connections for data and control lines is the same for both designs, any buffering/signal conditioning logic will be comparable. In a VLSI design, this implies identical pin counts.

Thus far the crossbar node appears to be the better choice. It is however, decidedly more complicated to arbitrate the requests of four packets simultaneously (as opposed to two) while assuring each packet equal access to each output link on the average. To determine whether one 4x4 control unit is actually more complex than four 2x2 control units, the functional components of the control units are considered.

The control unit of a 2x2 interchange box contains two sets of queue control logic, input request arbitration (IRA) logic, output request arbitration (ORA) logic, and timing. The control unit for a 4x4 crossbar node contains four sets of queue control logic. The remaining components are the functional equivalents of those for the 2x2 interchange box. The most obvious difference between the two designs is that four 2x2 control units contain twice as many sets of queue control logic as one 4x4 control unit.

One set of queue control logic contains two registers which store pointers, one to the front and one to the back of its associated queue. If the queue is Q words long, $\log_2 Q$ bits are required for each register.

The IRA logic is quite simple. If a request is made for the i-th input, (i=0,1 for the 2x2; i=0,1,2,3 for the 4x4), it will be granted if the i-th queue is not full. Once again, four 2x2 control units require twice as much IRA logic as one 4x4 control unit.

The timing logic is identical in both cases. Three clock phases are generated. A request/grant/transfer protocol is implemented (see [9]).

None of the logic discussed thus far is affected by the inclusion of a broadcast capability. Thus, its analysis is equally applicable to the next subsection, which includes broadcast capabilities.

The most important and by far the most complex component of the control unit is the ORA logic. It is responsible for examining the routing tag bits and generating signals to set the multiplexers and make requests. It must also examine the grant signals and generate control signals for the "increment front pointer" input of each set of queue control logic. The complexity of this logic arises from arbitrating conflicting requests for access to the output ports.

To compare the ORA logic, equations are derived for all its output signals as a function of the tag bits and grant signals [14]. The total (NAND) gate count for 4 sets 2x2 of control unit logic is 104 gates. This corresponds to 24 chips. The control unit for the 4x4 crossbar node requires 124 gates. There is a 19% increase in the number of gates required by the crossbar node. In a discrete logic design, the chip count is 32. This is a 33% increase over the 24 chips required in the composite node.

The excess in ORA logic can be compensated for, since a 4x4 crossbar node requires half the queue control and IRA logic of a 4x4 composite node. From the equations derived, 20 extra gates or eight extra chips are required for the 4x4 crossbar ORA logic. Assuming one of the eight sets of queue control and IRA logic in a composite node will require more than 5 gates or 2 chips, the 4x4 crossbar node is actually less expensive to build. Despite the higher wiring complexity of the 4x4 crossbar node, the total design effort is comparable to that required by the 4x4 composite node.

Hardware With Broadcast Capability

Adding a broadcast capability requires the ORA logic to examine the broadcast mask bits in addition to the routing tag bits. The revised equations for the 2x2 control unit require 33 gates, which multiplied by 4 is 156. This is equivalent to 48 chips. A broadcasting capability costs 52 gates or 24 chips beyond the requirements for a 4x4 composite node without it. More details can be found in [14].

The circuitry needed to add the same broadcast capability to 4x4 crossbar nodes as was added to the composite nodes requires 233 gates, a 49% increase over the 156 required for the composite node. The chip count is 74, a 54% increase over 48. In this case it is likely that one of the eight sets of queue control and IRA logic will require more than 20 gates or 7 chips. If not, the savings in queue gates will compensate for the difference. Again the crossbar node is less expensive than a composite node where both have the same broadcast capability.

V. CONCLUSIONS

At a local level, the crossbar node is always faster at passing four messages that arrive simultaneously than the composite node. If the connection requests do not conflict in the composite, the crossbar is twice as fast. When the connection requests of the messages form a permutation which the composite node cannot pass without conflict, it takes 3 times longer for all messages to exit the composite node. Assuming each message chooses each output with equal probability, on the average it takes approximately 53% more time for all messages to pass through the composite node than through the crossbar node.

The ORA logic is the only logic requiring more hardware in a crossbar node than in a composite node. Otherwise, a crossbar node requires half as much queue control and IRA logic, and half as many queues. The multiplexer logic is less than or comparable to that needed by the composite node. The net result is that when packet switching is implemented, the 4x4 crossbar node requires less hardware and significantly out-performs a composite node.

If circuit switching is implemented, no queues or their associated control logic are required. In this case, the crossbar node does contain more hardware. However, it offers a significant improvement in connectivity/permuting ability. If the switching nodes are implemented as VLSI chips, since both nodes require the same number of pins, the gate/pin ratio is improved with a crossbar implementation. Only in the case where circuit switching is implemented in discrete logic is further consideration required. Without a broadcast capability (which is less important in a circuit switching environment), there is only a small difference in the chip count.

In summary, the implementation of cube-type networks using 2x2 and 4x4 crossbars were compared. It was shown that for packet switching the 4x4 crossbar is a more cost-effective approach.

REFERENCES

1 K. Batcher, "The flip network in STARAN," 1976 Int. Conf. Parallel Processing, pp. 65-71, Aug. 1976.

2 L. Ciminiera, A. Serra, "Modular interconnection networks with asynchronous control," 14th Hawaii Int. Conf. System Sciences, pp. 210-218, Jan. 1981.

3 D. Dias, J. Jump, "Packet communication in multistage shuffle-exchange networks," 1980 Int. Conf. Parallel Processing, pp. 327-328, Aug. 1980.

4 M. Flynn, "Very high-speed computing systems," Proc. IEEE, Vol. 54, pp. 1901-1909, Dec. 1966.

5 M. Franklin, "VLSI performance comparison of banyan and crossbar communications networks," Workshop on Interconnection Networks for Parallel and Distributed Processing, pp. 20-18. Apr. 1980.

6 G. Goke, G. J. Lipovski, "Banyan networks for partitioning multiprocessor systems," 1st Symp. Comp. Arch., pp. 21-28, Dec. 1973.

7 D. Lawrie, "Access and alignment of data in an array processor," IEEE Trans. Comp., Vol. C-24, pp. 1145-1155, Dec. 1975.

8 W. McDonald, J. Williams, "The advanced data processing test bed," Compsac, pp. 346-351, Mar. 1978.

9 R. J. McMillen, H. J. Siegel, "The hybrid cube network," Distributed Data Acquisition, Computing and Control Symp., pp. 11-22, Dec., 1980.

10 J. Patel, "Processor-memory interconnections for multiprocessors," 6th Symp. Comp. Arch., pp. 168-177, Apr. 1979.

11 M. Pease, "The indirect binary n-cube microprocessor array," IEEE Trans. Comp., Vol. C-26, pp. 458-473, May 1977.

12 U. Premkumar, et al., "Design and implementation of the banyan interconnection network in TRAC," NCC, pp. 643-653, June 1980.

13 H. J. Siegel, "A model of SIMD machines and a comparison of various interconnection networks," IEEE Trans. Comp., Vol. C-28, pp. 907-917, Dec. 1979.

14 H. J. Siegel, et al., Parallel/Distributed Multimicroprocessor Systems for Ballistic Missile Defense, Purdue, EE School, TR-EE 81-12, June 1981.

15 H. J. Siegel, R. J. McMillen, "The cube network as a distributed processing test bed switch," 2nd Int. Conf. Distributed Computing Systems, pp. 377-387, Apr. 1981.

16 H. J. Siegel, et al., "PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition," IEEE Trans. Comp., to appear.

17 H. J. Siegel, S. D. Smith, "Study of multistage SIMD interconnection networks," 5th Symp. Comp. Arch., pp. 223-229, Apr. 1978.

18 S. D. Smith, "LSI design considerations for multistage interconnection networks for parallel processing systems," 14th Hawaii Int. Conf. System Sciences, pp. 219-227, Jan. 1981.

19 A. Tripathi, G. J. Lipovski, "Packet switching banyan networks," 6th Symp. Comp. Arch., pp. 160-167, Apr. 1979.

20 C. Wu, T. Feng, "On a class of multistage interconnection networks," IEEE Trans. Comp., Vol. C-29, pp. 694-702, Aug. 1980.

ON NON-EQUIVALENT MULTISTAGE INTERCONNECTION NETWORKS

Dharma P. Agrawal and Sung-Chun Kim

Electrical and Computer Engineering
Wayne State University
Detroit, Michigan 48202

Abstract

This paper presents a systematic way of finding whether Multistage Interconnection (MI) Networks with $\log_2 N$ stages and implemented with 2x2 Switching Elements (SEs) are non-equivalent or not. The basic strategy is to employ a graph theoretic approach to model the MI networks in the form of the directed graph and use non-isomorphic properties of the loops or cycles in corresponding undirected graphs. The distance concept of a binary tree is utilized to ensure the full connectivity requirements of the MI networks.

I. Introduction

Various MI networks described in the literature [7] are topologically equivalent to each other [1-3,7] as the SEs of the networks are connected in such a way that they possess a "buddy" property [4]. This means that the outputs of two SEs at stage i are connected as inputs to only two SEs at the (i+1)th stage and thereby a uniformity is provided in the network. Patel [5] has stated that probably there are only two non-equivalent 8x8 networks and named these as delta networks. For finding non-equivalent networks, we considered several alternatives and the use of graph theory looked to be more promising.

II. Graph Model of the MI Network

We are concerned with N-input N-output network ($N=2^n$) utilizing 2x2 SE and having n stages connected in such a way that it provides full connectivity, i.e., any input terminals can be connected to any one of the output terminals. Fig. 1-a shows the SE and its two possible states and Fig. 1-b shows the proposed graph model of Fig. 1-a with the directions indicating flow of data. Note that the control line is totally eliminated in graph representation of the SE. Note that contrary to the topology describing rules [7], the SE with link patterns as shown in Fig. 1-c is modeled and treated just like the one shown in Fig. 1-b. The MI network can now be modeled as a directed graph by assigning nodes to each SE and input/output terminals, and providing connection links between nodes of various stages. This leads to the graph model of the Omega Network of Fig. 2-a as shown in Fig. 2-b.

It is also worth noting that the part of the graph (or subgraph) representing the connections between two stages are bipartite [6] in nature and the input/output terminals of Fig. 2-b can be excluded without losing any topology information thereby making the analysis much simpler. The reduced graph is shown in Fig. 2-c. Separating each N-node bipartite subgraphs leads to exactly (n-1) such subgraphs and such subgraphs are shown in Fig. 2-d.

III. Single Level Partitioning of Stages

With the proposed graph model, the non-equivalence between possible MI networks is transformed to non-isomorphism of bipartite subgraphs.

Theorem 1: There exist at least one closed loop (cycle) in a bipartite subgraph of the MI network.

Proof. From Section II each bipartite subgraph consists of N/2 nodes in each set with N edges, each node of one set is connected to two nodes of another set, and there has to be at least one undirected closed path. Q.E.D.

Corollary 1: All the paths in the bipartite graph form the closed loops.

Proof. This is obvious from the characteristics of bipartite graph with each node of degree 2 and total number of edges is the same as number of nodes. Q.E.D.

Corollary 2: The number of paths, j, constituting a loop is determined by the relation $j=4+2\cdot i$, where i = 0,1,2,...,(N/2-4), (N/2-2) and i gives the integer value satisfying corollary 1 and the loop will be defined as jP-L.

Proof. Follows from Corollary 1. Q.E.D.

For instance, the bipartite subgraph of 2^3 inputs MI network has only two possible kinds of loop structure and these 4P-L and 8P-L are shown in Fig. 3-a. For the case of 2^4 inputs/outputs network it is possible to make i equal to 0,1,2, 3,4,6 and these loops correspond to 4P-L, 6P-L, 8P-L, 10P-L, 12P-L, and 16 P-L as shown in Fig. 3-b.

Theorem 2: Among all possible bipartite subgraphs of the MI networks, the subgraphs are not isomorphic to each other iff the number and type of loops are not exactly the same.

Proof. Obviously, the loops consisting of different number of paths are non-isomorphic due to their own unique structure. If the number and type of loops are exactly the same in two bipartite subgraphs, then they can be shown to be isomorphic by changing their positions to correspond to each other. Q.E.D.

It may be noted that all the MI networks having same number of and type of loops may not be isomorphic and will be considered in Theorem 6 of section V.

Example 1: Two different loops for the bipartite subgraphs of 8 x 8 network result into two different topologies of Fig. 3-a (Theorem 2) [2(4P-L) means two 4P-L].

Example 2: For the case of 16 x 16 network, six different kinds of loops are possible and we can obtain 7 different topologies as illustrated in

0190-3918/81/0000/0234$00.75 © 1981 IEEE

Fig. 3-b.

IV. Isomorphism of MI Network

Theorem 3: Reordering the positions of the nodes in the same stage of the MI network does not affect the loop structure within each bipartite subgraph.
Proof. It is obvious from the graph theory [6] that reordering of a graph will provide another graph which will be isomorphic to the first graph. Hence, loop structure remains the same. Q.E.D.
Theorem 4: All possible combinations of the graphs resulting from grouping and joining of two or more non-isomorphic bipartite subgraphs are also non-isomorphic.
Proof. A network with one specific topology created by combination of some bipartite subgraphs provides its own unique combined loop structure and according to Theorem 3, a graph guarantees the uniqueness of the loops. Q.E.D.

V. Loop Analysis

Let us modify the bipartite graphs of Fig. 2-d in the form of planner graphs as shown in Fig. 4-a while its loop structure is maintained. The next step is to connect the nodes of two adjacent levels. This is done on the assumption that output nodes of the loops at the first level are at the input nodes at the second level. Hence, if we assume 4 and 4' etc. are not separate but the same nodes then we obtain the graph of Fig. 2-c back. These overlappings of nodes are indicated by a simple line connecting nodes 4 and 4'. Following the same procedure, it can be shown that 8x8 Baseline Network [7] will have the same loop diagrams as shown in Fig. 4-b and only difference will be numbering of nodes.

This synthesis procedure could be used to define the composite graph of all possible MI networks which are not isomorphic to each other. This could be done starting from the non-isomorphic loop structures and interpolate the unnumbered nodes of various stages with restrictions to satisfy full connectivity requirements.

Connecting the Squared Loops

A. Draw the loop structures and mark alternate nodes for inputs by '•' and the rest for the outputs by 'x'. The output nodes of the first level are to be overlapped or merged with the input nodes of the next level and this process is to be performed for all levels in such a way that full connectivity is provided by the resulting network (Fig. 4-b). The basic requirement is to form a binary tree from each input node [3] and hence in Fig. 4-b the output nodes of level 1 are connected to input nodes of level 2 in such a way that the minimum distance between input nodes in loop of level 2 to be at least 2 times the distance between the corresponding output nodes of level 1. If we have many levels, then the minimum distance to get the full connectivity has to be $4x(2^{\ell-1}-1)$, where ℓ is the level number of the loops. The distance between different loops of the same level is regarded as infinite.

Theorem 5: The above design procedure provides full connectivity in the resultant MI network.
Proof. The proof is obvious from the tree formation procedure [3]. Q.E.D.

B. If each level consists of more than 2 loops, then different topology of connecting schemes for a given set of loops might exist. For instance, from Section IV, 3 different topologies are possbile for 8x8 network. But according to Theorem 5, we see that only two topologies of [2(4P-L); 2(4P-L)] and [2(4P-L);8P-L] provide full connectivity (Figs. 4-b and 4-c) and the third [8P-L;8P-L] of Fig. 4-e fails to satisfy the distance requirements.

Theorem 6: For a given specific structure of the loops, it is possible that different alternative connections of loops might provide more than one non-isomorphic graph and hence non-equivalent networks.
Proof. The proof of Theorem 6 will be given using an example to show that there are many possible ways of connecting the loops at two adjacent levels. Take an example of Fig. 5-a which consists of four 4P-L in every level. To see the ways of connecting scheme explicitly, we represent each (4P-L) loop by a Macro node as shown in Fig. 5-a. This modified structure has already been covered in Fig. 4 and the two non-isomorphic schemes are shown in Fig. 5-b, leading to two different connections as in Fig. 5-c. The actual graphs resulting from the loop diagrams of Fig. 5-c showing two different topologies are illustrated in Fig. 5-d. Q.E.D.

VI. Construction Algorithm of the MI Networks

a. For each level, select the loops with each loop consisting of j nodes and edges where $j = 4 + 2 \cdot i$ for $i = 0,1, 2,\ldots,(N/2-4), (N/2-2)$.

b. In each loop, assign alternate nodes as for inputs and outputs.

c. Connect adjacent levels of loop diagram such that they satisfy Theorem 5 and assign numbers to all the nodes.

d. Obtain the graph model of the MI network by merging each adjacent levels of the loop diagram.

e. Convert it to the normal diagram showing 2x2 SEs. Choose any model of the SE from Figs. 1-a or 1-c to be used.

VII. Concluding Remarks

In this paper, we presented a systematic methodology for determining non-equivalence of MI networks and designing such networks. It is our firm belief that the novel approach introduced in this paper provides a better perspective of the non-equivalent MI networks.

References

[1] M.A. Abidi and D.P. Agrawal, "On Conflict-free Permutations in Multistage Interconnection Networks," Journal of Digital Systems, Vol. V, No. 2, Summer 1980, pp. 115-134.

[2] M.A. Abidi and D.P. Agrawal, "Two Single Pass Permutation in Multistage Interconnection Networks," Proc. 1980 Conf. Infor. Sci. and Sys., March 26-28, 1980, pp. 516-522.

[3] D.P. Agrawal, "Graph Theoretic Analysis and Design of Multistage Interconnection Networks," communicated for publication.

[4] D.M. Dias and J.R. Jump, "Analysis and Simulation of Buffered Delta Networks," IEEE Trans. Comp., April 1981, pp. 273-282.

[5] J.H. Patel, "Processor-Memory Interconnections for Multiprocessor," Proc. 6th Ann. Com. Arch. Conf., April 1979, pp. 168-177.

[6] S. Sahni, "Mathematical Concepts in Science and Engineering," Report, University of Minnesota, Sept. 1980.

[7] C.L. Wu and T. Feng, "On a Class of Multistage Interconnection Networks," IEEE Trans. Comp., Aug. 1980, pp. 694-702.

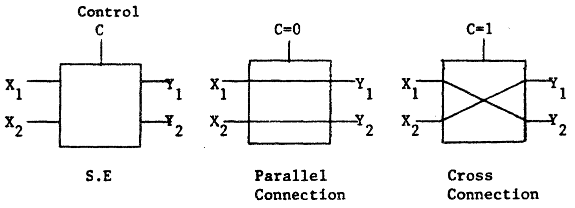


Fig. 1-a Switching Element and its two possible states.

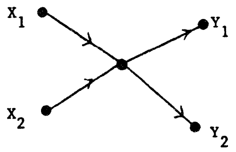


Fig. 1-b Graph model of S.E.

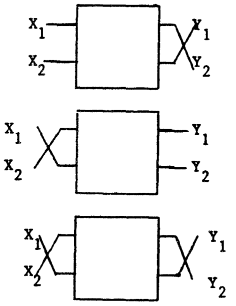


Fig. 1-c Alternate Positioning of input/output link.

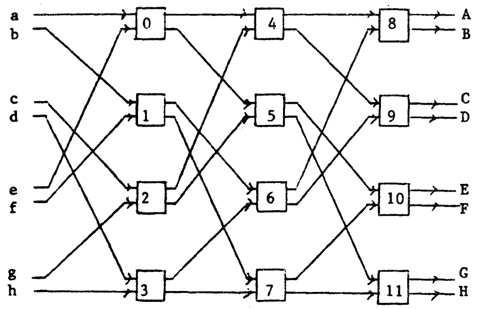
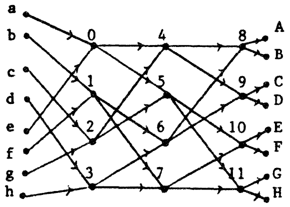


Fig. 2-a 8 x 8 Omega network : Ω_8.



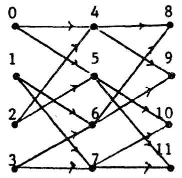


Fig. 2-b Graph model of Omega network. Fig. 2-c Simplified Omega network.

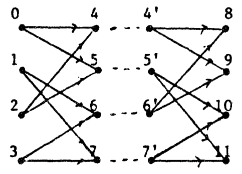


Fig. 2-d Two single bipartite graphs.

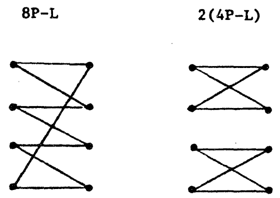


Fig. 3-a Two kinds of bipartite graph of 8 x 8 M I network.

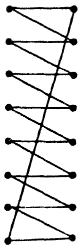
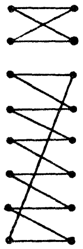
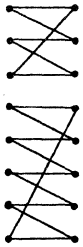
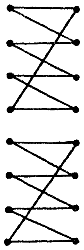
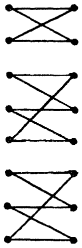
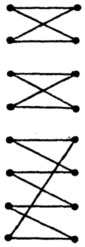
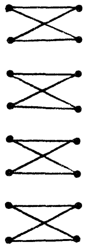
Number of loops in a bipartite graph	1 Loop	2 Loops			3 Loops		4 Loops
Structure of single bipartite graph	 16P-L	 4P-L & 12P-L	 6P-L & 10P-L	 2(8P-L)	 4P-L & 2(6P-L)	 2(4P-L)& 8P-L	 4(4P-L)

Fig. 3 -b Seven kinds of bipartite graph of 16 x 16 MI network.

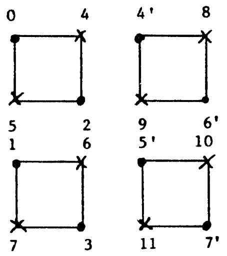


Fig. 4-a Modified bipartite graphs marked by '•' and 'x' as input and output nodes respectively.

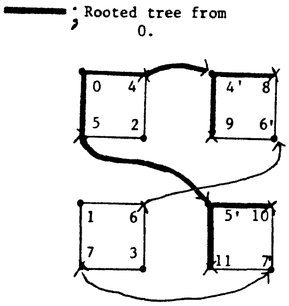
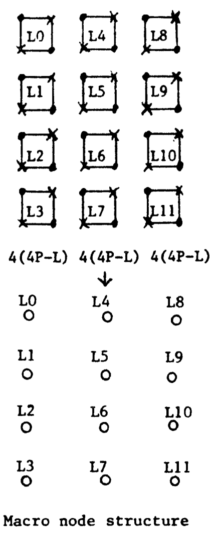


Fig. 4-b Full connectivity is satisfied in this connection. The distance between input 4 and 5, for example, is greater than 3.



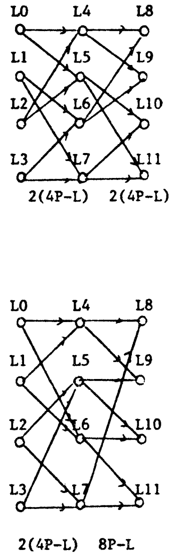


Fig. 5-a Loop structure of 4(4P-L);4(4P-L);4(4P-L).

Fig. 5-b Macro loop diagram

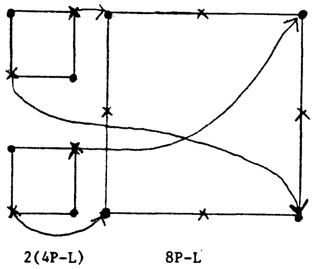


Fig. 4-c loop structure of 2(4P-L);8P-L.

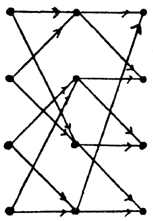


Fig. 4-d One example of graph which has loop structure of Fig. 4-c.

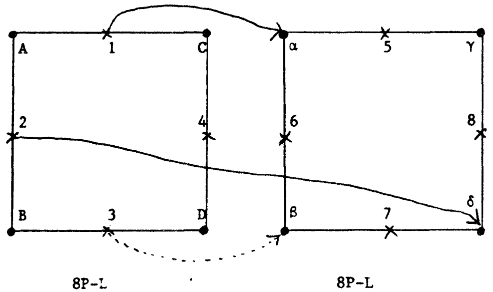
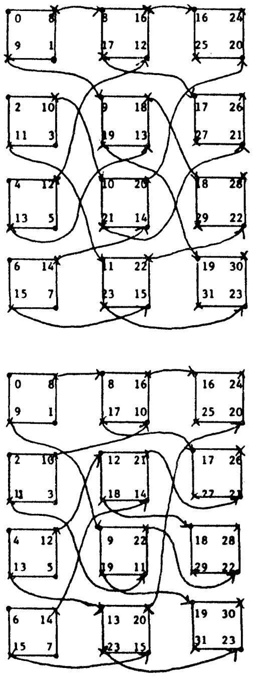


Fig. 4-e Loop structure which does not meet full connectivity.



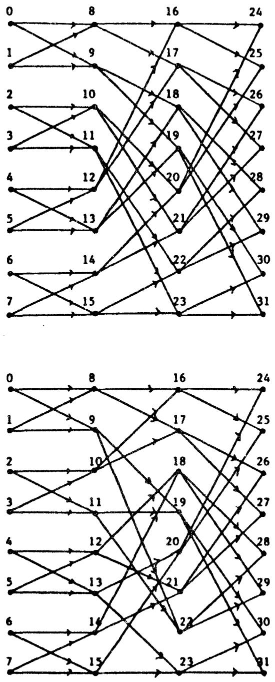


Fig 5-c Two different ways of connecting loops.

Fig. 5-d Graph models resulted from Fig. 5-c. They are non-equivalent.

INTERCONNECTION TOPOLOGIES FOR FAULT-TOLERANT PARALLEL AND DISTRIBUTED ARCHITECTURES

by

Dhiraj K. Pradhan*

School of Engineering
Oakland University
Rochester, Michigan 48063

ABSTRACT

Communication architectures are presented, suited for the design of fault-tolerant parallel and distributed processors. Attractive features of these communication topologies include small internode distances, low interconnection complexity, ease of message routing, modularity, fault-tolerance, and reconfigurability.

I. INTRODUCTION

A key component of parallel/distributed processor architecture is the interconnection structure used for communication between processors, memories, etc. This paper presents a new class of interconnection topologies which combine some of the interesting features of the existing structures, like the loop [1], the binary tree [2], and regular networks [3]. The following elaborates on some of the important features of the proposed interconnection structures:

(i) suitability for parallel processor design: interconnection patterns - such as binary tree [2], perfect shuffle [4], inverse perfect shuffle [4], uniform shift [4] - are embedded in our structure. Thus, the topology presented here is well-suited for various parallel algorithms which are used for sorting, FFT, matrix manipulations [4]. Also, other algorithms which are based on principles of "divide and conquer" or recursion can be well-executed because of the imbedded binary tree structures.

Furthermore, our interconnection structures are amenable to parallel processor design in VLSI because of the simplicity of the layout and the low degree of interconnection.

(ii) suitability for distributed processor design: the proposed topologies have small internode distances, ease of message routing and modularity. Thus, they satisfy some of the key requirements to be useful as communication architectures for distributed systems [5].

(iii) fault-tolerance: fault-tolerance is becoming an increasingly important attribute of computer systems. The topologies given here are fault-tolerant against both node and link failures. Also these topologies possess relatively short redundant paths. Consequently, fault recovery and reconfiguration can easily be achieved without significant degradation.

This paper is organized into the following sections. Section II develops certain notations and definitions. Section III presents the proposed topologies. Also included here are certain related results and routing algorithms. Next in Section IV fault-tolerance, reconfigurability and modularity of the topologies are studied. Following this, techniques that modify the topologies are presented which achieve greater fault-tolerance.

II. NOTATIONS AND DEFINITIONS

An interconnection topology will be represented as an undirected graph, G, with n nodes, $\{0,1,2,\ldots,(n-1)\}$.

An edge, e(i,j), in G represents a bidirectional data link between i and j.

A pair of nodes, i and j, are neighbors of each other if there is an edge, e(i,j), in G.

The degree, d(i), of node i is the number of neighbors of i.

Let $d = \min\{d(i) \mid 0 \leq i \leq (n-1)\}$ represent the minimum of the degrees of nodes in G.

Let $D = \max\{d(i) \mid 0 \leq i \leq (n-1)\}$, represent the maximum of the degrees of nodes in G.

Let k(i,j) represent the distance between nodes i and j. The distance, k(i,j), is equal to the number of edges in the shortest path between i and j. If there are no paths between i and j, then k(i,j) will be assumed to be ∞.

Let $k = \max\{k(i,j) \mid 0 \leq i,j \leq (n-1)\}$ represent the diameter and be the maximum internode distance in G.

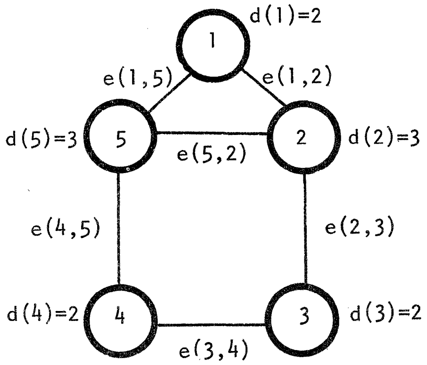
Let k_e represent the maximum of the diameters of all graphs that can be obtained from G by removing some e nodes from G. This k_e will be referred to as the e-diameter of G.

Thus, k_e can be viewed as a measure of the worst case propagation delay when e (faulty) nodes are removed from G.

Let c denote the node connectivity of G. Thus, c represents the minimum number of nodes which when removed from G, will disconnect G. Obviously, $c \leq d$ and $k_c = \infty$.

Thus, (c-1) will be referred to as the fault-tolerance of G.

Consider G, shown below. Here, d=2, D=3, k=2, k_1=3 and c=2.



[*] Research reported supported by AFOSR 80-0217

0190-3918/81/0000/0238$00.75 © 1981 IEEE

III. INTERCONNECTION TOPOLOGIES

This section presents two different classes of system topologies. Although both classes have certain similarities, they are significantly different in structure, and each class will be shown to possess certain distinct advantages over the other.

System Topology I (ST-I)

A pair of nodes, i and j, are neighbors if they satisfy any of the following relationships:

(a) $j = i+1 \mod n$
(b) $j = i-1 \mod n$
(c) $j = 2i \mod n$
(d) $i = 2j \mod n$

System Topology II (ST-II)

Nodes i and j are neighbors if they satisfy any of the following relationships:

(a) $i = 2j \mod n$
(b) $j = 2i \mod n$
(c) $i = 2j+1 \mod n$
(d) $j = 2i+1 \mod n$

Fig. 1 and Fig. 2 illustrate a 12-node ST-I and a ST-II, respectively.

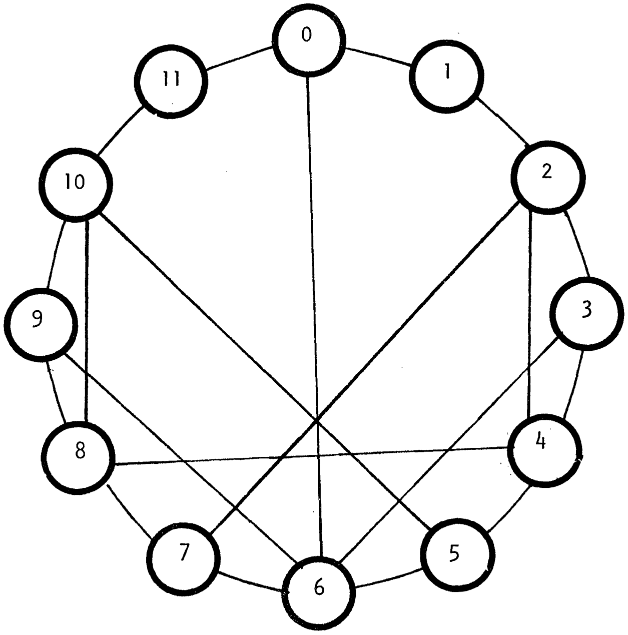


Fig. 1. ST-I for n=12

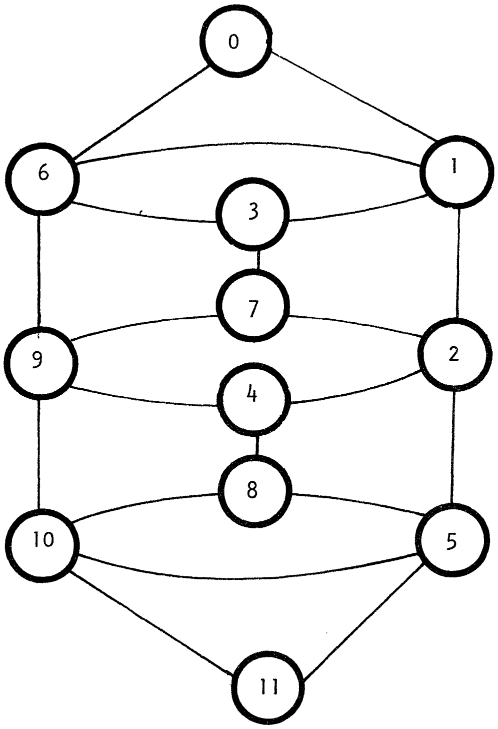


Fig. 2. ST-II for n=12

The ST-I represents a superimposition onto a basic loop structure. However, when compared to that basic loop structure, the ST-I will be seen to have significantly smaller internode distances; also, it is not vulnerable to single point failures.

The second topology, ST-II, (a generalization of Pradhan-Reddy [6]), has better internode distances than ST-I in the absence of any faults. On the other hand with a single node failure ST-I has the potential to achieve better internode distances than ST-II. Furthermore, ST-I enjoys certain implementational and functional advantages over the ST-II because of the following:

In the ST-I, unlike in the ST-II, two nodes that have consecutive logical addresses are neighbors; and thus, are adjacent to each other. Consequently, at least half of the links in the ST-I can be short and laid out on a single plane. So, data exchanges between i and i+1 (which constitute a major portion of the overall communication) can be done rapidly.

The following notations are used throughout this paper:

Notations: (i) [x] denotes the smallest integer that is greater than or equal to x.

(ii) log x denotes [$\log_2 x$].

(iii) all arithmetic operations used in respresenting node numbers are modulo-n operations.

The following theorems provide certain characterizations of ST-I and ST-II.

Theorem 1: In ST-I, if n=odd and $n \geq 5$, then:

(i) node 0 has degree 2
(ii) nodes 1, 2, (n-1) and (n-2) have degree 3 □
(iii) all other nodes have degree 4

Theorem 2: In ST-I, if n=even and $n \geq 8$, then:

(i) the nodes 1 and (n-1) have degree 2
(ii) all other odd nodes and the node 0 have degree 3
(iii) nodes 2, (n-2)(and n/3, 2n/3 if 3 divides n) have degree 4
(iv) all other even nodes have degree 5 □

Theorem 3: In ST-II, for $n \geq 8$:

(i) the nodes 0 and $(n-1)$ have degree 2

(ii) the nodes of degree 3

$$\text{are} \begin{cases} (n/3),\ (2n/3),\ (n-3)/3,\ (2n-3)/3, & \text{if 3 divides } n \\ (n-1)/3,\ (2n-2)/3, & \text{if 3 divides } (n-1) \\ (n-2)/3,\ (2n-1)/3, & \text{if 3 divides } (n-2) \end{cases}$$

(iii) all other nodes have degree 4 □

Theorem 4: In ST-I for all $n, n \geq 5$, the total number of data links $= \begin{cases} (2n-4) & \text{if 6 divides } n \\ (2n-3) & \text{otherwise} \end{cases}$ □

Theorem 5: In ST-II for all n, $n \geq 8$, the total number of data links $= \begin{cases} (2n-4) & \text{if 3 divides } n \\ (2n-3) & \text{otherwise} \end{cases}$ □

Thus, both ST-I and ST-II require a small number of data links and have low degree of interconnection.

The following develops message routing strategies and upperbounds on path lengths in ST-I and ST-II.

Message routing

The following develops a distributed routing algorithm for ST-I. Analogous procedures for ST-II can easily be formulated.

First a technique is developed to construct a path from a given source node, i to a destination node j. With this technique, a message routing algorithm is developed. This algorithm uses certain tag (control) bits. These bits are generated at the source and carried by the message. Each intermediate node use these tag bits to determine the next node in the path. Minor modifications of the tag bits are carried out by the intermediate nodes. As will be seen the routing procedure is very simple and requires only a small number of tag bits which are easily generated. Thus, the communication overhead can be fairly low.

Definition: Let m, $m \geq 0$, be the least integer for which:

$(j - i\ 2^{m-1}) \bmod n \leq 2^m - 1$.

Obviously there always exists such an m where $m \leq \log n$.

Let $p = (j - i\ 2^{m-1}) \bmod n$.

Let $p = p_{m-1} 2^{m-1} + p_{m-2} 2^{m-2} + \ldots + p_1 2 + p_0$ in radix-2 (binary).

The following procedure constructs a path from i to j in ST-I.

Path Construction Procedure, P1, for ST-I:

The following procedure is described by using a pointer, x, which travels through the path from i to j.

The value of x during each iteration represents successive nodes in the path.

```
S1: Let x=i and h=m
S2: Repeat following while $h \geq 1$:
    Let h=h-1, if $p_h=0$. Then let the new value of x=2x. On the other hand, if $p_h=1$, then first let x=(x+1), and next, x=2x.
S3: Finally, if $p_0=1$, then let x=(x+1), and stop else stop.
```

The following two observations may be made:

(i) the final value of x at the end of the procedure is $(i2^{m-1} + p)$ which is equal to j, the destination node.

(ii) the path length is, at most, $(2m-1)$.

Routing Strategy:

Each message may be formatted, as shown in Fig. 3. The message has two tag bit fields, α and β, which consist of m and log m bits.

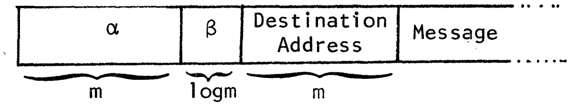


Fig. 3. Message Format

The source node, i, computes $p = (j - i\ 2^{m-1}) \bmod n$. Then α and β are initialized to p and $(m-1)$, respectively. Thus, at the start, $\alpha_q = p_q$, $0 \leq q \leq m-1$ and $\beta = (m-1)$.

When a message arrives at node y, the following steps are carried out. The source node also uses the following procedure to route the message to the next node.

Step 1: if the destination address is y, then the message is removed; otherwise Step 2 is performed.

Step 2: if $\alpha_\beta = 0$, then let $\beta = \beta - 1$, and the message is forwarded to node 2y next.

On the other hand, if $\alpha_\beta = 1$, then first let $\alpha_\beta = 0$ (change α_β to 0) and forward the message to the node, $(y+1)$. (Thus, if $\alpha_\beta = 1$ then the message is effectively forwarded to $2(y+1)$ via $(y+1)$).

Thus, the above procedure is simple and requires only $(m + \log m)$ bits - a small number compared to the number of nodes n since $m = \log n$. However, it should be noted that above procedure does not always result in routing the message through the shortest path.

Example 1: Let $i = 3$, $j = 9$. Consider routing from 3 to 9 in ST-I in Fig. 1 where n=12.

The least m that satisfies, $(j - i\ 2^{m-1}) \bmod n \leq 2^m - 1$, is $m = 2$.

Thus, $p = 3$ and hence, $\alpha = p = (11)$ and $\beta = m - 1 = 1$ in binary. (Initially $\alpha_1 = \alpha_0 = 1$.)

Hence, it can be computed that the message will go to node 9 via nodes 4 and 8. The tag bits at each node are shown in the following Table.

node	at arrival		at departure	
	α	β	α	β
3	11	1	01	1
4	01	1	01	0
8	01	0	00	0
9	00	0		

The following theorems are direct consequences for the routing procedures for ST-I and ST-II.

Theorem 6: For ST-I, the diameter, k, is bounded from above by

$$k \leq 2(\log n) - 1.$$ □

Theorem 7: For ST-II, the diameter, k, is bounded from above by

$$k \leq \log n. \qquad \square$$

IV. FAULT-TOLERANCE AND RECONFIGURABILITY

This section formulates results regarding the fault-tolerant capacity of the ST-I and ST-II. Techniques are also developed that reconfigure the paths in the event of a fault. The proposed reconfiguration will be shown to cause only a minimal degradation of the path lengths.

Although the fault-tolerance of the ST-I is self-evident, (because of its embedded loop structure) the effect of a fault on path lengths is not obvious. In the case of the ST-II, neither the fault-tolerance nor the effect of fault is readily apparent. The results of this section will provide some insight into these aspects.

The following describes a routing strategy for ST-I that can be used to bypass any faulty node with only a resulting minimal increase in the path length.

Routing procedure for ST-I with faulty node, t

From the routing procedure developed earlier for the ST-I, the following observations may be made:

(i) Any intermediate node, t, in a message path may receive the message from only two of its neighbors: the node (t-1) or (t/2).

(ii) There are only two possible neighbors of t to which the message may be forwarded from t; these are 2t and (2t+1).

In the following, we show a technique for sending the message from {(t-1) or (t/2)}to {(t+1) and 2t}without going through t. Consider the following paths where $3 \leq t \leq n-1$:

$$t-1 \longrightarrow 2t-2 \longrightarrow 2t-1 \longrightarrow 2t \longrightarrow 2t+1 \longrightarrow 2t+2 \longrightarrow t+1$$

$$t/2 \longrightarrow t/2+1 \longrightarrow t+2 \longrightarrow t+1 \longrightarrow 2t+2 \longrightarrow 2t+1 \longrightarrow 2t$$

It may be noted that the above paths are of length 6 and do not pass through t. Thus, if the message is routed through one of these alternate paths, then there will be a net increase of only 4 in the path length.

For t = 0,1,2, (n-1), (n-2), a set of paths of length, at most, 6 can also be constructed that allow for sending the message from {(t-1), (t/2)} to {(t+1), 2t} without going through t.

For example, let t = 2. In this case, (t-1)=(t/2)= 1, (t+1)= 3 and 2t = 4. Consider the following path for n = even:

$$1 \rightarrow 0 \rightarrow n/2 \rightarrow n/2+1 \rightarrow n/2+2 \rightarrow 4 \rightarrow 3$$

Thus, the node, 2, can be bypassed with a longer path.

The following is an immediate consequence of the above observations:

Theorem 8: The 1-diameter, k_1, of ST-I is bounded from above by: $k_1 \leq 2(\log n) +3$ $\square$

Thus, k_1 is comparable to k.

Now we will develop a routing procedure for ST-II. This procedure is significantly different from that given for ST-I, and is based as the following Lemmas:

Lemma 1: For any node, i, in ST-II, there is a path between 0 and i of length, at most, log n, where each intermediate node, y, in the path is strictly less than i, y<i. $\square$

Lemma 2: For any node, i, in ST-II, there is a path between i and (n-1) of length, at most, log n, where each intermediate node, y, in the path is strictly greater than i, y>i. $\square$

Lemma 3: There are two node disjoint paths between 0 and (n-1) in ST-II, of length, at most, log n. $\square$

A message routing algorithm that can be used in the event of a fault can be formulated by using the following path construction procedure P_2.

Notations: (i) Let $0 \rightarrow x$ $(x \rightarrow 0)$ represent a path, from 0 to x (x to 0), that satisfy Lemma 1.

(ii) Let $(n-1) \rightarrow x$ $(x \rightarrow (n-1))$ represent a path from (n-1) to x (x to (n-1)), that satisfy Lemma 2.

(iii) Let $0 \overset{y}{\rightarrow} (n-1)$ $((n-1) \overset{y}{\rightarrow} 0)$ represent a path of length, at most, log n from 0 to (n-1) ((n-1) to 0), that do not pass through the node, y. Lemma 3 guarantees the existence of such paths.

Path Construction Procedure, P2, for ST-II with Faulty Node t

i:—Source j:—Destination

Case	$t=0$
p(t):	$i \rightarrow (n-1) \rightarrow j$
Case II	$t=(n-1)$
p(t):	$i \rightarrow 0 \rightarrow j$
Case III	$t \neq 0$ and $t \neq (n-1)$
(a)	$t>i$ and $t<j$
p(t):	$i \rightarrow 0 \overset{t}{\rightarrow} (n-1) \rightarrow j$
(b)	$t>i$ and $t>j$
p(t):	$i \rightarrow 0 \rightarrow j$
(c)	$t<i$ and $t<j$
p(t):	$i \rightarrow (n-1) \rightarrow j$
(d)	$t<i$ and $t>j$
p(t):	$i \rightarrow (n-1) \rightarrow 0\ j$

The paths satisfying Lemmas 1, 2 and 3 can be constructed by using procedures that are similar to those given in Pl. Therefore, a routing strategy like the one formulated for ST-I, using tag bits, can also be developed here for ST-II. It may be noted that for ST-II, in the event of a fault, there is a potential for "bottleneck" because each message has to be forwarded through 0 or (n-1). The following Theorem is immediate from the above path construction procedure.

Theorem 9: The 1-diameter, k_1 of ST-II is bounded from above by

$$k_1 \leq 3(\log n) -1 \qquad \square$$

The following discusses possible modifications of ST-I and ST-II for greater fault-tolerance. It will be seen that the addition of a single link can make these 2-fault-tolerant; i.e., any two (faulty) nodes can be removed.

Since d=2, for ST-I, II and $d \geq c$, in order to increase the connectivity, c, one has to increase d. There are only two nodes of degree 2; therefore, one can increase d by simply adding a link that connects these two nodes. In the following, we show that the fault-tolerance is also increased with the addition of this extra link.

Modified System Topology I (MST-I)

$$\text{Add link} \begin{cases} e(0,2) \text{ to ST-I} & \text{if } n=\text{odd} \\ e(1,n-1) \text{ to ST-I} & \text{if } n=\text{even} \end{cases}$$

Modified System Topology II(MST-II)
Add link e(0,n-1) to ST-II.

By adding the link e(1,11) to Fig. 1 and e(0,11) to Fig. 2 one can obtain a 12 node MST-I and MST-II respectively.

Theorem 10: The connectivity, c, of MST-I is 3 □

Theorem 11: The connectivity, c, of MST-II is 3 and $k_2 \leq 2m$, when $n = 2^m$. □

MODULARITY

Modularity is an attribute of interconnection structures that refers to the ease with which incremental changes can be made.

The following illustrates a general technique for extending ST-I and ST-II, without significantly altering the existing system.

Let G be an existing ST with n nodes. If it is required that an additional n' node may be added, the following scheme may be used:

First, a new BST, G', is constructed using n' nodes. The nodes, 0 and (n-1) in G, are connected to 0' and (n'-1) in G, respectively.

This will not increase the D of either G or G' since nodes 0 and (n-1) have a degree of, at most, 3, in both ST-I and ST-II.

The diameter (1-diameter) of the composite system will be the sum of the diameters (1-diameters) of G and G'.

Subsequently, if it is required that a further set of n'' nodes is to be added, the following scheme may be used:

Construct a new ST G'', with n'' nodes. Then this is inserted between nodes 0 and 0' by connecting 0 and (n''-1), 0'' and 0' shown in Fig. 4.

The diameters will now again be the sum of the diameters of G, G' and G''. This latter process obviously can be repeated as many times as it may be required.

REMARKS:

1. (a) Given any node, y, one can reach nodes 2y and (2y+1) in, at most, two steps. Thus, data exchanges that correspond to a binary tree can be implemented easily for any node as a loot node.

(b) One can reach both (y/2) and (2y) from any node, y, in one step. Therefore, data exchange patterns that correspond to perfect shuffle and inverse perfect shuffle [4] can be implemented easily.

(c) Given enough buffer capacity at each node data exchange patterns that correspond to any arbitrary permutation can be implemented in order log n (O(log n)) steps.

Thus, from (a), (b) and (c), one may infer that the topologies are suited for fast implementation of various parallel processing algorithms.

2. Some of the key requirements for a topology to be suited for distributed processor interconnection is that it should possess a low degree of interconnection complexity, small internode distances, ease of routing and modularity. All of these requirements are satisfied by our topologies.

3. One of the other attractive features of the proposed topologies is that they are not only fault-tolerant but also easily reconfigurable.

4. Problems currently under investigation include formulation of good routing strategy that will allow for bypassing two faulty nodes without excessive degradation. (It would be of graph theoretical significance to derive a bound on the minimum number of edges required to achieve a specified k_e, for a given n and D.)

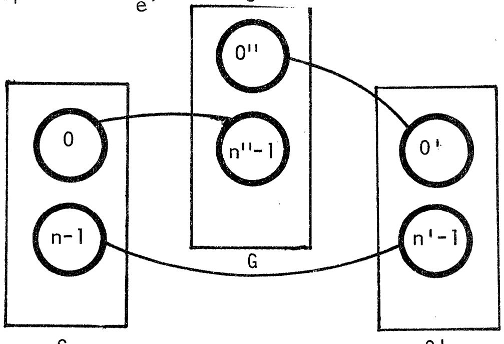


Fig. 4 Modular Extension

REFERENCES

[1] M.T. Liu et al, "System Design of the Distributed Double-loop Computer Network," Proc. of 1st International Conference on Distributed Computing System, Huntsville, Alabama, Oct. 1979.

[2] A.M. Despain and D.A. Patterson, "X-Tree, A Tree Structure Multi-processor Computer Architecture," Proc. 1980 Comp. Arch. Conf.

[3] K.J. Thurber and G.M. Masson, "Distributed Processor Communication Architecture," Lexington Books, Massachusetts, 1980.

[4] H.S. Stone, "Parallel Computers," in Introduction to Computer Architectures, Science Research Associates, 1980.

[5] H. Frank and I.T. Frisch, "Planning Computer-Communication Networks" in Computer Communication Networks, Prentice Hall, 1973.

[6] D.K. Pradhan and S.M. Reddy, "A Fault-tolerant Communication Architecture for Distributed Systems," Proc. of FTCS-11, IEEE Publications, June 1981.

FAULT DIAGNOSIS AND DESIGN OF FAULT-TOLERANT CONCENTRATORS

S. Sowrirajan and S. M. Reddy
Division of Electrical and Computer Engineering
The University of Iowa
Iowa City, Iowa 52242

Summary

Switching networks may be classified as connectors, concentrators, partitioners and expanders [1]. A concentrator is a contact switching network that provides a number of potential users (connected to its inputs) with access to a smaller number of equivalent resources (connected to its outputs) [2]. In a concentrator the outputs to which the inputs are to be connected cannot be specified a priori. A concentrator can be represented by the triplet (I,O,r), where I is the set of inputs, O is the set of outputs and r is the crosspoint placement relation between I and O. The capacity of a concentrator is said to be C if any k inputs, $k \leq C$, can be connected to some k outputs simultaneously.

Definition 1: $(n,m) = \binom{n}{m} = \frac{n!}{(n-m)!m!}$. A binomial (n,m) concentrator is a concentrator having (n,m) inputs, n outputs such that each input is connected to a unique choice of m out of n outputs by crosspoints. The capacity of binomial (n,m) concentrator is shown to be $\min\{m+2,n\}$ [3]. A binomial (4,2) concentrator is shown in Fig. 1.

Fault-Model of the Crosspoints:

Typically, in a crosspoint network a crosspoint is connected between an input line and output line as shown in Fig. 2. The crosspoint has two states under normal operation namely open and closed [4,5].

Definition 2: A crosspoint is said to be faulty if it is permanently in one of the two states namely, close and open. A crosspoint is said to be "stuck-at-close" (s-a-c) if it is permanently in the closed state and it is said to be "stuck-at-open" (s-a-o) if it is permanently in the open state. This is shown in Fig. 3.

Single Fault-Diagnosis in a Binomial (n,m) Concentrator:

A binomial (n,m) concentrator has $(n,m)\cdot m$ crosspoints and n outputs. Hence at most n crosspoints can be tested for a single s-a-o fault at any one instant - one crosspoint per output. Hence the optimal number of test sets to diagnose a single s-a-o fault is $(n,m)\frac{m}{n} = (n-1,m-1)$. The Theorem 1 says that such a test set can be obtained for a binomial (n,m) concentrator.

Definition 3: Let $U = (A_i\colon i \varepsilon I)$ be a family of subsets of a set E. Suppose that it is possible to select one element x_i from each set A_i in such a way that the elements x_i, $i \varepsilon I$, so selected are distinct. Then the set $\{x_i\colon x_i \neq x_j \ \forall\ i, j \varepsilon I$ and $i \neq j\}$ of these elements is called a transversal of U.

Definition 4: Let $B_i = \{j\colon j$ is an input such that there exists a crosspoint (j,i) and i is an output$\}$; $1 \leq i \leq n$. Since there are (n-1,m-1) crosspoints on each output line $|B_i| = (n-1,m-1)$, $1 \leq i \leq n$, where $|X|$ is the cardinality of set X.

Definition 5: A transversal T_j is said to be ordered if and only if $T_j = (x_{1,j}, x_{2,j}, \ldots, x_{i,j} \ldots, x_{n,j})$ where $x_{i,j} \varepsilon B_i$, $1 \leq j \leq (n-1,m-1)$ and $1 \leq i \leq n$.

Definition 6: Two ordered transversals T_i and T_j are disjoint if and only if $x_{k,i} \neq x_{k,j}$, $(\forall k)$ $1 \leq k \leq n$.

Theorem 1: (n-1,m-1) mutually disjoint ordered transversals $T_1, \ldots, T_{(n-1,m-1)}$ on the family $(B_i\colon i \varepsilon \{1,\ldots,n\})$ of subsets of inputs exist and can be calculated. For a proof see [7].

Procedure to Diagnose Single Fault in a Binomial (n,m) Concentrator:

Step 1: Set all the crosspoints to state open and apply a 1 to all the inputs. If there is an output such that a 1 is received on this output then there is a s-a-c faulty crosspoint on this output line and do Step 2; else go to Step 3.

Step 2: Test each crosspoint on this output line; stop.

Step 3: At instant j, $1 \leq j \leq (n-1,m-1)$ test the crosspoints $\varepsilon\, T_j'$ for s-a-o fault. If a s-a-o fault is diagnosed at instant j then stop; else if $j = (n-1,m-1)$ then there is no faulty crosspoint; else $j \leftarrow j+1$ and go to Step 3.

Procedure to Diagnose Multiple Faults in a Binomial (n,m) Concentrator:

Step 3 in the above procedure is to be modified to take into account s-a-c faulty crosspoints on output lines $x_1, x_2, \ldots, x_k$. If the crosspoint (y,x) is s-a-c and if $y \varepsilon T_j'$ for some j, $1 \leq j \leq (n-1,m-1)$, then while testing crosspoints $\varepsilon\, T_j'$, the output x will receive a 1 and hence the crosspoint (y_t,x) such that $y_t \varepsilon T_j'$ cannot be tested for s-a-o fault. This fact is taken into consideration to obtain the test sets. For more detail see [7].

Capacity of the Binomial (n,m) Concentrator in the Presence of a Single Fault:

Theorem 2: Let the crosspoint (a,b) be faulty. Then the capacity of the binomial (n,m) concentrator in the presence of a faulty crosspoint (a,b), denoted by $C_{(a,b)}$, is equal to (C-1) under the assumption that $m+2 \leq n$. For a proof see [7].

0190-3918/81/0000/0243$00.75 © 1981 IEEE

Capacity of the Binomial (n,m) Concentrator in the Presence of k Faults:

Theorem 3: The capacity, C_k, of the binomial (n,m) concentrator with $k < m$ faults $\geq$ (C-k) where C is the capacity of the fault free binomial (n,m) concentrator under the assumption that $m+2 \leq n$. For a proof see [7].

It can be shown that there exists a fault pattern of k crosspoints such that $C_k = C-k$.

The above two theorems give us a methodology to design a k fault tolerant, $m+k \leq n$, binomial concentrator. A binomial (n,m+k) concentrator has a capacity $\geq$ (C-k) if k or less faults occur where C is the capacity of binomial (n,m) concentrator. The number of inputs is however (n,m+k). The number of inputs can be increased to m•(n,m+k) without affecting the capacity of the concentrator with k faults.

Definition 7: An (m,0) concentrator is a concentrator with m inputs $x_1, x_2, \ldots, x_m$ and one output y_1 such that there exist crosspoints (x_i, y_1), $1 \leq i \leq m$.

Definition 8 [6]: An (m,0) × (n,m) concentrator has m•(n,m) inputs and n outputs such that each crosspoint in the (m,0) concentrator is replaced by the binomial (n,m) concentrator.

Theorem 4: An (m,0) × (n,m+k), $m+k \leq n$, concentrator in the presence of k faults has a capacity $\geq$ C, where C is the capacity of the binomial (n,m) concentrator. For a proof see [7].

References

[1] G.M. Masson, et al., "A Sampler of Circuit Switching Systems," Computer. Vol. 12, No. 6, pp. 32-48, June 1979.

[2] N. Pippenger, "On the Complexity of Strictly Nonblocking Concentration Networks," IEEE Trans. Comm., Vol. 22, 1974, pp. 1890-1892.

[3] G.M. Masson, "Binomial Switching Networks for Concentration and Distribution," IEEE Trans. Comm., Vol. 25, 1977, pp. 873-883.

[4] M. Rubin and C. E. Haller, "Communication Switching Systems," Reinhold Publishing Corporation, N.Y., 1966.

[5] K.M. So and J.J. Narraway, "Fault-Detection in Switching Networks," Proc. 20th Midwest Symp. Circuits Syst., Texas Tech. University, TX, pp. 552-556, Aug. 1977.

[6] S. Nakamura and G.M. Masson, "Higher Order Composite Concentrators," Proc. of the 1980 Conf. on Information Sciences and Systems, Princeton, pp. 531-535.

[7] S. Sowrirajan, "Fault Diagnosis and Tolerance in Connection Networks", Ph.D. thesis in preparation.

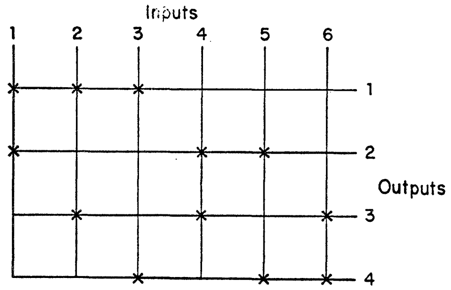


Fig. 1 $\binom{4}{2}$-binomial concentrator

x - crosspoint

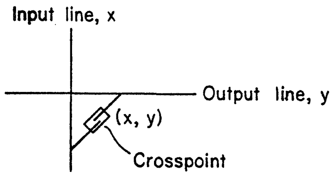
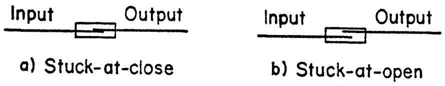


Fig. 2 A typical crosspoint



a) Stuck-at-close

b) Stuck-at-open

Fig. 3 Fault-model of a crosspoint

AN ALGORITHM FOR EFFICIENT LAYOUTS OF PARALLEL SUFFIX SOLUTIONS*

Avinoam Bilgory and Daniel D. Gajski

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

Abstract

The suffix problem has appeared in solutions of recurrence systems for parallel and pipelined machines and more recently in the design of gate and silicon compilers. In this paper we present an algorithm that generates parallel suffix solutions with minimum cost and size for a given length, time delay, availability of initial values and fanout. This algorithm generates a minimal solution for any length n and depth range from $\lceil \log_2 n \rceil$ to n.

1. Introduction

The suffix problem has appeared in solutions of recurrence systems for parallel and multiprocessor machines [Gajs81] and in design of gate and silicon compilers [GaBL81], [Kris81], and [LaFi80]. Many operations on a register-transfer level (addition, comparison, prioritization etc.) can be simply described by Boolean recurrence systems. The solution of the suffix problem is the single most important part of the more general solution of recurrence problems, which can be trivially extended to the solutions of fixed-length problems solved by finite-state transducers [LaFi80]. In this paper we give a new algorithm for generating area and time efficient parallel solutions for the suffix problem.

The solution of the recurrence system of length n and order 1, denoted by R<n,1>, is

$$x_i = f_i(x_{i-1})$$

for all i, $1 \le i \le n$, and given x_0. Furthermore,

$$\begin{aligned} x_i &= f_i(x_{i-1}) \\ &= f_i(f_{i-1}(x_{i-2})) = (f_i \circ f_{i-1})(x_{i-2}) = \cdots \\ &= (f_i \circ f_{i-1} \circ \cdots \circ f_2 \circ f_1)(x_0) \\ &= f_{i'}(x_0) \end{aligned}$$

where the symbol $\circ$ denotes the composition of functions. Thus, the solution of every recurrence system can be decomposed into two subproblems:

(a) Suffix problem: the computation of the functional composition $f_{i'} = f_i \circ f_{i-1} \circ \cdots \circ f_2 \circ f_1$ for all i, $1 \le i \le n$, and

(b) Functional evaluation: the computation of $x_i = f_{i'}(x_0)$ for all i, $1 \le i \le n$.

Subproblem (a) can be solved by a tree-like network, consisting of identical nodes, called Functional-Composition Cells (FCCs). Each FCC takes two functions f_i and f_j as inputs and generates their composition $f_i \circ f_j$.

Subproblem (b) is solved by n identical nodes, called Functional-Evaluation Cells (FECs). Each FEC takes a function $f_{i'}$ and its argument x_0 as inputs and generates $f_{i'}(x_0)$.

Detailed treatment of recurrences can be found in [BiGa81].

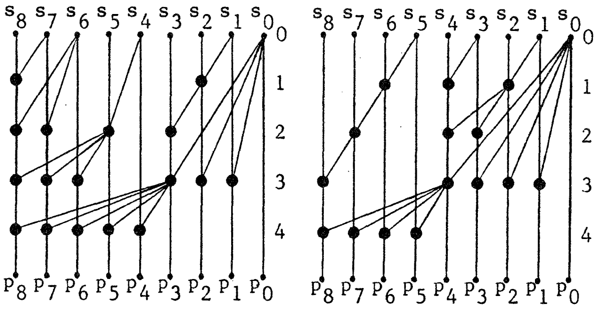
More abstractly, the suffix problem can be defined for any semigroup $<S,\circ>$: given $s_n, s_{n-1}, \ldots, s_1, s_0 \varepsilon S$, compute each of the products $p_k = s_k \circ s_{k-1} \circ \ldots \circ s_1 \circ s_0$ for all k, $0 \le k \le n$.

Each suffix-problem solution can be represented by a directed acyclic oriented graph. Each node of in-degree 2, called a product node, represents a product of its two inputs. The input nodes have in-degree 0 and are labeled with an element $s_i \varepsilon S$. The output nodes (the nodes that represent the solution of the suffix problem) have in-degree 1 and are labeled with an element $p_i \varepsilon S$. Hence, each node in the graph represents either an element from S or a product of some elements from S, which is itself an element from S. Two different graphs for the suffix problem of length 9 are shown in Figure 1. The numbers along the right side of the graphs denote levels, which correspond to time steps.

We will introduce several complexity measures used to characterize any suffix graph G. The size of G, s(G), is the number of product nodes in G, while the depth of G, d(G), is defined as the maximum number of product nodes on any directed path in G. Thus, d(G) equals the maximum level. The depth of G is proportional to the time delay through G. For the two graphs G_1 and G_2 shown in Figure 1, $s(G_1)=17$ and $s(G_2)=15$ while the depths of G_1 and G_2 are $d(G_1)=d(G_2)=4$. The cost c_i, of a $p_i \varepsilon S$, is the number of product nodes on the path from s_i to p_i. The cost of G, c(G), is defined as

* This work was supported in part by the National Science Foundation under grant No. US NSF MCS80-01561.

0190-3918/81/0000/0245$00.75 © 1981 IEEE


s_8 s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0
0
1
2
3
4
p_8 p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0

(a) (b)

Figure 1. Two graphs for suffix problem of length 9 and depth 4. (a) G_1. (b) G_2.

max c_i, $0 \leq i \leq n$. If the nodes are laid out in a two-dimensional array such that each node drives only nodes in the same row to the left and in the same column below, then the silicon area occupied by the layout of G is linearly proportional to the product $n \times c(G)$. In our example, $c(G_1)=4$ and $c(G_2)=3$. The fanout of G, f(G), is the maximum out-degree of any node in G. So, $f(G_1)=6$ and $f(G_2)=5$. The last parameter that characterizes the suffix graph G is the initial-value-availability of G, e(G), which is the minimum number of levels after which s_0 can be used as an input to a product node. s_0 is the first element and corresponds to the initial value of the recurrence system. For example, the initial value for parallel adders is the value of the input carry. This value may not be available at the same time as the rest of input values. For the two suffix graphs in Figure 1, $e(G_1)=e(G_2)=2$. In comparison of the two different graphs shown in Figure 1, the solution represented by G_2 requires less area of silicon, less power, and has smaller time delay then the solution represented by G_1. Furthermore, the fanout in G_2 is smaller then in G_1. This will eventually result in better performance in the implementation of G_2.

Many papers considered problems related to the suffix problem but never proposed algorithms based on suffix-problem solutions. Brent and Kung [BrKu79], considering a regular layout for parallel binary adders, proposed a suffix solution with depth $2\log_2 n$ and size $2n-2-\log_2 n$ with fanout f=2. Ladner and Fischer [LaFi80] were the first to define the suffix problem, although they call it a prefix problem. They developed an algorithm for constructing suffix solutions with minimum size for the given length n of the suffix problem and the depth of the solution. However, their algorithm works only for depth $\lceil \log_2 n \rceil \leq d \leq 2\lceil \log_2 n \rceil$. The solutions for lengths that are not an integer power of 2 are not optimal. They did not consider either the cost of the suffix-solution layouts or the fanouts of their solutions, although they have given an upper bound on fanout.

In this paper we present a different algorithm that generates suffix solutions with minimum cost and size for a given length n, depth d, initial-value-availability e, and fanout f. Our algorithm generates a minimal solution for any integer n. Furthermore, the depth range is extended to include all depths from $\lceil \log_2 n \rceil$ to n. The solution with depth n really represents a serial solution with size n and cost 1. In the case of binary adders this solution corresponds to a ripple-carry adder.

The complete algorithm consists of two parts and it is presented in the following two sections. In Section 2 we present Algorithm 1, which generates a minimum-cost solution for any given length, depth, initial-value-availability, and fanout. In Section 3 we present Algorithm 2, which takes the solution generated by Algorithm 1 and minimizes its size by using only local optimizations. Comments on our approach and description of some open problems are given in Section 4. Finally, in Section 5 we show that for all practical values of n our algorithm, in comparison with the Ladner-Fischer algorithm, generates suffix-solutions with smaller sizes.

2. An Algorithm to Construct the Minimum-Cost Graph

A graph that produces the solution for a suffix with length n+1, given the constraints d (depth), e (level after which s_0 is available), and f (fanout) is denoted by G<n,d,e,f>. Figure 2 shows G<8,4,2,5>. Figure 2-a has the following interpretation. Every vertical line is called a column. Each product node is represented by a number and is driven from two other nodes. One driving node is always above the driven node, in the same column, while the other driving node is located in some column to the right of the node and is called the right-driving node. Each group of nodes driven from the same right-driving node is connected by a horizontal line. The right-driving node is located in the column to the right of the group and one level above it, and their connection is represented by a diagonal line. The graph can be laid out in a number of rows that is equal to the cost c. The number that stands for a node indicates the number of the layout row that the node belongs to. Figure 2-b shows the relative locations of the nodes in the layout. Connections are not shown, but each node drives only nodes to its left in the same row, or below it in the same column. Figure 2-c shows a binary matrix representation of the graph. The presence of a node is represented by 1 and absence of a node by 0. Each column can be interpreted as a binary number. Let the uppermost row correspond to the least significant bit. Then the matrix can be represented by a series of numbers, as shown in Figure 2-d. Note that the series is strictly increasing from right to left. Figure 2 and Figure 1-b are different representations of the same graph.

To motivate the algorithm, let us look at the graph $G<2^k-1,k,0,2^{k-1}+1>$ which is generated recursively in an obvious manner. This graph may be represented by the series $<2^k-1,2^k-2,\ldots,3,2,1>$.

```
! ! ! ! ! ! ! !  0
! ! !/! !/! !/!
! ! 2 ! 1 ! 2 !  1
! !/! ! ! !/! !
! 2 ! ! 2-2 ! !  2
!/! ! ! ! ! ! !/
2 ! ! ! 3-3-3-3  3
! ! ! !/! ! ! !
3-3-3-3 ! ! ! !  4
! ! ! ! ! ! ! !
```

(a)

```
        1
2 2 2   2 2 2
3 3 3 3 3 3 3 3
```

(b)

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(c)

<12,10,9,8,7,6,5,4>

(d)

Figure 2. Graph G<8,4,2,5>.
(a) Connections scheme. (b) Layout.
(c) Binary matrix representation.
(d) Series representation.

Figure 3 shows this graph for k=4, namely, G<15,4,0,9>. Note that k is the lower bound for the depth of the solution for the suffix-problem of length 2^k, given e=0 and f=2^{k-1}+1.

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
! ! ! ! ! ! ! ! ! ! ! ! ! ! !  0
!/! !/! !/! !/! !/! !/! !/! !/
1 ! 2 ! 2 ! 3 ! 2 ! 3 ! 3 ! 4  1
! !/! ! ! !/! ! ! !/! ! ! !/!
2-2 ! ! 3-3 ! ! 3-3 ! ! 4-4 !  2
! ! ! !/! ! ! ! ! ! ! !/! ! !
3-3-3-3 ! ! ! ! 4-4-4-4 ! ! !  3
! ! ! ! ! ! ! !/! ! ! ! ! ! !
4-4-4-4-4-4-4-4 ! ! ! ! ! ! !  4
! ! ! ! ! ! ! ! ! ! ! ! ! ! !
```

Figure 3. Graph G<15,4,0,9>.

Three important observations can now be made:

(a) Column eliminations. 2^p-1 columns, 0≤p<k, can be eliminated from the right side of the graph as well as any number of columns from the left side of the graph. The remaining graph still represents a suffix problem solution. For example, eliminating columns 1, 2, and 3 from the right side and columns 14 and 15 from the left side changes the graph G<15,4,0,9> to G<10,4,2,7>. More important, certain columns can also be eliminated from the middle of the graph, as a result of the following Lemma 1.

Lemma 1

In G<n,d,e,f>, column j, n>j>1, can be eliminated if $c_{j+1}<c_j$. The remaining graph is G<n-1,d,e,f′>, f′<f. □

Proof

Let $s_{j,i}$ denote $s_j \circ s_{j-1} \circ \ldots \circ s_{i+1} \circ s_i$ for any i,j, n>j>i>0. From the construction of G<2^k-1,k,0,2^{k-1}+1> shown previously, it is obvious that any section of two consecutive columns (j+1,j) will generally look similar to that shown in Figure 4. That is, there is no node in column j+1 above node A, and if there is a node below node A in column j+1, then there is also a node on the same level in column j, and vice versa. There cannot be a node in column j on the level right below node B. Also, j>x>y.

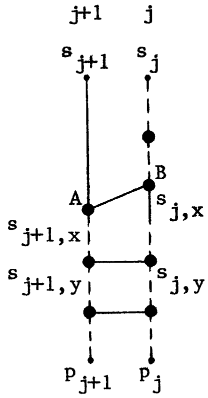


Figure 4. General two-column section of the graph G<2^k-1,k,0,2^{k-1}+1>.

The transformation shown in Figure 5 can now be carried out, resulting in the elimination of column j. Figure 5-a shows a portion of three consecutive columns (j+1,j,j-1), where $c_{j+1}<c_j$. In Figure 5-b, the uppermost node of column j+1 is disconnected from node U and connected to node V (in fact, every node in column k, k>j, driven by node U will now be driven by node V). Column j is renamed j′ and every column k and input node s_k, n≥k>j, are renamed k-1 and s_{k-1}, respectively. The lowest node in each renamed column k, n-1≥k>j, produces now p_k. The lowest node in the new column j produces p_j as does the lowest node in column j′. Column j′ no longer contains any right-driving-node, so it can be eliminated, resulting in the section shown in Figure 5-c. Column n has been left untouched, and since c_1=1, column 1 cannot be eliminated. Therefore d and e remain unchanged. The fanout may only be reduced. □

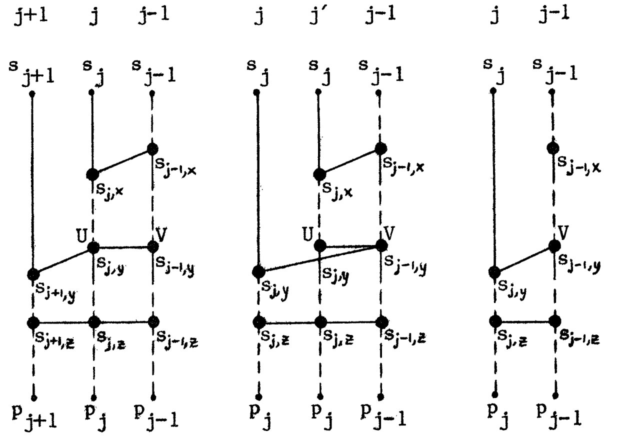


Figure 5. Elimination of column j.
(a) Original section. (b) Change of connection.
(c) Transformed section.

Since the graph portion shown in Figure 5-c preserves the properties of the general two-column portion shown in Figure 4, the transformation can be executed iteratively on the transformed graph.

In particular, columns having the largest number of nodes can be eliminated. For example, columns 3, 7, and 11 in $G<15,4,0,9>$ have this property and can be eliminated from the graph. Note that after applying Lemma 1 on $G<2^k-1,k,0,2^{k-1}+1>$ 2^k-k-1 times, the graph is reduced to $G<k,k,0,2>$.

(b) Segments. Each column has one FEC as the lowest node and zero or more FCCs above it. The graph is divided into d-e contiguous segments, according to the level where the FEC is located. Segment t contains columns that have their FECs on level t. In $G<15,4,0,9>$ there are four segments: 1, 2, 3, and 4. In segment t, the FCCs occupy the t-1 first levels: there are $\binom{t-1}{0}=1$ columns with no FCC, $\binom{t-1}{1}=t-1$ columns with one FCC in each and in general there are $\binom{t-1}{i}$ columns with i FCCs.

(c) Fanout. The leftmost FEC in each segment serves as a right-driving node for all the FECs of the next segment to its left. Therefore, the biggest FEC fanout equals the number of columns in the largest segment plus one. For example, the biggest fanout in $G<15,4,0,9>$ is 9. Each FCC drives only nodes in the segment it belongs to, so the biggest FCC fanout is always less then the biggest FEC fanout. Therefore, if we do not allow the fanout of the leftmost FEC in each segment to exceed f when constructing the graph, the fanout of every other node is automatically taken care of.

The minimum cost solution for $G<n,d,e,f>$ will be denoted $G_c<n,d,e,f>$. We will now present a simple but neither time-efficient nor space-efficient algorithm to construct $G_c<n,d,e,f>$. A more sophisticated algorithm that generates the same graph but takes linear time and constant space is given in the Appendix.

At any moment, n′ denotes the number of columns in the graph, n_t denotes the number of columns in each segment t, j_t denotes the column with the maximum cost in segment t, and j denotes the column with the maximum cost in the graph (if more then one column qualifies for j_t or j, any of them will do).

Algorithm 1

```
1. Construct the graph $G<2^d-1,d,0,2^{d-1}+1>$.
   If n′<n, stop. $G_c<n,d,e,f>$ is not realizable.
   Else continue to the next step.
2. If e>0, eliminate columns 1 through $2^e-1$.
   If n′<n, stop. $G_c<n,d,e,f>$ is not realizable.
   Else continue to the next step.
3. For each segment t do
   while $n_t>f-1$ do eliminate column $j_t$.
   If n′<n, stop. $G_c<n,d,e,f>$ is not realizable.
   If n′=n, stop. The current graph is $G_c<n,d,e,f>$.
   Else continue to the next step.
4. While n′>n do eliminate column j.                    □
```

Example: construct $G_c<8,4,2,5>$.

1. Construct $G<15,4,0,9>$ (see Figure 3).
2. Eliminate columns 1 through 3.
3. Eliminate columns 15, 14, 13, and 11 in segment 4.

We are now left with 8 columns. Therefore the graph is $G_c<8,4,2,5>$, as shown in Figure 2. □

As d approaches n, the algorithm becomes more inefficient in terms of space and time. However, if we compute in advance the maximum column cost and how many such columns are in each segment, then the graph can be constructed column by column. An efficient algorithm that exploits this idea has been devised, so that it takes time proportional to n. The details are given in the Appendix.

For the graph $G_c<n,d,e,n+1>$, a simple formula that implicitly gives the cost can be derived, using elementary combinatorial considerations. The cost is the smallest c that satisfies Inequality 1:

$$\sum_{i=1}^{c} \left(\binom{d}{i}-\binom{e}{i}\right) \geq n \qquad (1)$$

Theorem 1

The cost of $G_c<2^k,k+2,k+1,2^k+1>$ is $\lceil k/2 \rceil+1$. □

Proof

According to Algorithm 1, we execute the following steps:

1. Construct $G<2^{k+2}-1,k+2,0,2^{k+1}+1>$.
2. Eliminate columns 1 through $2^{k+1}-1$.
3. This step is not executed, since $f=2^{k+1}+1$.
4. The graph has now 2^{k+1} columns, from which 2^k columns have to be eliminated (in fact, the current graph is $G_c<2^{k+1},k+2,k+1,2^{k+1}+1>$). The number of columns having the cost i+1 is $\binom{k+1}{i}$, $0 \leq i \leq k+1$. Since $\sum_{i=0}^{k+1}\binom{k+1}{i}=2^{k+1}$ and $\binom{k+1}{i}=\binom{k+1}{k-i+1}$, $0 \leq i \leq k+1$, after the elimination of 2^k columns the maximum cost of the remaining columns is $k/2+1$ for an even value of k or $(k+1)/2+1$ for an odd k value. □

Theorem 1 implies the following important consequence. Obviously the cost of $G_c<2^k,k+1,k,2^k+1>$ (the graph with the minimum depth for $n=2^k$, e=k, and $f=2^k+1$) is k+1. Then by allowing the depth d (as well as e) to be greater by just one, the cost is reduced to about half (refer also to Table 1-a, Section 5).

Figure 6 shows two graphs generated by Algorithm 1.

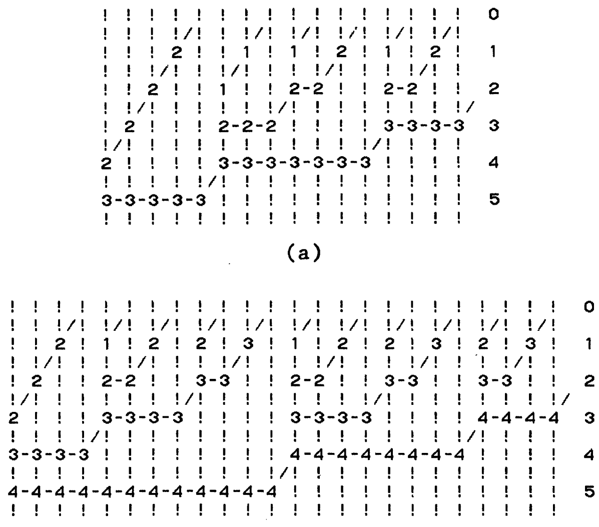


Figure 6. Minimum-cost graphs.
(a) G_c<16,5,2,8>. (b) G_c<24,5,2,13>.

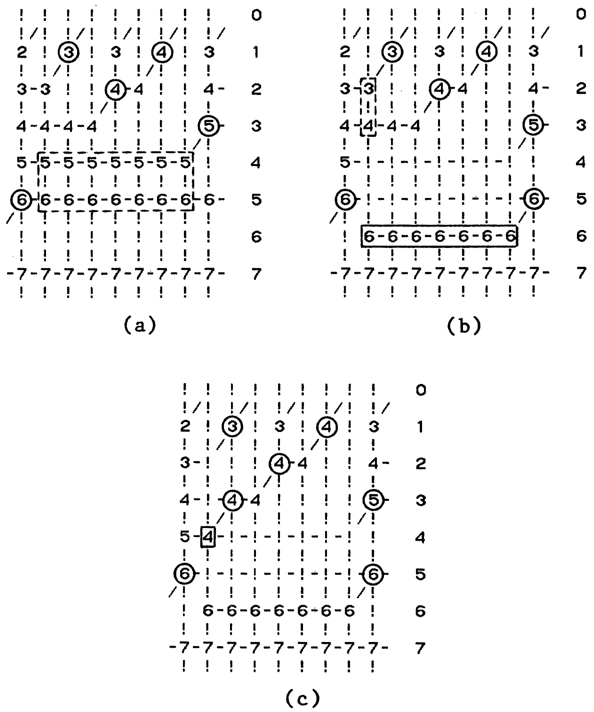


Figure 7. Local-optimization of columns 24 to 32 of G_c<64,7,6,65>. (a) Original section.
(b) Section after first local-optimization step.
(c) Section after second local-optimization step.

3. An Algorithm to Construct the Minimum-Size Graph

This algorithm operates on a graph G_c<n,d,e,f> generated by Algorithm 1 and generates a minimum-size graph, denoted as G_s<n,d,e,f>. The algorithm performs local optimization on G_c<n,d,e,f> by trying to reduce the number of nodes wherever possible. Figure 7 illustrates how local optimization is accomplished. Figure 7-a shows a section of the graph G_c<64,7,6,65> (columns 24 to 32). The nodes surrounded by the dashed box are deleted, and new nodes, surrounded by a full box, are introduced as shown in Figure 7-b. This transformation, called the local-optimization step, reduces the number of nodes in the graph while preserving the suffix solution p_k, 0<k<64. This can be shown by reasoning similar to that utilized in Lemma 1. A local-optimization step can be applied once more on the nodes surrounded by the dashed box in Figure 7-b. The final result is shown in Figure 7-c.

Note that right-driving-nodes may not be deleted, because of their connections to the nodes driven by them. Thus, the nodes are classified into two groups: (a) fixed nodes, which are the right-driving nodes, and (b) non-fixed nodes, which are all the other nodes. Fixed nodes are circled in figure 7.

Figure 8 illustrates the general local-optimization step. The nodes shown in Figure 8-a, except for node F, are non-fixed. There may be nodes on levels v_u+1 to v_d-1, but if there is a node on level v, v_d<v<v_u in column j, k>j>i, then there must be a node on level v in each column j, and these nodes are non-fixed, also. There are no nodes on level v_d+1 in this section. Figure 8-b shows the section after the local-optimization step. The nodes on levels v_u through v_d in columns i+1 through k are deleted and new non-fixed nodes are introduced on level v_d+1 in these columns. Node A is marked as fixed, since it now drives the group of new nodes. The effect of the transformation is the reduction of the size of the graph. We are now ready to state the algorithm.

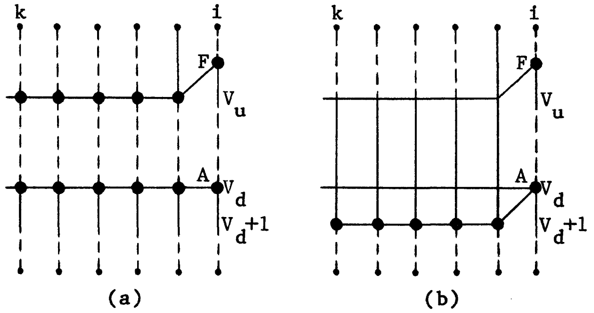


Figure 8. General local-optimization step.
(a) Section before step. (b) Section after step.

Algorithm 2

1. Mark fixed nodes.
2. Scan the graph and perform the local-optimization step wherever possible. □

Notes: (a) The cost of the graph is not changed.

(b) f is the upper bound for G_s<n,d,e,f>. There may or may not be any node with fanout equal to f.

Step 2 of this algorithm is somewhat vague, since the local-optimization step can replace any number of rows of non-fixed nodes with one row of new non-fixed nodes. However, if we restrict ourselves to replacing only two rows at each local-optimization step, we can limit the scanning to each individual column independently, as follows. The column is scanned from bottom to top. If a triple of levels $\{v_a, v_b, v_c\}$, $v_a = v_b + 1$, $v_b > v_c$ can be found such that non-fixed nodes are located on levels v_b and v_c and the levels between v_b and v_c as well as level v_a are empty, then these two nodes are deleted and a new non-fixed node is introduced on level v_a. The net effect of this local-optimization step is the reduction of the number of nodes by one. Local-optimization steps are carried out iteratively on the column until no such sequence is found. Figure 9 demonstrates the local-optimization done on column 31 of G_c<64,7,6,65> (compare it to Figure 7).

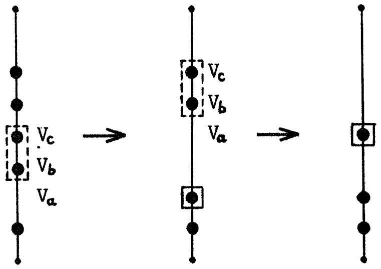


Figure 9. Local-optimization on a graph column.

Figure 10 shows the graphs of Figure 6 after applying local-optimization on each column.

```
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! !   0
! ! ! !/! ! !/! !/! !/! !/! !/!
! ! ! 2 ! ! 1 ! 1 ! 2 ! 1 ! 2 !   1
! ! !/! ! !/! ! ! !/! ! ! !/! !
! ! 2 ! ! 1 ! ! 2-! ! ! 2-! ! !   2
! !/! ! ! ! ! !/! ! ! ! ! ! ! !/
! 2 ! ! ! 2-!-! ! ! ! ! 3-!-3-3   3
!/! ! ! ! ! ! ! ! ! ! !/! !/! !
2 ! ! ! ! 3-!-!-3-!-3-3 ! 3 ! !   4
! ! ! ! !/! ! !/! !/! ! ! ! ! !
3-3-3-3-3 ! 3-3 ! 3 ! ! ! ! ! !   5
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! !
```

(a)

```
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! !   0
! ! !/! !/! !/! !/! !/! !/! !/! !/! !/! !/! !/!
! ! 2 ! 1 ! 2 ! 2 ! 3 ! 1 ! 2 ! 2 ! 3 ! 2 ! 3 !   1
! !/! ! ! !/! ! ! !/! ! ! !/! ! ! !/! ! ! !/! !
! 2 ! ! 2-! ! ! 3-3 ! ! 2-2 ! ! 3-! ! ! 3-! ! !   2
!/! ! ! ! ! ! !/! ! ! ! ! ! ! !/! ! ! ! ! ! ! !/
2 ! ! ! 3-!-3-3 ! ! ! ! 3-!-!-! ! ! ! ! 4-!-4-4   3
! ! ! !/! !/! ! ! ! ! ! ! ! ! ! ! ! ! !/! !/! !
3-3-3-3 ! 3 ! ! ! ! ! ! 4-!-!-!-4-!-4-4 ! 4 ! !   4
! ! ! ! ! ! ! ! ! ! ! !/! ! ! !/! !/! ! ! ! ! !
4-4-4-4-4-4-4-4-4-4-4-4 ! 4-4-4 ! 4 ! ! ! ! ! !   5
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! !
```

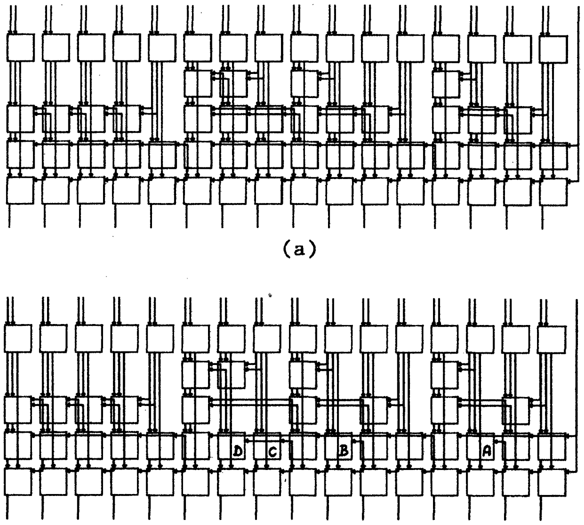
(b)

Figure 10. Minimum-size graphs. (a) G_s<16,5,2,8>. (b) G_s<24,5,2,13>.

4. Comments and Open Problems

The local-optimization step can replace any number of rows of non-fixed nodes with one row of non-fixed nodes. By choosing a different number of rows to be replaced in each local-optimization step we may end up with graphs of different sizes. Generally, the size reduction is greater as the cost of the graph grows. It is an open problem on how to choose the right number of rows to be replaced in each section during a local-optimization step, in order to achieve a smaller-sized graph. Also, Algorithm 2 operates on the graph generated by Algorithm 1, which eliminates columns according to their cost. If another strategy for eliminating columns is used by Algorithm 1, it may result in a smaller-sized graph generated by Algorithm 2. Another open problem is finding the expressions for the size of G_c<n,d,e,f> and G_s<n,d,e,f>.

Figure 11-a shows a structure of a 16-bit adder generated by a silicon compiler using Algorithm 1. Each square is a cell that represents a node. Rows 2, 3, and 4 from the top form G_c<16,5,2,8> (compare it to Figure 6-a). In Figure 11-b, Algorithm 2 was used, so these three rows form G_s<16,5,2,8> (compare it to Figure 10-a). 4 cells have been saved, but additional connecting lines which enter cells A, B, C, and D are introduced. For details of the cell functions and layouts see [GaBL81].



(a)

(b)

Figure 11. Structure of a 16-bit adder. (a) Using G_c<16,5,2,8>. (b) Using G_s<16,5,2,8>.

Algorithm 1 accepts the depth d as a parameter and generates the minimum-cost graph. A similar algorithm can be easily devised that accepts the cost c as a parameter and generates the minimum-depth graph, denoted as G_d<n,c,e,f>. G_d<n,c,e,f> is realizable for every n, c, e, and f. For the graph G_d<n,c,e,n+1>, the depth is the smallest d that satisfies Inequality 1.

5. Results and Comparison

In order to be able to compare our algorithms with the Ladner-Fischer algorithm, we will restrict ourselves to the graphs $G_c<n,d,d-1,n+1>$ and $G_s<n,d,d-1,n+1>$, where n is an integer power of 2. In these graphs, all the FECs occupy level d.

Table 1-a shows the cost of $G_c<n,d,d-1,n+1>$. The cost approaches 2 as d approaches n, for n>2. Table 1-b shows the sizes of these graphs. Table 1-c shows the sizes of these graphs after applying local-optimization on each column independently, as described at the end of Section 3. Table 1-d shows the sizes of the graphs generated by the Ladner-Fischer algorithm. This algorithm covers only the range $\log_2 n+1 \leqslant d \leqslant 2\log_2 n+1$, whereas our algorithm covers the full range $\log_2 n+1 \leqslant d \leqslant n$. For all practical cases, $G_s<n,d,d-1,n+1>$ has a smaller size then the graph generated by the Ladner-Fischer algorithm and sometimes even $G_c<n,d,d-1,n+1>$ has a smaller size (especially when d approaches n).

n\d	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
2	-	2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
4	-	-	3	2	-	-	-	-	-	-	-	-	-	-	-	-	-
8	-	-	-	4	3	3	3	2	-	-	-	-	-	-	-	-	-
16	-	-	-	-	5	3	3	3	3	3	3	3	3	3	3	2	-
32	-	-	-	-	-	6	4	4	3	3	3	3	3	3	3	3	3
64	-	-	-	-	-	-	7	4	4	4	4	3	3	3	3	3	3
128	-	-	-	-	-	-	-	8	5	4	4	4	4	4	4	4	3
256	-	-	-	-	-	-	-	-	9	5	5	5	4	4	4	4	4

(a)

n\d	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
2	-	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
4	-	-	8	7	-	-	-	-	-	-	-	-	-	-	-	-	-
8	-	-	-	20	18	17	16	15	-	-	-	-	-	-	-	-	-
16	-	-	-	-	48	41	40	39	38	37	36	35	34	33	32	31	-
32	-	-	-	-	-	112	98	90	86	85	84	83	82	81	80	79	78
64	-	-	-	-	-	-	256	218	209	199	188	179	178	177	176	175	174
128	-	-	-	-	-	-	-	576	500	455	444	432	419	405	390	374	366
256	-	-	-	-	-	-	-	-	1280	1093	1036	968	931	917	902	886	869

(b)

n\d	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
2	-	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
4	-	-	8	7	-	-	-	-	-	-	-	-	-	-	-	-	-
8	-	-	-	20	18	17	16	15	-	-	-	-	-	-	-	-	-
16	-	-	-	-	47	41	40	39	38	37	36	35	34	33	32	31	-
32	-	-	-	-	-	106	88	87	86	85	84	83	82	81	80	79	78
64	-	-	-	-	-	-	232	198	182	181	180	179	178	177	176	175	174
128	-	-	-	-	-	-	-	497	415	399	372	371	370	369	368	367	366
256	-	-	-	-	-	-	-	-	1049	897	802	800	766	753	752	751	750

(c)

n\d	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
2	-	3	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-
4	-	-	8	8	8	-	-	-	-	-	-	-	-	-	-	-	-
8	-	-	-	20	19	19	19	-	-	-	-	-	-	-	-	-	-
16	-	-	-	-	47	43	42	42	42	-	-	-	-	-	-	-	-
32	-	-	-	-	-	106	94	90	89	89	89	-	-	-	-	-	-
64	-	-	-	-	-	-	232	201	189	185	184	184	184	-	-	-	-
128	-	-	-	-	-	-	-	497	423	392	380	376	375	375	375	-	-
256	-	-	-	-	-	-	-	-	1048	880	806	775	763	759	758	758	758

(d)

Table 1. Cost and size of suffix-graphs.
(a) Cost of $G_c<n,d,d-1,n+1>$.
(b) Size of $G_c<n,d,d-1,n+1>$.
(c) Size of $G_s<n,d,d-1,n+1>$.
(d) Size of graphs generated by the Ladner-Fischer algorithm.

References

[BiGa81] Bilgory, A. and Gajski, D. D., "Automatic Cell Generation for Recurrence Structures," 18th Design Automation Conference, 1981.

[BrKu79] Brent, R. P. and Kung, H. T., "A Regular Layout for Parallel Adders," Tech. Report, CMU-CS-79-131, Dept. of Computer Science, Carnegie-Mellon Univ., June 1979.

[GaBL81] Gajski, D. D., Bilgory, A. and Luhukay, J., "Algorithmic Layout of Gate Macros," Proc. 2nd Caltech conf. on VLSI, January 1981.

[Gajs81] Gajski, D. D., "An Algorithm for Solving Linear Recurrence Systems on Parallel and Pipelined Machines," IEEE Trans. on Comp., Vol. C-30, No. 3, March 1981.

[Kris81] Krishnan, M. S., "A Structured Approach to VLSI Layout Design," Proc. 2nd Caltech conf. on VLSI, January 1981.

[LaFi80] Ladner, R. E. and Fischer, M. J., "Parallel Prefix Computation," J. ACM, Vol. 27, October 1980, pp. 831-838.

Appendix

We present here an efficient algorithm to construct $G_c<n,d,e,f>$. First, we will introduce notations for the algorithm. The word "current" means "at the moment of execution", as opposed to "final", which means "after completion of the algorithm". The parameters n, d, e, and f have been defined in Section 1. Some notations have slightly different meaning in each part of the algorithm. References to the algorithm parts are made where confusion might be caused.

c - Cost of current graph (part a);
  Cost of final graph (part b).

t - Segment number.

c_t - Cost of current segment t (equals the maximum cost of a column that belongs to segment t) (part a);
  Cost of final segment t (part b).

c'_t - Maximum cost of column to be added to segment t.

q - Maximum number of columns each having cost c that current segment t may have if the fanout is unlimited.

Δn - Number of columns still to be added to current graph.

Δf - Number of columns still to be added to current segment t.

n_t - Number of columns in current segment t.

m_t - Number of columns with cost c_t in current segment t (part a);
  Number of columns with cost c_t in final segment t (part b).

s_t - Status of current segment t:
  open: n_t<f-1 (columns may be added to current segment t);

Closed: n_t=f-1 (segment t is full or maximum fanout has been reached).

f′ - Current fanout of the leftmost FEC in segment t-1.

j - Column number.

b - Binary number representing column j.

u(b) - Position of the least significant 1 in b (l.s.b. is in position 0).

w(b) - Position of the most significant 1 in b (l.s.b. is in position 0).

z(b) - Cost of column represented by b (equals number of 1′s in b).

Algorithm 1′

Part a: Evaluation of c_t and m_t for each segment t in the graph.

```
{ initialization }
for t ← e+1 to d do
begin
    $n_t$ ← 1
    $s_t$ ← open
end
$\Delta n$ ← n
c ← 0

{ main loop }
repeat
    c ← c+1
    t ← e
    repeat
        t ← t+1
        if $s_t$=open then
        begin
            q ← $\binom{t-1}{c-1}$
            if q=0 then $s_t$ ← closed
            else
            begin
                $\Delta f$ ← f-$n_t$
                $c_t$ ← c
                if q<$\Delta f$ and q<$\Delta n$ then
                begin
                    $n_t$ ← $n_t$+q
                    $m_t$ ← q
                    $\Delta n$ ← $\Delta n$-q
                end
                else if $\Delta f$<q<$\Delta n$ or $\Delta f$<$\Delta n$<q then
                begin
                    $s_t$ ← closed
                    $m_t$ ← $\Delta f$
                    $\Delta n$ ← $\Delta n$-$\Delta f$
                end
                else { $\Delta n$<q<$\Delta f$ or $\Delta n$<$\Delta f$<q }
                begin
                    $m_t$ ← $\Delta n$
                    $\Delta n$ ← 0
                end
            end
        end
    until $\Delta n$=0 or t=d
until $\Delta n$=0 or $\Delta n$ has not been changed
```

Part b: Graph generation.

```
t ← e
f′ ← f
for j ← 1 to n do
begin
    if f′=f then
    begin
        t ← t+1
        b ← $2^{t-1}$
        generate column j
        f′ ← 2
        $c'_t$ ← $c_t$
    end
    else
    begin
        if z(b)=$c'_t$ then b ← b+$2^{u(b)}$ else b ← b+1
        if w(b)>t-1 then
        begin
            t ← t+1
            b ← $2^{t-1}$
            generate column j
            f′ ← 2
            $c'_t$ ← $c_t$
        end
        else
        begin
            generate column j
            f′ ← f′+1
            if z(b)=$c'_t$ then
            begin
                $m_t$ ← $m_t$-1
                if $m_t$=0 then
                begin
                    b ← b-$2^{u(b)}$
                    $c'_t$ ← $c'_t$-1
                end
            end
        end
    end
end
```

The statements "generate column j" in part b of the algorithm mean "use the binary number b as column j in the binary matrix representation of G_c<n,d,e,f>". We have shown how to perform local optimization on each column independently, so if "generate column j" is replaced by "generate and local-optimize column j", the algorithm will produce G_s<n,d,e,f>.

PIN LIMITATIONS AND VLSI INTERCONNECTION NETWORKS*

Mark A. Franklin and Donald F. Wann
Department of Electrical Engineering
Washington University
St. Louis, Missouri 63130

Abstract: Multiple processor interconnection networks can be characterized as having N' inputs and N' outputs, each B' bits wide. Construction of large networks requires partitioning of the N'*N'*B' network into a collection of N*N switch modules of data size B (B < B') each implemented on a single chip and interconnecting them with a specific interchip network type T'. The major constraint in the VLSI environment is the pin limitation, N_p, of the individual modules; these are allocated between data and control lines, Q. This paper presents a methodology for selecting the optimum values for N and B given values of the parameters, N', B', T', Q, and N. Models for both the banyan and crossbar networks are developed and arrangements yielding minimum number of chips, average delay through the network, and product of number of chips and delay, are presented. A bit slice approach (B = 1) produces the optimum arrangement for the crossbar, while for the banyan the optimum is achieved with multiple bits per module.

Introduction

Over the past few years a variety of physically local, closely coupled multiple processor systems have been proposed (1,2,3,4). One key issue in the design of such systems concerns the communications network used by these multiprocessor systems. Various studies have focused on the functional properties of such networks (i.e., what permutations are possible, what control algorithms are needed, etc.), on their complexity, and to some extent on performance issues (5, 6, 7, 8, 9). In most cases network complexity has been measured by the number of elementary switching components needed by a network of a given size and type, while performance has been determined by the average number of elementary switching components through which a message must pass (i.e. average delay). Recently work has begun on examining complexity and performance questions in the context of VLSI implementation of such interconnection networks. Franklin (10) has compared two networks, crossbar and banyan, operating in a circuit switched mode in terms of their space (area) and time (delay) requirements. The networks were assumed to be implemented as complete modules on a single VLSI chip.

Closer examination of VLSI network implementation problems shows that pin limitations, rather than chip area or logical component limitations, are a major constraint in designing very large interconnection networks. Consider, for instance, a network with N' inputs, M' outputs and with each output being B' bits wide (N'*M'*B'). The number of required pin connections (ignoring power, ground and general control) for a single chip implementation is given by B'(N'+M'). For a square network of size twelve with B'=16, the number of pins required would thus be 384; much larger than common commercially available integrated circuit carriers. Given that pins are typically placed on 100 mil centers along the periphery of the package, the total number of pins is limited mainly by the increase in the physical length of the package. For this pin placement and the 384 pin example, a 19.2 inch dual-in-line package would be required.

In this paper we focus on two of the more obvious solutions to this pin limitation problem. The first approach is to implement a large network (N'*N') requiring many pins as a interconnected set of smaller subnetworks (N*N) where each of the smaller networks can be contained on a single chip in which the chip pin constraints are met.

The second approach is to slice the network so that one creates a set of network planes, each plane handling one or more bits (e.g., B bits) of the B' wide datapath. This is commonly done in memory designs. A potential problem arises in this approach due to the difficulty in synchronizing the multiple planes. This is discussed in reference 11.

The remainder of this paper deals with determining the "best" combination of datapath slice B and chip network size N given:

1. N': An overall network size (N <= N'),
2. B': A data path width (B <= B'),
3. T : An intrachip network type (e.g., the interconnection network implemented within the N*N chip might be a crossbar).
4. T': An interchip network type (e.g., the interconnection network implemented between the N*N chips to achieve the overall N'*N' network might be an Omega network).
5. N_p : The maximum number of pins allowed on a chip.
6. N_k : The number of pins on a chip allocated to power, ground, and control.

"Best" in this context, refers to both chip count and bandwidth of the overall N'*N' network. Figures 1, 2 and 3 illustrate a general N'*N' network, and a possible decomposition of a sample 16*16 network. In the next section basic models

*This work was supported in part by NSF Grant MCS-78-20731 and ONR Contract N0014-80-C-0761.

0190-3918/81/0000/0253$00.75 © 1981 IEEE

for this problem are presented and used to determine the B and N combinations which minimize the total chip count, the overall network delay and an overall performance measure using the product of chip count and time delay.

The Basic Model

The basic model consists of two parts. The first relates to the chip count while the second concerns network time delay. For brevity, only square fully connected networks (i.e. there is a path from each input port to each output port) are considered. Note that certain input/output paths may have a common subpath and this may result in messages being temporarily blocked.

Let us refer to the N*N*B chip as a switch module; a number of these modules will be interconnected to realize the N' network. This paper considers two types of interchip networks (T'): the incremental crossbar, CB, and the banyan BA (12,13,14). While there are many ways of designing a crossbar network (e.g., demultiplexer/multiplexer configuration, switched multiple busses, etc), the incremental crossbar design (Figure 4) can be expanded on a unit basis by adding basic switch modules in a row-column arrangement. This modularity property permits flexible expansion while retaining the nonblocking and full connection properties of the crossbar. A price is paid for these properties in terms of number of switches and pins required on a switch module. While the number of switches required per switch module may not be a serious constraint with VLSI technology, the problem of pin constraints is severe. For the incremental crossbar, the modularity property requires 4NB data pins to implement a N*N*B switch module while the banyan, a blocking network, requires 2NB data pins.

To make global comparisons similar and to eliminate blocking at the switch module level, this paper examines cases in which the switch modules are constructed using an incremental crossbar architecture (T = CB). Two types of module interconnections are examined; the crossbar and the banyan (T' = CB or T' = BA).

Chip Count Model

As illustrated in Figure 4, the number of N*N*B chips required to implement an N'*N'*B' incremental crossbar network is given by:

$$N_{cb} = \lceil B'/B \rceil \lceil N'/N \rceil^{2} \qquad [1]$$

The banyan network is one of the class of blocking networks whose logical component complexity grows as O(N log N) rather than O(N**2). As illustrated in Figure 5, the number of N*N*B chips needed to implement as N'*N'*B' banyan network is given by:

$$N_{ba} = \lceil B'/B \rceil \; \lceil N'/N \rceil \; \lceil \log_{N} N' \rceil \qquad [2]$$

The first term in this expression is the number of bit slices or network planes that are required. The second term represents the number of chips at each level (row), while the third term is the number of levels.

Time Delay Model

A model giving the average time for a signal to propagate through a network must include the time to traverse each of the chips, the time to propagate from chip to chip, and since the bit slice approach separates the bits in a data word, the additional time that is needed to make certain that all the data bits have completed their movement through the N'*N'*B' network.

The average delay associated with a basic switch module will be designated as D_{cb} since these modules have a crossbar construction. Path setup delays (i.e., time to set switches in their desired positions) are not considered here. The delay of a pin driver and associated interconnection wires between modules (i.e. the intermodule delay) is denoted by D_{im}. The intermodule delays for the CB and BA networks are different and will be denoted as D_{imcb} and D_{imba}. Additional synchronization delay introduced by the designer to assure that all data bits have traversed the network will be represented by D_{syncb} and D_{synba}.

For the CB network the average delay can be determined by examining Figure 4. Note that this represents one of $\lceil B'/B \rceil$ planes. Assume that each switch module, implemented on a single chip, represents an N*N CB network. The pin drivers for each module are also located on the chip. For this arrangement the number of modules in an average path is $\lceil N'/N \rceil$ and each intermodule path has the same delay D_{imcb}. Therefore the average network delay D'_{cb} is given by:

$$D'_{cb} = \lceil N'/N \rceil D_{cb} + \lceil N'/N \rceil D_{imcb} + D_{syncb} \qquad [3]$$

Note that a circuit switched design is assumed here with no pipelining between modules.

For the BA network the number of switch modules and the number of intermodule connections is $\log_N N'$. Here, because of the connection topology, the intermodule paths are not constant in length. The average delay, D'_{ba}, through such a network (assuming no delay penalty for blocking) is given by:

$$D'_{ba} = \lceil \log_{N} N' \rceil D_{cb} + \lceil \log_{N} N' \rceil D_{imba} + D_{synba} \qquad [4]$$

Pin Constraints

For a square N*N*B chip with N_k pins allocated to power, ground and control, the pin constraint is given by:

$$N_p >= KBN + N_k \qquad [5]$$

where K = 4 for the CB network and K = 2 for the BA network. The equality will be used since it is advantageous to utilize as many available pins as possible. Two cases may be considered. Case 1 is the situation where the number of data pins is much larger than N_k (i.e., KBN >> N_k) and thus equation 5 becomes:

$$N = \lfloor N_p/KB \rfloor \qquad [6]$$

This is typical of a clocked system where a small number of clock lines are needed to synchronize all the data lines.

Case 2 encompasses the situation where N_k is not neglible and there is a control line overhead associated with the data paths. Assuming that the number of control lines is proportional to the number of ports, N, on an individual chip (i.e. $N_k=QN$ where Q is a constant), N can be expressed as:

$$N = \lfloor N_p/(KB+Q) \rfloor \qquad [7]$$

This would be the appropriate model if network chips communicated with each other in an asynchronous manner and the control line overhead consisted of request/acknowledge pairs (Q = 2).

Chip Count Minimization

For large networks with large datapath widths and chips with a large number of pins, the ceiling and floor functions can be removed from [1] and [2], and [6] and [7]. Then N_{cb} and N_{ba} can be approximated as continuous functions. Assume that all available pins are used and consider Case 1 where N is given by equation 6. Substituting equation 6 with K = 4 and K = 2 respectively into continuous versions of equations 1 and 2 yields:

$$N_{cb1} = 16BB'(N**2)/N_p**2 = K_{cb}B \qquad [8]$$

$$N_{ba1} = \frac{2B'N'\log N'}{N_p(\log N_p - \log 2B)} = \frac{K_{ba}}{\log N_p - \log 2B} \qquad [9]$$

For a given pin constraint N_p, and overall network requirements N' and B', K_{cb} and K_{ba} are constants. Minimizing N_{cb1} and N_{ba1} for this case requires that B be minimized. The smallest datapath width possible is B = 1, hence with this model N should be selected to be N_p/K. This result corresponds to memory chip design where the slice width is generally taken as one bit. Note however, that this was obtained with a continuous approximation to equations 1 and 2; while B = 1 yields a minimum number of chips in most cases, there are situations where other values of B are better. For instance, with a BA network with $N_p = 60$, N' = 128 and B'=16, a B = 1 solution yields $N_{ba1} = 160$, while a B = 2 solution yields $N_{ba1} = 144$.

For case 2 where N_k is not negligible equation 7 is used for N and substituted back into the continuous versions of [1] and [2] to give:

$$N_{cb2} = \frac{(4B+Q)^2B'N'^2}{BN_p^2} = \frac{K_{cb}(4B+Q)^2}{16B} \qquad [10]$$

$$N_{ba2} = \frac{K_{ba}(2B+Q)}{2B(\log N_p - \log(2B+Q))} \qquad [11]$$

The derivatives of N_{cb2} and N_{ba2} with respect to B can now be taken, and the values of B and N which minimize the chip count obtained.

For the case of T' a CB, the number of chips N_{cb2} is minimized when B = Q/4. Thus for a request/acknowledge pair associated with each chip datapath (Q = 2), B would be selected as 1. While this is true for almost all cases considered, the continuous model approximation should be checked when N' is less than 64 or B' is less than 16 (e.g., For N' = 32 $N_p = 75$, Q = 2 and B' = 16; B = 1 yields N_{cb2} = 144: B = 2 yields N_{cb2} = 128).

For the case of T' a BA network, an equation can be derived for obtaining the optimum B and N and indicates that the continuous model does not yield optimum values in many situations. For instance, for $N_p = 90$, N' = 512, Q = 2 and B'= 16, a search procedure working directly with equation 2 gives an optimum B = 4 and yields N_{ba2} = 684. Note that using B = 1 in this case results in N_{ba2} = 1152. This is not unusual, and in most cases ($N_p < 140$) where $Q \geq 2$, a choice of B = 1 will be nonoptimal.

Equations 1 and 2 were solved using optimal values of N and B. and the chip count was obtained as a function of the parameters N_p, N' and network type T'. Figure 6 illustrates how the total number of chips varies as a function of the network size. Plots for two different values of N_p and Q are also given. For a given N', N_p and Q, the BA requires fewer chips than the CB implementation and the curves agree with the observation that the crossbar grows as O(N'**2) while the banyan grows as O(N'LogN'). As expected, increasing Q or N_p requires a larger number of chips for both the banyan and the crossbar. Although not shown explicitly in these graphs, the optimum value of B is 1 for the crossbar ($N_p \geq 64$), while for the banyan the optimum B ranged from 1 to 4 ($N_p \geq 64$).

Network Delay Minimization

Next we determine expressions for the delays, D_{cb}, D_{im}, and D_{syn} and incorporate these into equations 3 and 4 to compute the average delay through the two networks.

Crossbar Network

The value of D_{cb} has been developed by Franklin (10) using NMOS NOR gates for construction of the crossbar module and is given as:

$$D_{cb} = N[2.5mf\tau + \tau(1+2.25\alpha_{cb})] = NA_o \qquad [12]$$

The parameters are defined in Table 1 which also gives some typical values. The equation assumes a circuit switched CB, and uniformly distributed addressing of module output ports. The first term in the brackets represents the delay through an individual switch within a module, while the second term is the delay between switches in a module.

The delay encountered when a signal goes off the chip, propagates along an interconnecting path and enters another switch module is D_{imcb}. A buffer (e.g., a series of inverters) must be included within the switch module to allow the minimum size transistor to drive the module pin and associated load with minimum delay. The buffer delay is determined by the gate capacitance of the minimum size transistor, the number of stages in the buffer, the capacitance of the pin being driven, the capacitance along the interconnecting path, and the capacitance of the receiving module pin. This delay is minimized when exponentially sized cascaded inverters are used (14). The delay in this case is:

$$D_{imcb} = \tau e \log_e \beta_{cb} \qquad [13]$$

where β_{cb} is the ratio of the buffer load capacitance to the buffer input transistor gate capacitance. The transistor gate capacitance, C_g, is the capacitance per unit area times the gate area of the minimum transistor. To determine the load capacitance assume that the driving and receiving pin capacitances are equal and each has a value of C_{pin}. Further postulate that the modules for the CB will be placed on a circuit board and interconnected via printed circuit copper paths. Given the planar topology of the CB, the spacing between modules will generally be less than one inch. Pin capacitance will dominate in this case and $\beta_{cb} = (2C_{pin}+C_{path}) / C_g \simeq 2C_{pin}/C_g$.

The synchronization delay depends upon the specific design technique used to determine that all bits have traversed the network. Assuming selftimed design strategy, a reasonable design practice is to include a tolerance or guard region that is proportional to the average delay time. The average delay can thus be expressed as:

$$D'_{cb} = K_1 \left\lceil N'/N \right\rceil (D_{cb} + D_{imcb}) \qquad [14]$$

where $K_1 = 1 + K_s$, and D_{cb} and D_{imcb} are given in [12] and [13]. Numerical studies have shown that for the CB with Q = 0 the continuous form of [14] usually gives the same results as the discrete form Therefore we shall replace $\lceil N'/N \rceil N$ by N'. Finally, using equation 7 with K = 4 gives:

$$D'_{cb} = K_1 N' [A_o + D_{imcb}\ (4B+Q)/N_p] \qquad [15]$$

To minimize D'_{cb}, 4B + Q should be minimized. With Q = 0, this means that D'_{cb} is minimized when B = 1. Notice that D'_{cb} is directly proportional to N', and decreases to a minimum value as the number of pins N_p increases. Consider next the typical parameter values given in Table 1. For N_p large (i.e. >= 64), Q = 0 and B = 1, the average delay can be approximated as:

$$D'_{cb} \simeq 6.2N' \text{ nsec} \qquad [16]$$

Banyan Network

The average delay through the BA network is given by equation 4. The value of D_{cb} is known from [12] and we assign a value to D_{syn} that is proportional to the average path delay through the network. The only remaining quantity to determine is the value for D_{imba}. The development follows that presented for the CB. In this case however the separation between switch modules in the BA is not constant, and C_{path} will vary according to the banyan level. Since the number of levels required for a specific configuration is not known a priori, the inclusion of a variable for C_{path} complicates the delay computation. The last level has the longest path (S inches) and therefore the maximum capacitance. The capacitance of a typical printed circuit path is approximately 1 pf/inch thus the delay in driving this longest path is:

$$D_{imba} = \tau e \log_e((2C_{pin}+S)/C_g) \qquad [17]$$

By decreasing the pin driver area as the banyan level decreases this value applies to all levels.

The average delay through the banyan network can now be expressed as: [18]

$$D'_{BA} = K_1 \left\lceil \log_N N' \right\rceil [NA_o + \tau e \log_e((2C_{pin}+S)/C_g)]$$

The continuous version of this equation is a poor approximation to the discrete version, thus only the discrete will be used. Using the values from Table 1 the banyan delay can be expressed as:

$$D'_{BA} = 6.17 \left\lceil \log_N N' \right\rceil (N + 1.78) \qquad [19]$$

The discrete relations for the CB and the BA delays were solved using optimal values of N and B, and the delays obtained as a function of parameters N_p, N' and network type T'. The banyan delay is consistently smaller than the crossbar for networks of reasonable size.

Chip Count-Time Product Minimization

The chip count-time product P, can be obtained by multiplying the appropriate equations given previously. Earlier discussion indicated that for reasonable size networks, both chip count and delay were minimized in the CB case with B = 1. Consequently the product is also minimized with this choice (for N' $\geq$ 64, B' $\geq$ 16).

For the BA, the situation is more complex and a computer search for the optimum B and N values must undertaken. Consider the case of Q = 0 and N' = 512 Table 2 shows the values of N,B which optimize the number of chips, the delay, and chip count-time product. The B and N values required for minimizing P fall between those needed for minimization of the chip count and delay measures by themselves. The count minimization is achieved by attempting to place as large a network as possible on a given chip. Delay minimization is achieved by balancing the delays associated with the module network and the delays associated with increasing the number of levels in the overall network. In this case placing as large a network on a chip as possible is not the best strategy from a delay point of view. Note that this analysis does not consider delays associated with network blocking which can have a significant effect in a saturated network.

Values for N and B which minimized P were obtained for both network types over a range of N', N_p and Q values (Figure 7). As expected P increases with increasing N' and increasing Q, and decreases with increasing N_p. Once again the banyan does better than the crossbar on this overall performance measure.

Summary and Conclusions

This paper concerned the design of multiple processor interconnection networks. Models for both the banyan and crossbar networks (T') were developed and arrangements yielding minimum: number of chips, average delay through the network, and product of number of chips and delay, were presented. The results show that for the crossbar a bit slice approach (B = 1) produces the optimum arrangement, while for the banyan the optimum is achieved with multiple bits per module. The impact of the number of control lines on chip count, delay and product were also modelled.

The analysis presented made a number of assumptions whose effects are being further investigated. In particular the role of blocking in the banyan case, the potential gain which would accrue from a pipelined design, and the problem of synchronization between network planes is being studied.

Parameter	Symbol	Units	Typical Value
minimum feature size	λ_{min}	um	3
minimum gate area	$A_{min}=4\lambda^2$	$(um)^2$	36
gate capacitance	C_g	pf	$1.4*10^{-2}$
switch module pin cap.	C_{pin}	pf	5
transit time	τ	nsec	0.5
NOR gate logic levels per crossgate	m	--	2
NOR gate fanout	f	--	2
Metal path cap. to transistor gate cap. ratio (switch module)	α_{CB}	--	0.1
guard region multiplier	K_s	--	0.1
printed circuit path cap.	C_{path}	pf	lpf/inch
length of longest BA path	S	inches	12

TABLE 1: TIME DELAY PARAMETERS

	N	B	CHIP COUNT	DELAY (nsec)	CDP $(*10^3)$
N_p = 60					
COUNT MINIMIZATION	30	1	576	392	226
DELAY MINIMIZATION	5	6	1236	168	207
PRODUCT MINIMIZATION	10	3	936	218	204
N_p = 90					
COUNT MINIMIZATION	45	1	384	578	222
DELAY MINIMIZATION	5	8	824	168	138
PRODUCT MINIMIZATION	11	4	564	237	133
N_p = 120					
COUNT MINIMIZATION	60	1	288	763	220
DELAY MINIMIZATION	5	11	824	168	138
PRODUCT MINIMIZATION	10	6	468	218	102

TABLE 2: BANYAN NETWORK MINIMIZATION RESULTS
(N' = 512, Q = 0, B' = 16)
(CDP: Count Delay Product)

References

1. Swan, R.J., et. al., "Cm*-A Modular Multi-Microprocessor", Proc. NCC (1977).
2. Dennis, J.B., and Misunas, D.P., "A preliminary architecture for a basic data-flow processor", Proc. 2nd Ann. Symp. on Comp. Arch. (Dec. 1974).
3. Sejnowski, M.C., et. al., "An overview of the Texas Reconfigurable Computer", Proc. AFIPS Nat. Comp. Conf. (1980).
4. Sullivan, H. and Baskow, T.R., "A Large Scale, Homogeneous, Fully Distributed Parallel Machine I", Proc. 4th Ann. Symp. on Computer Architecture (March 1977).
5. Benes, Y.E., Mathematical Theory of Connecting Networks and Telephone Traffic, Academic Press, New York (1965).
6. Siegel, H.J.; McMillen, R.J., and Mueller, P.T., Jr., "A Survey of interconnection methods for reconfigurable parallel processing systems", Proc. 1979 Nat. Comp. Conf. (June 1979).
7. Anderson, G.A., and Jensen, E.D., "Computer interconnection structures: taxonomy, characteristics, and examples", ACM Comp. Sur. 7, 4 (Dec. 1975).
8. Thurber, K.J., "Interconnection networks - a survey and assessment", Nat. Comp. Conf. (May 1974).
9. Franklin, M.A.; Kahn, S.A. and Stucki, M.J., "Design Issues in the Development of a Modular Multiprocessor Communications Network", Proc. of the Annual Symposium on Comp. Architecture (April 1979).
10. Franklin, M.A., "VLSI Performance Comparison of Banyan and Crossbar Communications Networks", IEEE Trans. on Comp. C-30,4 (April 1981).
11. Franklin, M.A. and Wann, D.F., "Word Inconsistency in Partitioned VLSI Interconnection Networks", Center for Computer Systems Design, Washington University., St. Louis, Internal Report #81-101 (May 1981).
12. Goke, L.R. and Lipoviski, G.J., "Banyan Networks for Partitioning Multiprocessor Systems", The First Ann. Symp. on Comp. Arch., University of Florida, Gainesville, Florida (1973).
13. Lawrie, D.H., "Access and Alignment of Data in an Array Processor", IEEE Trans. on Comp., Vol. C-24, No. 12 (Dec. 1975).
14. Pease, M.C., "The indirect binary n-cube microprocessor array", IEEE Trans. Comp. C-26,5 (May 1977).
15. Mead, C. and Conway, L., Introduction to VLSI Systems, Addison-Wesley Pub. Co. Reading, MA (1980).

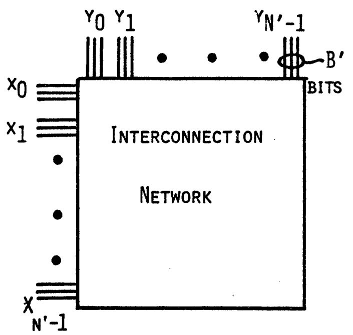


FIGURE 1: AN N'*N' NETWORK

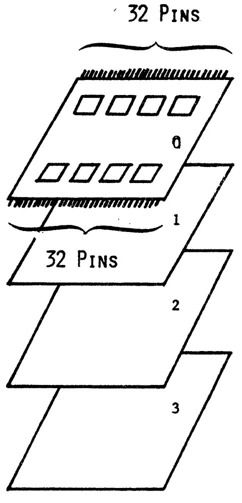


FIGURE 3: A FOUR PLANE 16*16*8 NETWORK

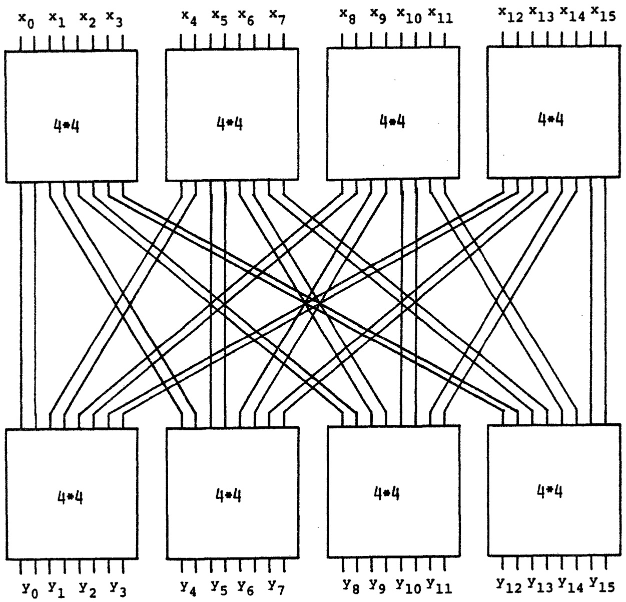

x_0 x_1 x_2 x_3
x_4 x_5 x_6 x_7
x_8 x_9 x_{10} x_{11}
x_{12} x_{13} x_{14} x_{15}
4*4
y_0 y_1 y_2 y_3
y_4 y_5 y_6 y_7
y_8 y_9 y_{10} y_{11}
y_{12} y_{13} y_{14} y_{15}

FIGURE 2: DECOMPOSITION OF A 16*16*8 NETWORK USING 4*4*2 NETWORK CHIPS (ONE PLANE OF FOUR SHOWN. CONTROL AND POWER PINS NOT SHOWN).

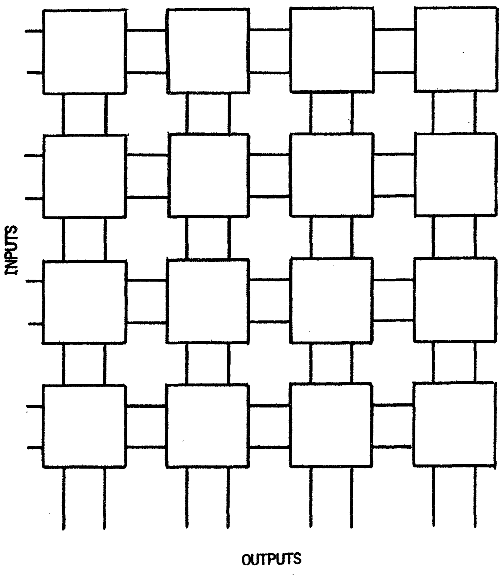


FIGURE 4: AN 8*8*1 INCREMENTAL CROSSBAR COMPOSED OF 2*2*1 MODULES

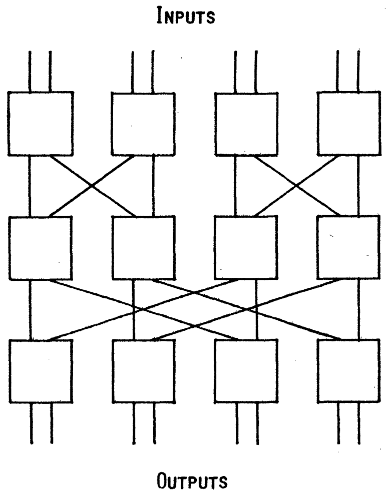


FIGURE 5: AN 8*8*1 NETWORK COMPOSED OF 2*2*1 CHIP COMPONENTS ARRANGED IN A BANYAN CONFIGURATION

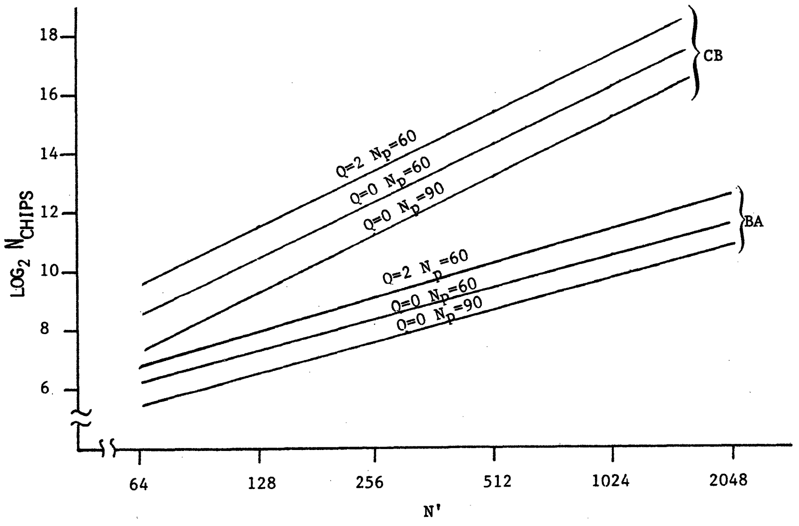


FIGURE 6: NUMBER OF CHIPS, N_{CHIPS}, VERSUS NETWORK SIZE N'

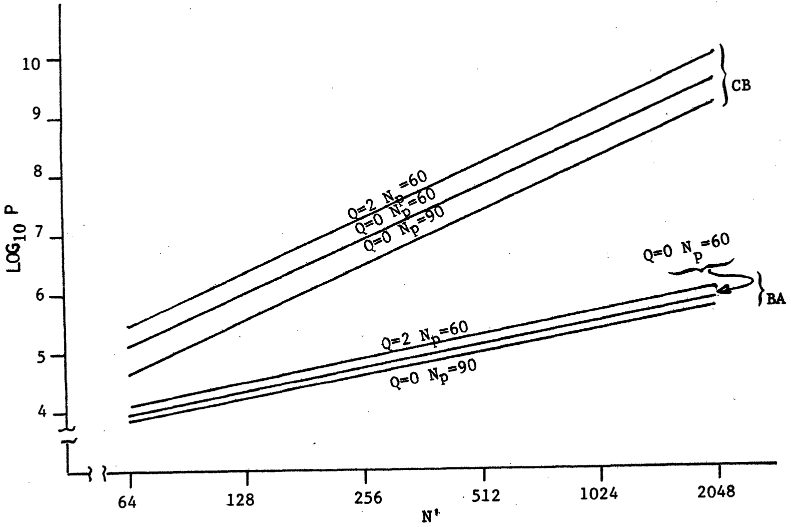


FIGURE 7: PERFORMANCE MEASURE, P, VERSUS NETWORK SIZE N'

LINEAR RECURRENCE SYSTEMS FOR VLSI: THE CONFIGURABLE, HIGHLY PARALLEL APPROACH.

Dennis Gannon [1]
Lawrence Snyder [2]

Department of Computer Sciences, Purdue University
West Lafayette, Indiana 47907.

SUMMARY

1. CHiP Overview.

Recently, much VLSI based parallel processing research has focused on algorithmically specialized processors, highly parallel computers with fixed, etched-in-silicon interconnection structures tailored to a particular algorithm or small class of algorithms [4,7,11] and many others. While these processors are highly successful at exploiting locality, the inflexibility of the rigid interconnection structure is a liability for problems outside their area of specialization.

Among the more notable attempts to study a silicon based general purpose parallel computing system is the Cube Connected Cycle work of Preparata and Vuillemin [8] and designs that may be based on the optimal area embeddings of the shuffle-exchange graph [5]. By providing an interconnection network of nearly universal permuting power these systems host a number of important algorithms with the property that if the cost of signal propagation along long lines is ignored the asymptotic complexity of the program is the same as that of the algorithm with all communication costs eliminated. The price for such a universal network is paid in two ways: the surface area of silicon grows rapidly, $O(n^2 / log^2 n)$, for an n processor device, and the lack of local uniformity makes it difficult to partition the system into a small number of easily packaged units.

A second approach to general purpose parallel computing is being followed in the design of a Configurable, Highly Parallel (CHiP) computer [9]. The motivating goals are to exploit locality (i.e. permitting easy implementation of pipelined systolic processes as well as simple decomposability into reproducible components) and, at the same time, provide algorithmic flexiblility.

The CHiP architecture has the following set of components.

1. A regular array of simple microprocessing elements (PEs) each with a small but reasonable amount of local memory.
2. A front end controller.
3. A switch lattice.

The switch lattice is a regular structure composed of progammable switches connected by data paths (see Figure 1 for an example). The PEs are connected to the lattice at regular intervals. The difference between programming the CHiP computer and a machine based on a universal connection network lies in the programming of the switches. With the network the programmer seeks to decompose the data flow of his algorithm into a sequence of stages realizable by the network (i.e. a sequence of convolutions or shuffles). With the CHiP array the idea is to decompose the data flow graph into a sequence of simple subgraphs that can be embedded into the switch lattice in a manner that optimizes some parameter such as locality.

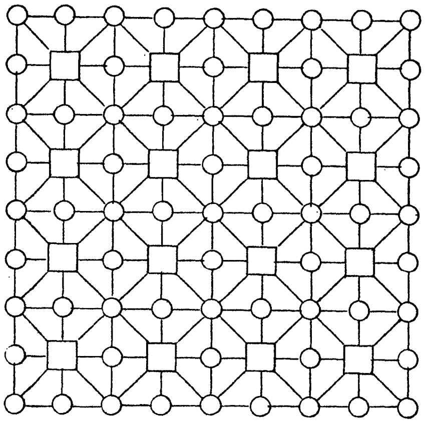


Figure 1. CHiP lattice. PEs shown as squares, switches as circles.

Each switch is equipped with a small amount of local memory capable of storing several local configurations. A particular configuration setting enables a switch to connect two or more of its incident data paths directly and *statically*[3]. The controller can realize a new global interconnection by a system of signal broadcasts directing each switch to select the next setting stored in its local memory.

2. A Programming Example: Linear Recurrences.

Linear recurrences arise so frequently as subprocesses of other algorithms that they are an important benchmark for exhibiting the speed and versatility of a multiprocessor. Following Sameh and Chen [1], and Kuck [6] we observe [12] that any linear recurrence of order q can be put in the form

$$X_i = C_i + B_i X_{i-1} \quad i = 2, \ldots, n$$

given B_i, C_i, $i = 1,\ldots,n$ and X_1 where each B_i is a q by q matrix and each C_i and X_i are column vectors of size q. Gajski [3] has observed that if we define a semigroup under the composition rule

$$(C_i, B_i) * (C_j, B_j) = (C_i + B_i C_j, B_i B_j)$$

then the recurrence relation problem is equivalent to

[1] Research supporeted by NSF grant MCS-8109512.

[2] Supported in part by the Office of Navel Research contracts N00014-80-K-0816 and N00014-81-K-0360 Spec. Res. Op. Task SRO-100.

[3] That is, the lattice uses circuit switching rather than packet switching.

first computing the products

$$D_{ki} = \prod_{s=i}^{k}(C_s, B_s) \quad i = 2,\ldots,n \quad k < i$$

and then evaluating

$$X_i = \overline{C}_i + \overline{B}_i X_i \quad i = 2,\ldots,n$$

where $(\overline{C}_i, \overline{B}_i) = D_{1i}$. With n a power of 2 the standard computational flow graph of the required D_{ij} takes the form shown if Figure 2.

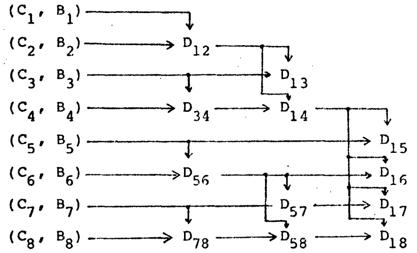


Figure 2. Recurrence Computation Graph.

Because standard systolic matrix multiplication algorithms are easy to implement on a mesh structure embedded in the CHiP, the programming for $q > 1$ is a straight forward generalization of the first order case described below. While it is clear that the flow graph above can be embedded directly into a CHiP lattice of dimension n by $log(n)$, this approach has two drawbacks: (1) for the evaluation of a single recurrence relation the PE utilization is low and (2) the last stage requires a data broadcast along a channel of length that grows linearly in n.

A more careful programming of the switch lattice can improve this situation. Collapse the flow graph by rows and assign each row of the computation to a single PE. In this approach the program consists of $log(n)$ stages where at stage i processor P_{j_k} with $j_k = 2^{i-1} + k2^i$, $k = 1,\ldots,n/2^i$ will broadcast $D_{\bullet j_k}$ to processors P_{j_k+t} $t = 1,\ldots,2^i$. The main problem is to assign the logical processors to the physical PEs in a $n^{1/2}$ by $n^{1/2}$ CHiP lattice. While many solutions are possible one method that makes good use of locality and has a straight forward control structure is shown in Figure 3. The processors are numbered in the "shuffled-row-major" indexing of Thompson and Kung [11].

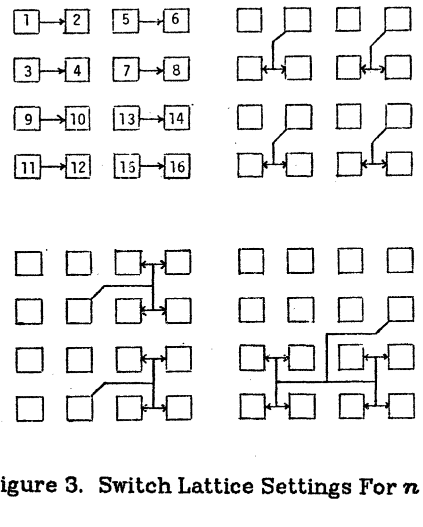


Figure 3. Switch Lattice Settings For n = 16.

The broadcasts in each stage are completed with "hyper-H" fan-out trees in which no edge is of length greater than $n^{1/2}/2$. If one assumes that the propogation time of an nMOS signal is linear in the distance traveled then the time to complete the recurrence computations will be of the form $clog(n) + dn^{1/2}$ for constants c and d with $d \ll c$. In the special case that the recurrence relation represents binary addition the lower bounds of Chazelle and Monier [2] show that the scheme above is optimal in area and time. Furthermore, if initially the data resides outside the CHiP array then the I/O complexity is bounded by the $n^{1/2}$ perimeter size. Consequently, one has an asymptotic match between computation and I/O complexity.

3. References

[1] S. C. Chen, D. J. Kuck, and A. H. Sameh, "Practical Parallel Band Triangular System Solvers", *ACM Trans. on Math Software*, Sept 1979, pp. 270-277.

[2] B. Chazelle and L. Monier, "A Model of Computation for VLSI with Related Complexity Results", *STOC*, ACM, Milwaukee, 1981, pp.318-325.

[3] D. D. Gajski, "Solving Banded Triangular Systems on Pipelined Machines", Proceedings of the Int'l Conf. on Parallel Computation, IEEE, 1979.

[4] L. J. Guibas, H. T. Kung, C. D. Thompson, "Direct VLSI Implementation of Combinatorial Algorithms", Proc. Conference on VLSI Architecture, Design, and Fabrication, Calif. Inst. of Techn., Jan 1979.

[5] D. Kleitman, F. T. Leighton, M. Lepley, G. L. Miller, "New Layouts for the Shuffle-Exchange Graph", *STOC*, ACM, Milwaukee 1981, pp.278-292.

[6] D. J. Kuck, *The Structure of Computers and Computations*, Vol. 1, John Wiley & Sons, Inc., New York, 1978.

[7] H. T. Kung, C. E. Leiserson, "Algorithms for VLSI Processor Arrays", C. Mead and L. Conway, *Introduction to VLSI Sytems*, Addison-Wesley, Reading, Ma., (1980) pp. 271-292.

[8] F. P. Preparata, J. Vuillemin, "The Cube-Connected-Cycles: A Versatile Network For Parallel Computation", Proceedings of the 20th. Annual Conference on the Foundations of Computer Science, pp.140-147, 1979.

[9] L. Snyder, *Introduction to the Configurable, Highly Parallel Computer*, Dept. of Computer Sciences Purdue University, Technical Report CSD-TR-351, May 1981.

[10] L. Snyder, D. Gannon, "Linear Recurrence Systems for VLSI: The Configurable, Highly Parallel Approach.", *in preparation, 1981.*

[11] C. D. Thompson and H. T. Kung, "Sorting on a Mesh-Connected Parallel Computer", *CACM*, v.20, no 4, pp.263-270, 1977.

EMBEDDING A TREE IN THE NEAREST NEIGHBOR ARRAY

Amar Mukhopadhyay and Ratan K. Guha
Department of Computer Science
University of Central Florida
Orlando, Florida 32816

Introduction

The binary tree is an important interconnection structure for hierarchically organized parallel machines. With the advent of VLSI, considerable interest has been generated in dirct hardware implementation of a tree. However, the parameters for VLSI are quite different from those used in traditional design. Since a large amount of ciruitry must be packed in a chip, the total area of the chip is an important design parameter. The cost of communication rather than computation is also a major factor in VLSI design. This cost is reflected not only in the area occupied by the interconnection paths within the chip but also by the requirement of providing additional communication channels with the external world via the input/output pins, which cannot be arbitrarily large for VLSI chips. For example, in the so-called "H" layout (Mead and Rem, 1979) of a tree, a large fraction of the leaf nodes are within the chip and communication to the external world needs additional area. This also increases the signal propagation time between any two nodes to $O(\sqrt{n})$ rather than being the natural $O(\log n)$ value for a tree, since the nodes at higher levels of the tree use longer and longer wires for interconnection. Another important design parameter for VLSI design is known as the "regularization factor" (Lattin, 1979) which measures the regularity of the embedded structures in the chip. Intuitively, it means that the number of "templates" of interconnection and "cells" for computation should be kept as small as possible in defining the layout of the chip.

In this paper we consider the problem of laying out a complete binary tree with n leaves on a chip and show that for practical size trees in which the nodes correspond to processors and the connection between two nodes correspond to communication paths between the processors, the proposed solutions represent the best known area efficient layout with maximum availability of the leaf and root nodes at the boundary and low communication cost. We show that this could be done on a nearest neighbor array with three basic connection templates. The problem of laying out a tree on a plane of minimal area has recently been researched extensively [Leiserson (1979), Valiant (1981), Browning (1980), Locanthi (1980), Brent and Kung (1979) and Krishnan (1981)].

The Model of the Silicon Surface

The proposed model of the two dimensional surface has the following attributes: first, each processor takes a unit area on the surface; second the processors are connected in the nearest neighbor array using the interconnection templates, as shown in Figure 1(a), (b) and (c) and their rotations by 90°, 180° and 270°, where P_0 is the parent processor and P_1 and P_2 are its sons. In templates (a) and (b), the parent processor is directly connected to both its sons and we will say that the communication between them will be provided by implicit wiring. The necessary area devoted for this purpose is some fraction of the unit area located at the interface. In template (c), the P_0 is directly connected to P_2 which in turn is directly connected to P_1. P_0 and P_1 are connected to each other via P_2.

Each processor has two or three sets of internal and/or external ports. Any information sent to P_2 by P_0 must be retransmitted to P_1 if the information was directed to both P_1 and P_2 or P_1 only. Any information sent by P_1 to P_2 is always transmitted to P_0 by adding a bit to it so that P_0 knows that P_1 is transmitting to P_0. When P_0 is transmitting information to its sons, it should indicate which son is supposed to receive it. When P_0 receives information from one of its sons, it should check who sent the information. Finally, if a wire is used to communicate between processors, the width of the wire will be $1/f$, when $f \geq 1$. Such wiring will be called explicit wiring, as shown in Figure 2 between processors X and Y.

The model differs from the previously studied models in distinguishing between the explicit and implicit wiring and also in not assuming that a wire has unit width. The communication between P_0 and P_1 in template (c) requires more time compared to time taken for direct communication, which has to be taken into account in synchronizing the computations performed by the tree. The implementation of the necessary protocols seems straightforward in terms of both additional hardware and software.

An alternate communication geometry will consist of 8-neighbor array interconnection in which each processor can directly communicate with four nearest neighbors and four nearest diagonal neighbors, as shown in Figure 7. The regions shaded at the boundaries of the cells denote the areas devoted to build communication channels between cells.

The "Ideal" Tree Layout

In this section, we prove a result of academic interest which apparently contradicts the result obtained by Brent and Kung (1979). Assume that each processor needs the same area independent of its shape and that the communication between the parent node and its sons takes place by implicit wiring. Then, a binary tree can be laid out in annular zones of an expanding circle or square, as shown in Figures 3(a) and 3(b), respectively. Let r_i denote the radius of the ith circle or the side of the ith square. Then, obviously, we have the relation

$$r_i^2 - r_{i-1}^2 = 2(r_{i-1}^2 - r_{i-2}^2)$$

Thus,

$$r_i = \sqrt{2^i - 1}, \quad i = 1, 2, \ldots$$

Thus, a k-level tree with $n = 2^k - 1$ nodes can be laid out in an area of exactly n units with all the leaf nodes being accessible at the boundary. With increasing n, the boundary processors have

0190-3918/81/0000/0261$00.75 © 1981 IEEE

thinner and more elongated shapes compared to those near the center. This inhomogeneity in shape makes the scheme practically unrealizable and constitutes the key point of difference between our result and that obtained by Brent and Kung where the "aspect-ratio" of each processor is assumed to be 1.

The Proposed Layout Scheme

Even if we can live with the arbitrary aspect ratio of the "ideal layout," the realities of pin limitations will restrict layouts of trees beyond depth $k = 9$ or $n = 511$ in the most optimistic case. Current packaging techniques allow a maximum of about 120 pins on a chip. A factor of 4 increase puts an upper bound of about 500 pins on a chip. Thus, if we assume that all communication to a processor chip should take place via a single pin, serial port in a "message-switching" mode, the maximum size of the tree will be limited to $n = 511$. However, if accessibility to leaf nodes is not a requirement, bigger size trees could be laid out by restricting the communication through the root node with the penalty of slow communication.

The layout algorithm will consist of special cases for $n = 3$, 7, 16, 31 and 63 and a general algorithm for $n \geq 127$. The layouts for $n = 3, 7, 16$ and 31 are shown in Figure 4. Note the trees can be laid out using only implicit wiring and all the leaf nodes and the root node are accessible at the boundary. For $n = 63$, the tree can be laid out in almost an implicit wired form, as shown in Figure 5, which needs 16f explicit wiring and an additional 24f units of area over the densely packed layout, with all leaf nodes and the root node made accessible at the boundary. For $n = 127, 255$ and 511 above, the floor plans are obtained by following the general layout algorithm as described below and illustrated in Figure 6 for the tree WXYZ with $n_k = 2^k-1$ from the pair of trees ABCD and EFGH, each with $n_{k-1} = 2^{k-1}-1$ which will form the two subtrees of the root r of the tree. Assume that each of the sides AB, CD, EF and GH contains C_{k-1} external connections; each of AD and EH contains r_{k-1} external connections; the external conections on the sides BC and FG include connection for the root and contain $r_{k-1}+1$ external connections and are denoted by r'_{k-1}.

1. The sides BC and FG of the subtrees are aligned so that the connection to their roots can be connected to the root node r of the tree.

2. The connection of r to the roots of the subtrees partitions the external connections at BC and FG into two classes, each having an equal number $r_{k-1}/2$ connections: those that can be routed upwards for external connection and those that can be directed downward for external connection. The connection to the root node r can be brought out either to downward or upward direction.

In the resulting tree WXYZ, the side YZ contains the external connections for the root r. The external connections for XY and WZ are given by

$$C_k = r_{k-1}$$

The external connections for WX and YZ are respectively given by

$$r_k = 2C_{k-1}+r_{k-1}$$

$$r'_k = 2C_{k-1}+r_{k-1}+1$$

If the length of a side of the subtrees (viz. length of BC) is h, the total area A of the layout can be expressed as

$$A(2^k-1) = 2A(s^{k-1}-1)+hf(r_{k-1}+1)$$

and total average length of explicit wiring W can be expressed as

$$W(2^k-1) = 2W(2^{k-1}-1)+r_{k-1/2}(fr_{k-1}+h+2f)+h/2$$

A comparison of the proposed layout with those obtained by "H" method or Krishnan's method in terms of total area, explicit wiring cost, signal propagation time and accessibility will be included in a detailed paper under preparation.

Acknowledgement

The work is sponsored by a National Science Foundation Grant MCS-8005096.

References

Brent, R.P., and Kung, H.T. "On the Area of Binary Tree Layouts." Technical Report TR-CS-79-07, Department of Computer Science, Australian National University, July 1979. (Also published as a technical report of Carnegie-Mellon Univ.)

Browning, S.A. "The Tree Machine: A Highly Concurrent Computing Environment." Ph.D. Dissertation, Computer Science Dept., California Institute of Technology, 1980.

Krishnan, M.S. "A Structured Approach to VLSI Layout Design" presented at the 2nd Cal Tech conference on Very Large Scale Integration Technology, January 1981.

Lattin, W.W. "VLSI Design Methodology: The Problems of the 80's for Microprocessor Designs." Proc. 1st Cal Tech Conference on VLSI, Jan. 79.

Leiserson, C.E. "Area-Efficient Graph Layouts." Dept. of Computer Science Technical Report, Carnegie-Mellon University, August 1979.

Locanthi, B.N. "The Homogeneous Machine." Ph.D. Thesis, California Institute of Technology, January 1980.

Mead, C.A., and Rem, A. "Cost and Performance of VLSI Computing Structure." IEEE Journal of Solid State Circuits, SC-14, 2, April 1979: 455-462.

Valiant, L.G. "Universability Consideration in VLSI Circuits." IEEE Transcript on Computers. C-30, No. 2, February 1981: 135.

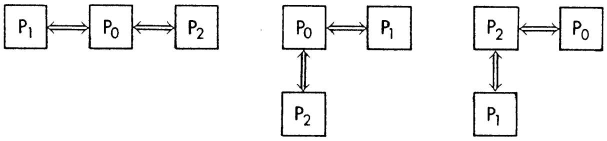


Figure 1

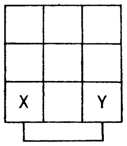


Figure 2

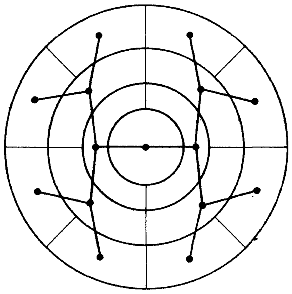


Figure 3 (a)

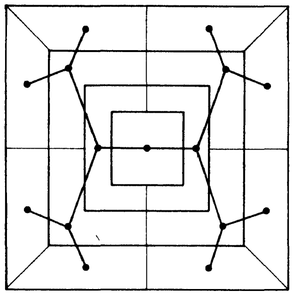


Figure 3 (b)

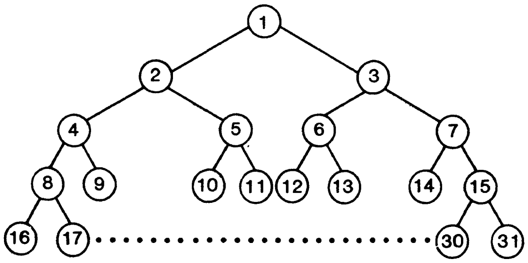


Figure 4 (a)

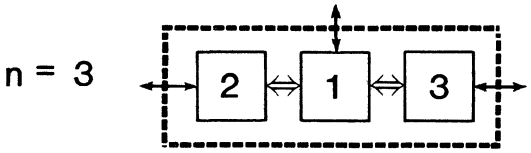


Figure 4 (b)

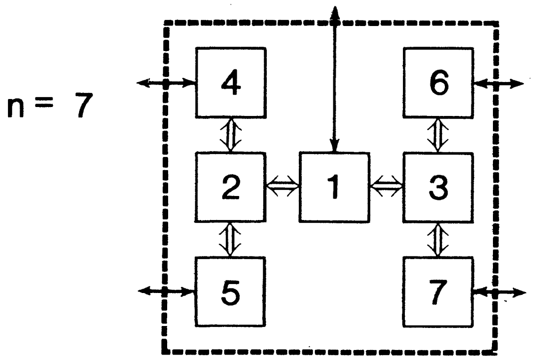


Figure 4 (c)

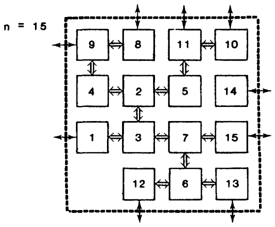


Figure 4 (d)

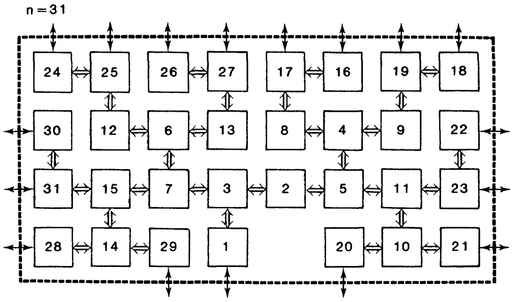


Figure 4 (e)

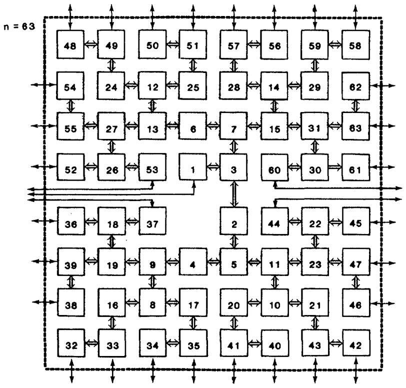


Figure 5

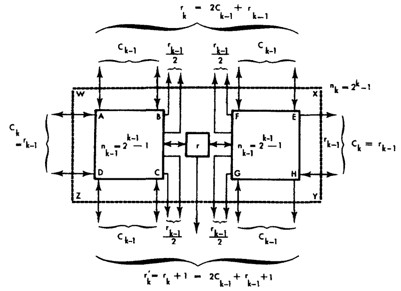


Figure 6

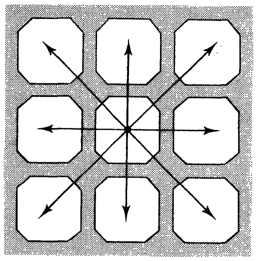


Figure 7

A Constructive Approach to Fault Tolerance in VLSI-Based Systems

Steven E. Butner

Center for Reliable Computing
Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

Summary

As technology advances, the trend has developed not only for more functionality per system but also for more hardware. Demands for higher levels of performance and throughput are also present. These demands are pushing hard on centralized von Neumann computer architectures which are already optimized to near their ultimate physical limits. Solutions in such architectures take the form of increasingly exotic high-technology components and structures. This makes the incorporation of redundant elements necessary for enhanced reliability ever more costly.

As the quantity of circuitry per system increases, the probability of failure also rises. Use of newly-developed materials and devices (e.g. GaAs, Josephson effects) as well as the continuing micro-miniturization of existing technologies is making the typical gate less reliable. In order to maintain acceptable levels of reliability, major computer suppliers are designing redundancy into their equipment. For example, the IBM 370/168 uses parity-prediction in the adder, as well as parity protection on all data paths. Error detection and correction circuitry makes up over 15% of the 370/168. Because of the considerable cost of this additional circuitry, it is clear that overall system reliability has become a very important issue.

Thus, two opposing forces are acting on computer designers. The demands for ever higher performance and functionality are causing more exotic and expensive circuitry. At the same time, the use of more circuits per system is causing overall system reliability to decrease. Centralized architectures are being forced into becoming either less reliable or more costly. It is now time to consider other solutions.

There are two potential avenues of approach to the problem, use of non-centralized architectures and adaptation of reliability enhancement techniques. This paper explores the issues of a combined approach using a distributed architecture and a modified fault-masking method. The objective is to find a realistic, effective solution to the conflicting design demands while retaining acceptable levels of reliability and cost.

As an alternative to the centralization-induced bottlenecks of von Neumann machines and a possible avoidance of difficulties with SIMD and MIMD forms, an architecture known as data flow was introduced [7, 3, 5]. The data flow model of computation utilizes a directed graph to depict the transformations and dependencies of the user program. It features an activation-by-availability execution model which can exploit virtually all concurrency in the user problem. The architecture lends itself to a totally distributed implementation with many copies of the basic processing elements sharing the load. As such, each individual element can be *much simpler*. Performance is no longer the single dominating design issue since additional throughput can be achieved at the system level by adding more parallel units.

Such an architecture provides exactly the sort of relief that is needed for designing reliable, yet cost-effective digital systems. By solving the performance problem at the architecture level, we make room for addressing the issues of reliability at the package level. Based on ideas from data flow, we offer a design style that can be used to construct reliable and practical subsystems.

There are several effective techniques for enhancing the reliability of digital systems . A commonly-used fault-masking technique is triple modular redundancy [8]. Because of costly triplication, TMR (as depicted in Figure 1) is normally too expensive to consider for most applications. By using a distributed architecture, we can consider adaptations of this fault-masking technique which can be practical and cost-effective even for modest reliability requirements.

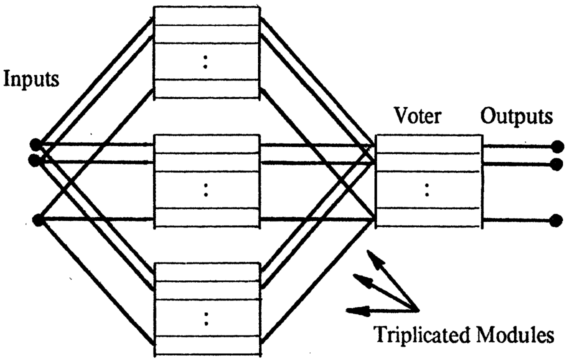


Figure 1: Triple Modular Redundancy (TMR) of a Byte-Sliced Module

In this work we use TMR, but make a basic trade-off of time for space. We do not require triplication of all circuitry. Rather, we advocate slicing the simplex form of the system to create *n* identical byte-wide slices. Triple redundancy is then achieved by multiplexing inputs to the slices *in time* (as shown in Figure 2). The price paid for our trade-off is up to a factor of 3 in time. We remark, however, that the presence of asynchronous portions of a design may allow a significant fraction of this time penalty to be overlapped. Thereby, we get the full fault-masking capability of TMR without the high cost of triplication. The majority of the system redundancy is in the time domain.

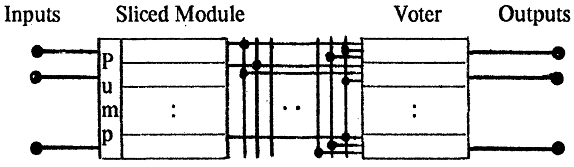


Figure 2: TTR Implementation

Because we have eliminated physical triplication of the module, there is considerably less equipment in the system (as compared with TMR). Less equipment means less circuitry to fail. Because they are simpler, the sliced circuits should individually be more reliable. We show in [2] that the reliability and

0190-3918/81/0000/0264$00.75 © 1981 IEEE

availability of our fault-masking scheme are potentially superior to those of TMR.

In [2] we define several basic elements and rules of composition for constructing a fault-tolerant digital subsystem. Throughout this work, as in TMR systems, a single fault model is assumed. The basic building blocks are slice elements (with arbitrary unidirectional carries), transformation elements, and memories. The subject of fault tolerant memory design is not addressed. Use of a Hamming SECDED code or equivalent is assumed.

For **sliced (S-)elements** we use triple time redundancy, TTR, a generalization of a method used in [6]. In three successive TTR steps, a *pump* circuit presents sets of inputs to each slice. Each set is rotated by one byte width. Thus, a slice produces its own output and that of the two immediately previous slices. Use of pipelining allows overlap of some TTR overhead. A sequential majority voter element is used at the end of pipelines to restore correctness in the presence of arbitrary single slice failures. Specialized elements for subsetting, splitting, joining, and (de)multiplexing S-element streams are defined in [2]. Some of these elements perform necessary rearrangements of outputs during TTR phases φ_2 and φ_3.

For so-called **transformation (T-)elements** (i.e. unsliceable, full-width functions), the problem of preserving correctness in the presence of single faults is somewhat more difficult. It has been shown [1] that by utilizing the regular structure of PLAs *concurrent error detection* can be accomplished. This immediate error detection capability, (along with simple duplication) can be used to render a T-element single fault-tolerant. Such elements form the basis for reliable finite state machines.

A fault-tolerant clock scheme [4] is used in our system. In this scheme three independent oscillators are voted by three majority voters. The voter outputs are fed back to the oscillators for synchronization and also are distributed to interleaved slices so that a failure in a voter will be maskable via our normal slice voting methods.

Since some specialized S-elements must recognize the TTR phase currently active, we provide paths in all elements for *clock tokens*. The presence of a logical one at a pipeline stage and particular clock token line specifies the phase currently active in that element. This technique has built-in redundancy which can be monitored if desired, i.e. at most one of the three token lines may be asserted in a given S-element at any instant.

The series of pipelined S-elements between a pump element and a voter is called, inclusively, a *segment*. Each segment is strictly synchronous with flow controlled by the voter module. Each voter has an ENABLE line by which its successor controls the outflow of data. Whenever the successor is not ready the ENABLE line is held *false*. This causes the voter to continue sequencing until φ_3 and then stop the pipeline. The signal to stop the pipeline is sent back to the pipeline's predecessor element as its ENABLE line. Thus, data in a segment can "pack up" when the voter outflow becomes blocked. The travelling clock tokens keep track of logical phase in our system.

The pipeline control in splitters, join elements, and (de)multiplexers is straightforward, e.g. the predecessor ENABLE line in a splitter (a one-to-many structure) is the result of an AND of its incoming predecessor ENABLE lines. Details for all element types and a complete design example are presented in [2].

The example subsystem is a simplified processing element for the cell block data flow architecture [5]. A four word instruction packet in TTR form is assumed for input. Two result packets are produced. The design is a multi-pipeline aggregate comfortably supporting instruction packet rates in excess of 1 MIP. The *time* limitation in our example is the speed achievable in the packet-transferring S-elements. A crude estimate of the *space* overhead as compared with a non-fault-tolerant design is 200%. The timing and packaging issues are studied and presented in [2]. Due to lack of space, we refer the reader to the report.

A computation of the reliability of our fault-masking technique has been performed. Figure 3 gives comparative plots of reliability versus normalized time for simplex, TMR, and TTR systems. The computation is based upon a constant module failure rate (Poisson) model where $p = e^{-\lambda t}$. For *n* identical, independent slices, this corresponds to a *slice* failure rate of λ/n. Since the slices are roughly twice as complex as an unsliced simplex system, we use $p = e^{-2\lambda t/n}$.

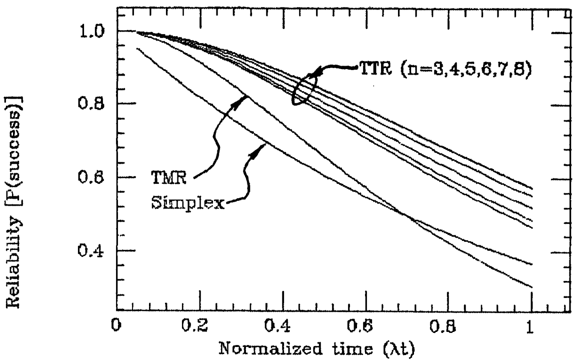


Figure 3: Comparative Reliability of TTR, TMR and Simplex

The technique compares favorably with TMR systems, both in terms of cost and reliability enhancement. As in any fault tolerant system, there are critical regions upon which the reliability of the entire subsystem depend. In our technique, these regions are the storage nodes of voters and the parallel portions of the interconnect. These represent a small portion of the overall chip area and can be rendered more reliable through the use of conservative design layout rules.

This work was supported in part by the National Science Foundation under grant MCS-7904864. Advice given by Professor E. J. McCluskey is gratefully acknowledged. Discussions with D. Hao, R. K. Iyer, A. Kanuma, and J. Khakbaz were also helpful.

References

[1] Avizienis, A. & Wang, S. L., "The Design of Totally Self-Checking Circuits Using Programmable Logic Arrays", *Proceedings of FTCS-9* (June 1979) pp.173-180

[2] Butner, S. E., "Triple Time Redundancy, Fault-Masking in Byte-Sliced Systems", Dept of Electrical Engr., Stanford, TR #CRC-81-2, (June 1981).

[3] Chamberlin, D. D., "Parallel Implementation of a Single Assignment Language", PhD th., Stanford, (Jan 1971).

[4] Davies, D. & Wakerly, J. F., "Synchronization & Matching in Redundant Systems", *IEEE Transactions on Computers C-27*, (June 1978), 531-539.

[5] Dennis, J. B. & Misunas, D. P., "A Preliminary Architecture for a Basic Data-Flow Processor", Symposium on Computer Arch, (Feb 1975), pp.126-132.

[6] Leung, C. K. C. & Dennis, J. B., "Design of a Fault-Tolerant Packet Communication Computer Architecture", MIT-TR, (Feb 1980).

[7] Tesler, L. G. & Enea, H. J., "A Language Design for Concurrent Processes", NCC, (Spring 1968), pp. 403-408.

[8] VonNeumann, J., "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components", in *Automata Studies*, Sharon & McCarthy, Eds., Princeton Univ Press, 1956, pp.43-98

SYNCHRONOUS NETS FOR SINGLE INSTRUCTION STREAM - MULTIPLE DATA STREAM COMPUTERS

Annette J. Krygiel
Defense Mapping Agency
Washington, D.C. 20305

Abstract -- Synchronous Nets, or S Nets, are developed as a modeling tool particularized for describing processes on Single Instruction Stream - Multiple Data Stream (SIMD) computers. S Nets are a modification of Petri Nets, using transitions and places to model events and conditions. However, S Nets introduce vector-mask places to model the conditions of the array resources of SIMD machines. These places are distinguished from scalar places which model the scalar resources. S Nets also introduce a new kind of transition. One type correlates with the Petri Net transition, but the mask firing transition is particularized to the SIMD environment, modeling the inherent capability of a computation executing on a SIMD machine to alter the participation of the vector aggregates in successor events.

Introduction

This paper is concerned with the problem of mapping algorithms onto certain classes of parallel processors to exploit parallelism in the algorithm to the maximum extent supportable by the machine on which it is to be implemented. The approach taken is one of providing a tool -- a graph-based modeling system called Synchronous Nets or S Nets -- to describe such an implementation.

SIMD Architectures

The processors of interest are of the single instruction stream-multiple data stream (SIMD) architectures as described by Flynn [1] who distinguishes four classes:

Single instruction stream - single data stream (SISD)

Single instruction stream - multiple data stream (SIMD)

Multiple instruction stream - single data stream (MISD)

Multiple instruction stream - multiple data stream (MIMD)

Figure 1 illustrates an SIMD architecture, which typically consists of a control unit with its own memory, and (possibly) some limited processing capability; an array or vector unit consisting of N Processing Elements (PEs) and at least N memories (PEMs); and an interconnection network for interprocessor communication. Associated with each PE is some indicator (mask) for signaling participation or non-participation in instructions. In implementation, usually a conventional sequential machine is attached to the control unit, i.e. a mini-host.

A Multiple-SIMD (MSIMD) architecture is configured as two or more independent SIMD machines, each with its own control unit, array unit, etc., and with one interconnection network. These SIMD components have the ability to perform synchronously, and using the same instruction stream or different instruction streams. Such an architecture is illustrated by Figure 2.

SIMD machines are considered "special purpose." They perform spectacularly on problems to which they are well suited [2, 3, 4, 5]. To derive high performance, the application should have a high degree of parallelism, with the algorithm consistent with the topology of the machine. Unfortunately there are no simple means to gauge this desired isomorphism. Modeling is one approach that can be employed. S Nets were specifically developed to accomplish and facilitate this, and are a modification of Petri Nets [6, 7, 8, 9] supplying constructs particularized to SIMD (and MSIMD) architectures.

Definition of Synchronous Nets

System Overview

A Synchronous Net, or S Net, is a directed graph with a marking and a set of descriptors [10]. The vertices of the graph are vector-mask places, scalar places, and transitions. Scalar places and vector-mask places are connected with arcs to transitions and vice versa. A marking associates a non-negative integer with each scalar place, and associates a tuple of non-negative integers with each vector-mask place. The non-negative number is called the number of tokens. Descriptors are associated with each transition and characterize the behavior of the transition.

As with Petri Nets S Nets use transitions to model events and places to model conditions with arcs representing the paths allowed for passage of control. Analogous to Petri Nets S Nets exhibit dynamic behavior resulting from the firing of transitions. The firing of a transition models the occurrence of an event; tokens in a place can model the holding of a condition.

The key differences between Petri Nets and S Nets are the S Net innovations of vector-mask places and mask firing transitions. Vector-mask places model aggregates of logically associated and homogeneous conditions whose initial and

0190-3918/81/0000/0266$00.75 © 1981 IEEE

ceasing events are synchronized, i.e., the conditions of a set of array processors. These aggregates are further characterized by the fact that the marking of some members of the aggregate may be relevant to the firing of a successor transition while others may not. This characteristic can model the participation or non-participation of some elements of the array processor in subsequent events.

Unique to S Nets is the concept of two kinds of transition firings -- one of which — the mask firing — provides for alternatives in the markings of the aggregates. This enables modeling of changes in the participation or non-participation of the elements of an array processor as it proceeds from event to event. These alternatives are formalized by descriptors associated with each transition.

S Net Graphs

S Nets will be defined in terms of sets. The element of a set will be designated within { }. The CARDINALITY of any set shall be designated | | and refers to the number of elements in the set, i.e., $|S|$ represents the number of elements in S. For example,

$$\text{if } S = \{s_1, s_2, \ldots s_j\}, \text{ then } |S| = j.$$

Also important in the S Net definition is the notion of tuples denoted by < > and consisting of ordered components. The cardinality of a tuple is also designated | |, but it is more appropriately called its DIMENSIONALITY.

An S Net Graph will be a quadruple (T,S,U, A), with an initial marking K_0 and a set of transition descriptors D, where:

T = A finite set of transitions $\{t_1, t_2, \ldots t_{|T|}\}$.

S = A finite set of scalar places $\{s_1, s_2, \ldots s_{|S|}\}$.

U = A finite set of vector-mask places $\{<V_1, M_1>, <V_2, M_2>, \ldots <V_{|U|}, M_{|U|}>\}$.

A = A finite set of directed arcs $\{a_1, a_2, \ldots a_{|A|}\}$, such that

$A \subseteq (P\text{x}T) \cup (T\text{x}P)$, where $P = U \cup S$ and P is called the set of places.

Thus the elements of A are of the form $<p_j, t_k>$ or $<t_j, p_k>$, so that an arc either connects a place to a transition or a transition to a place.

The set U is defined as a subset of VxM, where:

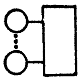
V = A finite set of elements called vector places $\{V_1, V_2, \ldots V_{|V|}\}$; each element of V, designated V_i, is a tuple containing some number of ordered components, i.e., $V_i = <v_{i1}, v_{i2}, \ldots v_{ip}>$, $p > 1$, and $|V_i|$ does necessarily equal $|V_j|$ when $i \neq j$, but $|V| = |U|$.

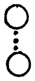
M = A finite set of elements called masks $\{M_1, M_2, \ldots M_{|M|}\}$; each element of M, designated M_i, is a tuple containing some number of ordered components, i.e.,

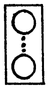
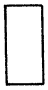
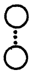
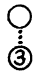
$M_i = <m_{i1}, m_{i2}, \ldots m_{iq}>$, $q > 1$, and $|M_i|$ does not necessarily equal $|M_j|$ when $i \neq j$, but $|M| = |U|$.

If P is $U \cup S$, and A is as defined, the triple (P,T,A) is a bipartite directed graph since all nodes can be partitioned into two sets, transitions and places, such that each arc directed FROM an element of one set is directed TO an element of the other set, and vice versa. Therefore arcs from(to) a vector-mask place or a scalar place are always directed to(from) a transition.

In the S Net Graph, transitions are represented by |, the scalar places are represented by ◯, and the vector-mask places by

 . Within that last symbol, the vector

symbol V_i is  and the mask symbol M_i is

 or  . The dimensionality of V_i or $|V_i|$ in  is portrayed as  .

The dimensionality of M_i is not noted on the graph, but is specified in the formal designation of M_i components, i.e.,

$<m_{i1}, m_{i2}, \ldots m_{i|M_i|}>$. Arcs are denoted as ⟶ .

Given the S Net shown in Figure 3, we shall delineate the graph of the S Net as follows:

$T = \{t_1, t_2, t_3, t_4, t_5, t_6\}$

$S = \{s_1, s_2, s_3\}$

$V = \{V_1, V_2, V_3, V_4\}$

$V_1 = \langle v_{11}, v_{12}, v_{13}\rangle$

$V_2 = \langle v_{21}, v_{22}, v_{23}\rangle$

$V_3 = \langle v_{31}, v_{32}, v_{33}\rangle$

$V_4 = \langle v_{41}, v_{42}, v_{43}\rangle$

$\mathcal{M} = \{M_1, M_2, M_3, M_4\}$

$M_1 = \langle m_{11}, m_{12}, m_{13}\rangle$

$M_2 = \langle m_{21}, m_{22}, m_{23}\rangle$

$M_3 = \langle m_{31}, m_{32}, m_{33}\rangle$

$M_4 = \langle m_{41}, m_{42}, m_{43}\rangle$

$U = \{\langle V_1, M_1\rangle, \langle V_2, M_2\rangle, \langle V_3, M_3\rangle, \langle V_4, M_4\rangle\}$

$A = \{\langle s_1, t_1\rangle, \langle t_1, \langle V_1, M_1\rangle\rangle, \langle\langle V_1, M_1\rangle, t_2\rangle, \langle t_2, \langle V_2, M_2\rangle\rangle, \langle\langle V_2, M_2\rangle, t_3\rangle, \langle t_3, s_2\rangle, \langle t_1, \langle V_3, M_3\rangle\rangle, \langle\langle V_3, M_3\rangle, t_4\rangle, \langle t_4, \langle V_4, M_4\rangle\rangle, \langle\langle V_4, M_4\rangle, t_3\rangle, \langle s_2, t_5\rangle, \langle t_5, s_1\rangle, \langle s_2, t_6\rangle, \langle t_6, s_3\rangle\}$

S Net Structure

Analogously to a Petri Net, the structure of an S Net is defined so as to make clear the relationship of places and transitions. The INPUT PLACES of a transition $\mathcal{I}(t)$ are all scalar places and vector-mask places directed immediately TO the transition. The OUTPUT PLACES of a transition $O(t)$ are all scalar places and vector-mask places directed immediately FROM the transition.

Markings

The infinite set of non-negative integers $\{0, 1, \ldots\}$ is designated N; the set of Boolean numbers $\{0, 1\}$ is designated B; the set $\{0\}$ is designated Z. Also the r-fold Cartesian products are defined:

$N^r = N x N x \ldots N$; each member is of the form $\langle N_1, N_2, \ldots N_r\rangle$

$B^r = B x B x \ldots B$; each member is of the form $\langle B_1, B_2, \ldots B_r\rangle$

$Z^r = Z x Z x \ldots Z$; each member is of the form $\langle 0, 0, \ldots 0\rangle$

where N_i and B_i, are elements of N and B, respectively.

A MARKING is a function K where

$$K: \quad S \rightarrow N; \; V_i \rightarrow N^{|V_i|} \text{ for all } V_i \in \mathcal{V};$$
$$M_i \rightarrow B^{|M_i|} \text{ for all } M_i \in \mathcal{M}.$$

The marking associates a non-negative integer with each scalar place — $K(s)$ for each $s \in S$ — and two vectors of non-negative integers with each vector-mask place, one of those associated vectors being a Boolean vector — $K(V_i)$ for each $V_i \in \mathcal{V}$ and $K(M_i)$ for each $M_i \in \mathcal{M}$.

A marking for an S Net must specify all three components.

An INITIAL MARKING K_0 is defined as the first marking of the S Net.

A MASK MARKING for a mask M_i is a function K such that $M_i \rightarrow B^{|M_i|}$.

The set of possible mask markings for any M_i is $W(M_i)$ and denotes the co-domain of a mask marking, consisting of designated tuples of $B^{|M_i|}$, or if appropriate, the entire product set $B^{|M_i|}$.

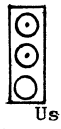
Notation for Markings. The convention adopted to show markings will be that of ( ) and < >. The former is used to distinguish the marking of a single element or component, and the latter is used when more than one element or component is involved, thereby denoting an ordering with respect to markings of elements or components.

As an example, markings for places s_1, s_2, s_3 are designated $K(s_1) = (0)$; $K(s_2) = (0)$; $K(s_3) = (0)$.

If $\mathcal{V} = \{V_1, V_2\}$, and if $V_1 = \langle v_{11}, v_{12}, v_{13}\rangle$, then $K: \; V_1 \rightarrow Z^3$ is equivalent to: $K(V_1) = \langle 0, 0, 0\rangle$; alternately $K(v_{11}) = (0)$; $K(v_{12}) = (0)$; $K(v_{13}) = (0)$. Similarly, for $M_1 = \langle m_{11}, m_{12}, m_{13}\rangle$, a marking $K(M_1) = \langle 1, 0, 0\rangle$ designates that $K(m_{11}) = (1)$, $K(m_{12}) = (0)$, $K(m_{13}) = (0)$.

Graphic Portrayal of Markings. Markings are illustrated with the presence of tokens. Dots in any place represent tokens. Tokens in masks may, alternatively, be represented by Boolean symbols for legibility.

The symbol $\boxed{\begin{matrix}1\\1\\0\end{matrix}}$ for M_i shows a token in m_{i1} and m_{i2}. This is synonomous to the symbol



.

Using the S Net example of Figure 3, Figure 4 illustrates a marking where masks are marked with tokens but vector places are not marked with tokens. This example assumes the initial marking is:

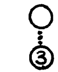
$K_0(s_1)$ = (1); $K_0(s_2)$ = (0); $K_0(s_3)$ = (0)

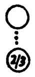
$K_0(V_1) = K_0(V_2) = K_0(V_3) = K_0(V_4)$ = <0, 0, 0>

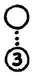
$K_0(M_1) = K_0(M_2)$ = <1, 0, 0>

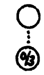
$K_0(M_3)$ = <1, 1, 1>

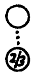
$K_0(M_4)$ = <0, 0, 1>

An assignment of tokens to a vector place V_i may leave some of the component places marked with tokens and others empty, i.e., all elements v_{ij} of V_i may not have tokens. Since the $|V_i|$ may be large, graphic designation of which components are marked must necessarily be limited. For example, a 

portrays a V_i with three components; then 

indicates that two v_{ij} ε V_i are marked. Synonomous are the symbols 

and 

. However 

only conveys that two v_{ij} are marked but does not distinguish the individual elements, nor does it indicate how many tokens are in each marked v_{ij}. However S Nets use vector-mask places to model conditions resulting from and leading to events, and V_i is always expressed graphically in conjunction with M_i. It is M_i which will be used graphically to enhance comprehension of which v_{ij} are marked — at least in markings resulting from an execution of the S Net.

Rules for Execution

The graph and structure of S Nets have been addressed in previous Sections. Now discussed is the dynamic behavior of S Nets.

Enabled Transitions. A scalar place is HOLDING if it has at least one token in it. A vector-mask place $<V_i, M_i>$ is HOLDING if:

at least one $K(m_{ij})$ = (1), j = 1, 2,... $|V_i|$, and v_{ij} ε V_i has a non-zero marking for all those j for which m_{ij} ε M_i has a non-zero marking.

A holding for a vector-mask place is in contrast to a marking of that place. Whereas a marking associates some set of integers with vector-mask places, a holding for a vector-mask place REQUIRES that the components of V_i be marked with tokens everywhere that their associated M_i components are marked with tokens.

A transition t is ENABLED also called FIRABLE under the following conditions: t is ENABLED if all scalar places in I(t) are holding and all vector-mask places in I(t) are holding.

Firing Transitions. A FIRING is a function of a transition which has for its domain and range the marking of the input places and output places of the transition. There is a firing associated with every enabled transition t. When a transition t is enabled, its firing function is defined at a given marking K_n of the S Net, and the firing yields K_{n+1}, a new marking.

Transition Types. A TRANSITION TYPE specifies the firing capabilities of the transition — either simple or mask firing — designated SFT and MFT respectively.

Transition Descriptors. A TRANSITION DESCRIPTOR D[t] specifies the transition type, either SFT or MFT, and for every vector-mask output place $<V_i, M_i>$ of the transition, specifies $W(M_i)$, the set of markings for M_i. Descriptors for a transition t with vector-mask output places $<V_i, M_i>$, $<V_j, M_j>$,...$<V_r, M_r>$ are specified:

$$D[t] = [\text{type}; K(M_i) \in W(M_i), K(M_j) \in W(M_j), \ldots K(M_r) \in W(M_r)].$$

Rules for a Simple Firing. A SIMPLE FIRING associated with an enabled transition t is such that:

For every scalar input place s, then

$$K_{n+1}(s) = K_n(s) - 1$$

For every scalar output place s, then

$$K_{n+1}(s) = K_n(s) + 1$$

For every vector-mask input place $<V_i, M_i>$, then: for v_{ij} ε V_i, j = 1, 2,... $|V_i|$,

$K_{n+1}(v_{ij}) = K_n(v_{ij}) - 1$ for those j for which m_{ij} ε M_i has a non-zero marking; and for m_{ij} ε M_i,

$K_{n+1}(m_{ij}) = K_n(m_{ij})$ for all j.

For every vector-mask output place $<V_i, M_i>$, then: for v_{ij} ε V_i, j = 1, 2,... $|V_i|$,

$K_{n+1}(v_{ij}) = K_n(v_{ij}) + 1$ for those j for which m_{ij} ε M_i has a non-zero marking; and for m_{ij} ε M_i,

$K_{n+1}(m_{ij}) = K_n(m_{ij})$ for all j.

As seen from the firing rules, SFTs do not alter their input or output masks.

Rules for a Mask Firing. A MASK FIRING is associated with an enabled transition t that has at least one $<V_i, M_i>$ output place, and is such that:

For every scalar input place s, then

$$K_{n+1}(s) = K_n(s) - 1$$

For every scalar output place s, then

$$K_{n+1}(s) = K_n(s) + 1$$

For every vector-mask input place $<V_i, M_i>$,

then: for $v_{ij} \varepsilon V_i$, $j = 1, 2, \ldots |V_i|$,

$K_{n+1} (v_{ij}) = K_n (v_{ij}) - 1$ for those j for which $m_{ij} \varepsilon M_i$ has a non-zero marking; and for $m_{ij} \varepsilon M_i$,

$$K_{n+1} (m_{ij}) = K_n (m_{ij}) \text{ for all j.}$$

For every vector-mask output place $<V_i, M_i>$,

then for M_i,

$K_{n+1} (M_i) \varepsilon W(M_i)$, where $W(M_i)$ is specified by the transition descriptor D[t], and $K_{n+1}(M_i)$ is non-deterministically chosen.

For every vector-mask output place $<V_i, M_i>$,

then for $v_{ij} \varepsilon V_i$, $j = 1, 2, \ldots |V_i|$,

$K_{n+1} (v_{ij}) = K_n (v_{ij}) + 1$ for those j for which $m_{ij} \varepsilon M_i$ has a non-zero marking, i.e., where $K_{n+1} (m_{ij}) = (1)$.

The assignment of a Boolean vector to M_i by MFT is a mapping of M_i INTO $W(M_i)$, where the domain is M_i and the co-domain consists of the elements of $W(M_i)$, i.e., $M_i \xrightarrow{\text{INTO}} W(M_i)$.

By the firing rules, firings remove tokens from places and add tokens to other places, and in the case of the mask firing mark the masks of the vector-mask output places. It should be noted that the number of tokens subtracted by a transition firing does not necessarily equal the number that it adds.

Transitions and Their Descriptors. As seen from the firing definitions, SFTs on firing do not change the $K(M_i)$ of their $<V_i, M_i>$ input and output places. The transition descriptor is noted simply as D[t] = [SFT; _].

For MFTs, the $|W(M_i)| \geq 1$ for all output masks, and since these markings are determined by the transition firing and not the initial marking, the set of markings must be listed in the transition descriptor, i.e.,

$$D[t] = [MFT; K(M_i) \varepsilon \{< \quad >, < \quad > \ldots\}]$$

Example of Mask Firing Transitions. Figure 5 through Figure 8 show MFTs in an S Net and illustrate their graphic portrayal and behavior. Given an initial marking and descriptors:

$$K_0(s_1) = (1): K_0(s_2) = (0); K_0(s_3) = (0)$$

$$K_0(V_1) = K_0(V_2) = <0, 0, 0>$$

$$D[t_1] = [MFT; K(M_1) \varepsilon \{<1, 0, 0>, <0, 0, 1>\}]$$

$$D[t_2] = [MFT; K(M_2) \varepsilon \{<1, 0, 0>, <0, 0, 1>\}]$$

$$D[t_3] = D[t_4] = D[t_5] = [SFT;\_]$$

Figure 5 reflects the initial marking and shows that t_1 is enabled. Transition t_1 has one output mask and $|W(M_1)| = 2$, both elements of which are shown on the graph. When transition t_1 fires, the results are shown in Figure 6. The marking assigned to M_1 by t_1 was <1, 0, 0> which is designated in $D[t_1]$ and is shown on the graph as one member of the set $W(M_1)$. (At the time of firing the mask marking that is chosen by the transition is arbitrary.) After t_1 fires, v_{11} receives one token since m_{11} is marked with a token, and a token is removed from s_1.

The v_{11} token enables t_2 since m_{11} also holds a token, so that the firing of t_2 can commence. If t_2 fires changing the marking $K(M_2)$ to <1, 0, 0>, and if the firing sequence t_1, t_2, t_3, t_4 is assumed, then Figure 7 illustrates the marking after t_4 has fired. In Figure 7 a token is again in s_1; $K(M_1)$ is <1, 0, 0> from the previous firing of t_1; $K(M_2)$ is <1, 0, 0> as marked from the previous firing of t_2; $K(V_1)$ is <0, 0, 0> since the token in v_{11} placed there as a result of the first t_1 firing was removed at the firing of t_2. $K(V_2)$ is <0, 0, 0> since the token in v_{21} placed there after the firing of t_2 was removed at the firing of t_3.

Where Figure 7 shows t_1 enabled, Figure 8 shows the results after the second firing of t_1. Here $K(M_1)$ = <0, 0, 1>, a marking alternative also described in $D[t_1]$. (The selection of the marking is arbitrary, and could have been <1, 0, 0> again.) Given $K(M_1)$ = <0, 0, 1>, by firing definition v_{13} now receives a token. Since m_{13} also holds a token, t_2 is enabled, and so on.

To analyze an algorithm, a sequence of S Net transition firings can be examined. The sequence resolves conflict, or indeterminacies in computational flow, in that a particular order of firings is assumed. (Conflict is typified in Figures 5 through 8 by t_4 and t_5 which share an input place;

when one token resides in the place, only one transition can fire and which transition fires is indeterminate [6, 7, 8, 9, 10]). Analogously, in modeling a specific computation, the indeterminacy of mask selection by MFTs is not troublesome by assuming an order of mask selections.

S NET APPLICATION

To apply S Nets it is necessary to relate the model to the actual algorithm. An INTERPRETATION of an S Net is an assignment of labels to the transitions and/or places of the Net to indicate for the transition the event that it models and for the place the condition that it models.

Consider the example of summing the rows of a 4x4 matrix A, multiplying the sum by a 4x1 vector B, and storing results in the first column of A. The FORTRAN description is:

```
    DO 200 I = 1, 4
    DO 100 J = 1, 3
100 A(I, 1) = A(I, 1) + A(I, J+1)
200 A(I, 1) = A(I, 1) * B(I)
```

Assume an 8 PE SIMD machine is available. A data storage scheme for the vectors is depicted in Figure 9, which indicates that PE_0 has the first row of A in its PEM, PE_1 has the second row, etc. With parallel hardware available, the four row sums can be formed simultaneously. This is shown in the S Net model of Figure 10 which is marked to reflect a holding of condition after a t_2 firing. The S Net uses vector-mask places to model conditions in array resources and scalar places to model conditions in scalar resources. The masks of the vector-mask places model control over the participation of the array resources.

Transition t_2 models the event which adds the Jth column of matrix A to the first column of matrix A. Places s_4 and s_5 model the conditions resulting from the test of J. Assuming a sequence of firings such that t_4 and t_5 have conflict resolved by the status of loop index J, when t_4 fires, all rows have been summed; then t_7 models a parallel multiplication of all four sums by the appropriate element of vector B. The utilization of the array resources is a by-product of this S Net execution, i.e., the marking on the vector-mask output place resulting from the t_2 firing is depicted on Figure 10 as 4/8. Both the parallelism achieved by the array resource (4) and the utilization with respect to the maximum parallelism supportable by the array hardware (8) becomes apparent. Also, the marking of the mask suggests some additional management activity required of the vector resource and is specific as to which PEs will participate -- PE_0 through PE_3.

With every execution of the array events modeled by SFTs t_2 and t_7, PE_0 through PE_3 always participate. However if the problem context is changed to require the alternative of using PE_4 through PE_7 in a different iteration of the computation, these array events are more aptly modeled by MFTs, i.e.,

$$D(t_2) = [MFT; K(M_1) \varepsilon \{<1^4, 0^4>, <0^4, 1^4>\}]$$

$$D(t_7) = [MFT; K(M_2) \varepsilon \{<1^4, 0^4> \ <0^4, 1^4>\}]$$

For analysis, an assumption would then be made about the order of selection of the masks, i.e., $<1^4, 0^4>$, then $<0^4, 1^4>$. This capability for alternation is readily distinguishable on the graph, contributing more detail for analysis.

More exposition of the modeling capability of S Nets, particularly the properties of concurrency and conflict, is supplied in [10] as are additional (and less simple) examples. SIMD algorithms can be modeled with Petri Nets, but with increased modeling complexity. S Nets are distinct from Petri Nets in the notions of vector-mask places and mask firing transitions. Many Petri Net places are created in lieu of a single vector-mask place. It requires many Petri Net transitions in forward conflict to model the more concise mask firing transition [10].

The richer detail of S Nets is illustrated by modeling Flynn's classes of architectures shown in Figure 11. Because of the availability of vector-mask places in addition to scalar places, the multiplicity of the data stream can now be depicted; also scalar and vector activity can be distinguished. The SIMD architecture is readily distinguished from SISD by the added detail of vector-mask places. The MIMD-2 architecture which allows both scalar and vector concurrency, is distinguishable from the MIMD-3 architecture which allows concurrent SIMD resources (MSIMD). Both are clearly different from the MIMD-1 architecture of conventional distributed processors.

SUMMARY

In this paper, Synchronous Nets, or S Nets, have been formally defined. S Nets are a modification of Petri Nets, specifically developed to provide richer detail for modeling the SIMD environment. It is possible to model SIMD computations with Petri Nets but at the expense of increased modeling complexity. However, the relationship of S Nets and Petri Nets is explored elsewhere [10].

REFERENCES

[1] Flynn, Michael J., "Very High-Speed Computing Systems", Proceedings of the IEEE, Volume 54, No. 12, December 1966, pp. 1901-1909.

[2] Thurber, Dennis J., and Wald, L. D., "Associative and Parallel Processors", Computing Surveys, Volume 7, No. 4, December 1975, pp. 215-255.

[3] Ruben, Sherwin, et al., "Application of a Parallel Processing Computer in LACIE", Proceedings of the 1976 International Conference on Parallel Processing, pp. 24-32.

[4] Krygiel, Annette J., "An Implementation of the HADAMARD Transform on the STARAN Associative Array Processor", Proc. 1976 International Conference on Parallel Processing, p. 34.

[5] Daley, J. S., and Underwood., B. D., "Short-Term Weather Prediction on ILLIAC IV", Proc. 1975 Sagamore Computer Conference on Parallel Processing, p. 240.

[6] Petri, Carl A., "Kommunikation mit Automaten", Translation by C. F. Greene, Supplement 1 to RADC-TR-65-337, Vol 1, RADC, Griffiss AFB, New York, 1962.

[7] Holt, A. W., et al, "Information System Theory Project", Applied Data Research, Inc., RADC-TR-68-305, Rome Air Development Center, Griffiss AFB, New York, September 1968.

[8] Holt, A. W., and Commoner, Frederic, "Events and Conditions", Applied Data Research, Inc., New York, 1979.

[9] Peterson, James L., "Petri Nets", Computing Surveys, Volume 9, pp. 223-252.

[10] Krygiel, Annette J., "Synchronous Nets for Single Instruction Stream - Multiple Data Stream Computers", D. Sc Dissertation, Sever Institute of Technology, Washington University, St. Louis, MO, May 1980.

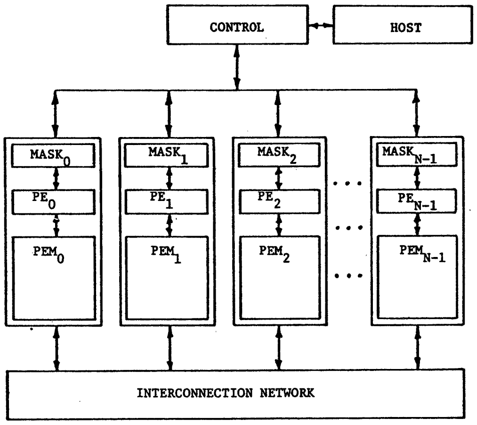


Figure 1 SIMD Architecture

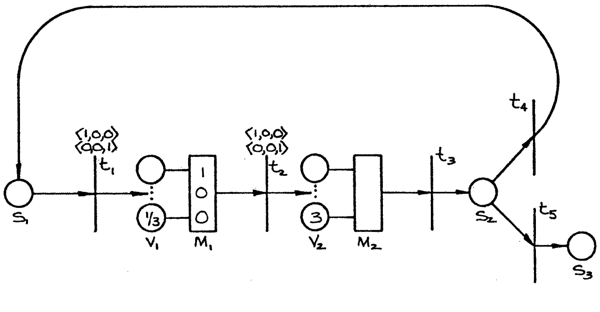
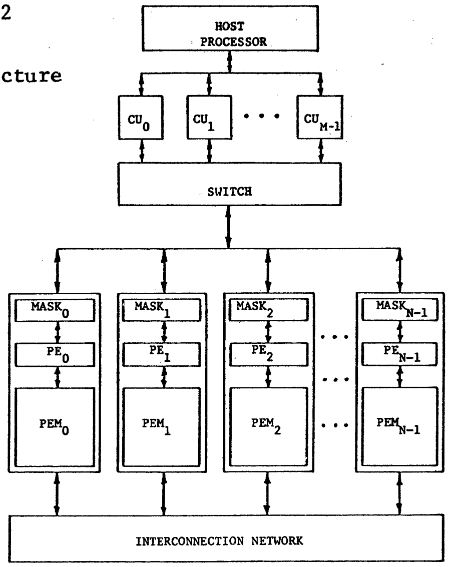


Figure 6 MFTS t_1 and t_2 at K_1

Figure 2 MSIMD Architecture



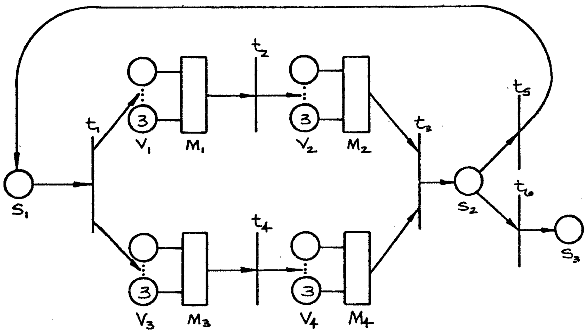


Figure 3 An S Net

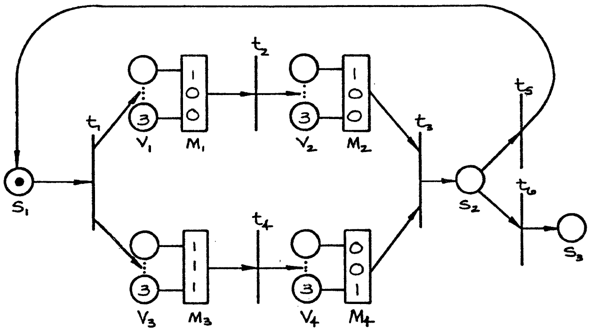


Figure 4 An S Net With M_i Marked

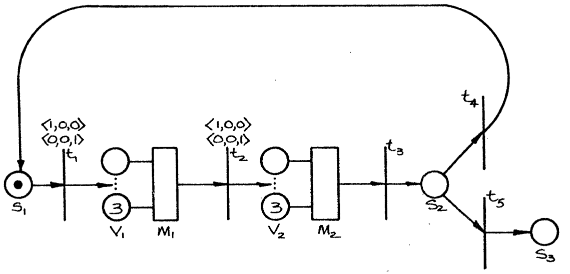


Figure 5 MFTs t_1 and t_2 at K_0

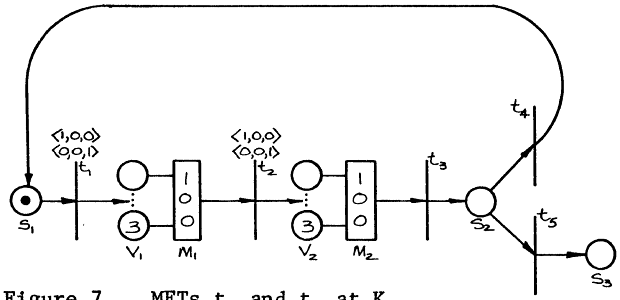


Figure 7 MFTs t_1 and t_2 at K_4

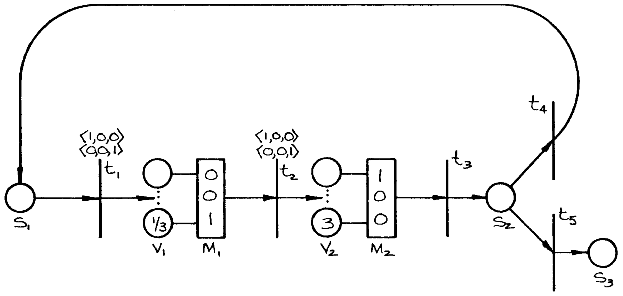


Figure 8 MFTs t_1 and t_2 at K_5

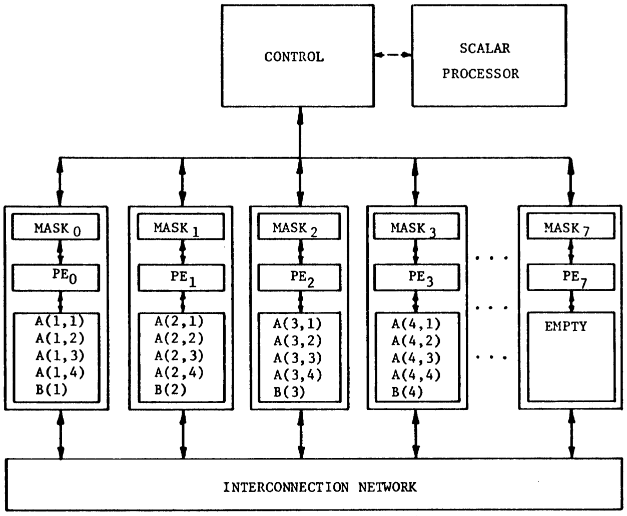


Figure 9 Data Storage Scheme for Row Sum

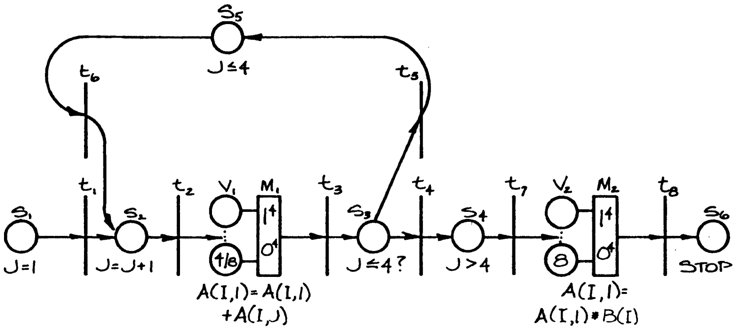
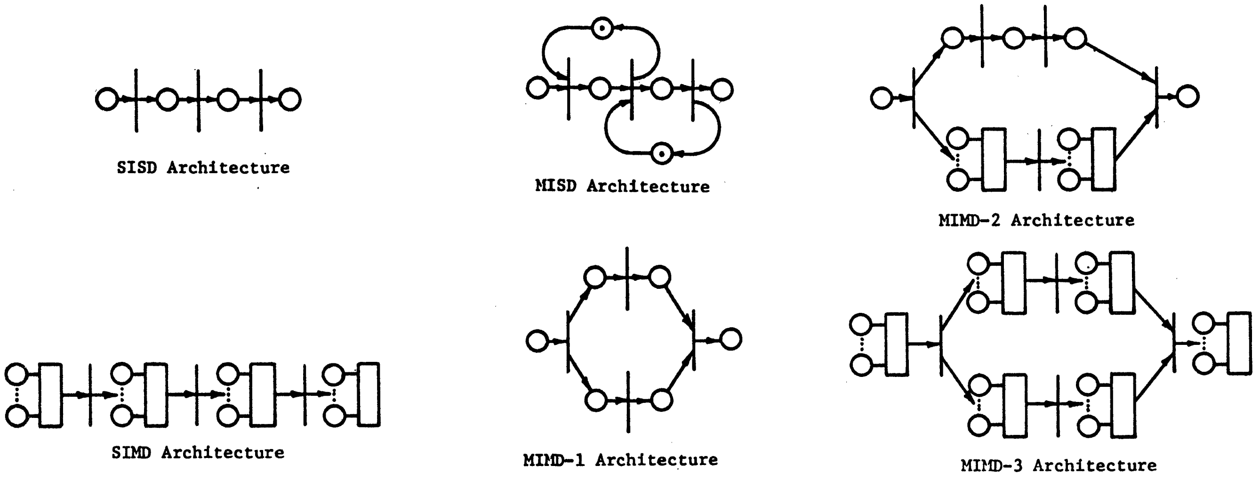


Figure 10 S Net Model for Row Sum



SISD Architecture

MISD Architecture

MIMD-2 Architecture

SIMD Architecture

MIMD-1 Architecture

MIMD-3 Architecture

Figure 11 S Nets Depicting Machine Architectures

MINIMIZATION OF INTERPROCESSOR COMMUNICATION FOR PARALLEL COMPUTATION

Keki B. Irani

Department of Electrical
and Computer Engineering
The University of Michigan
Ann Arbor, MI. 48109

Kuo-Wei Chen

Program in Computer, Information
and Control Engineering
The University of Michigan
Ann Arbor, MI. 48109

ABSTRACT -- This paper is concerned with minimizing the delay due to data communication during the execution of a parallel algorithm on an SIMD computer with a two-way circular unit-shift interconnection network. Algorithms are developed which determine, for a given parallel algorithm, the order of computation of a parallel arithmetic expression, the alignment of operands for every binary operation, and the mapping and remapping of data into physical memories so that the commmnication cost is minimized. The proposed techniques are applicable to array variables with special types of index functions.

1. INTRODUCTION

The total execution time of a parallel algorithm on a multiprocessor system can be broken down into the actual computation time and the time of interprocessor data communication. On an SIMD (Single-Instruction stream, Multiple-Data stream) machine, the computation time is usually dependent only on the number of processing elements, and the communication time is dependent on several factors such as the interconnection network and the storage scheme for data. Previous analyses [1,2,8] have shown that the data communication can be a major cause of degradation of the performance of the algorithm. In this paper, we study the minimization of the communication cost for a class of parallel algorithms and interconnection networks, which will be described shortly.

A restricted version of this minimzation problem is considered in [11], where it is called the mapping problem, and the emphasis is on the effect of data storage schemes on the communication time. Some further exploitation of the problem is reported in [6]. In section 2, the minimization problem is formally defined. In section 3, we provide algorithms for reordering of computation in order to minimize communication delay. In section 4, we provide algorithms first for static mapping of data and then for data remapping for further improvement of communication delay. Proofs for the optimality of these algorithms are to be found in [5].

2. THE MINIMIZATION PROBLEM

Since this minimization problem deals with both software (algorithms) and hardware (interconnection network) of the SIMD system, a few parameters from both parts are needed for formally defining the problem [6]. In what follows, N denotes the number of both logical memories and physical memories. These parameters are:

i. The index set $\{0,1,2,\ldots,N-1\}_m$ of the logical memories m of the algorithm. Each component a_i of a vector A is assumed to be stored in logical memory m_i.

ii. A sequence of logical data transfers that satisfies the communication requirement of the algorithm. Each logical transfer p_{uv} is a partial function, for the alignment of data u and v. It maps the index set $\{0,1,\ldots N-1\}_m$ into itself.

iii. The index set $\{0,1,\ldots,N-1\}_M$ of the physical memories.

iv. The set Q of interconnection functions $\{q_1,q_2,\ldots,q_k\}$ that defines the interconnection network. Each q_i is a bijection on the index set $\{0,1,\ldots,N-1\}_M$, and designates the transfer that can be physically effected in one routing step.

v. The distance function D, associated with the interconnection network, on the set of all partial functions on $\{0,1,\ldots,N-1\}_M$. The value of this function on a partial function is the minimum number of routing steps needed by the network to realize that partial function.

vi. A mapping, or, storage scheme, F_v for vector v of the algorithm is a bijection from $\{0,1,\ldots,N-1\}_m$ onto $\{0,1.\ldots,N-1\}_M$.

Let F_u and F_v be the mapping functions for variables u and v. respectively, and let p_{uv} be a logical transfer for aligning u and v. Suppose p_{uv} aligns u and v by moving u to v. Then at the logical level, p_{uv} corresponds to moving the element $u(F_u^{-1}(i))$ from logical memory $m_{F_u^{-1}(i)}$ to $m_{p_{uv}F_u^{-1}(i)}$. At the physical level, however, this data movement is from memory M_i, where $u(F_u^{-1}(i))$ is stored, to $M_{F_v p_{uv} F_u^{-1}(i)}$. Hence the cost of such an alignment operation is $D(F_v p_{uv} F_u^{-1})$.

Instead of aligning the variable u with the variable v, or v with u for a binary operation involving u and v, suppose the variables are aligned somewhere else. For example, suppose the binary operation $A(i + k_1) + B(i + k_2)$ is performed in the processor $(i + t)$ and the result stored in the memory M_{i+t}. There is no loss of

0190-3918/81/0000/0274$00.75 © 1981 IEEE

generality if the resulting vector is named $W(i + t)$. In other words, the mapping function F_W is an identity function. The total cost of communication in this case is the sum of the costs of aligning u with w and aligning v with w. This sum is given by $D(p_{uw}F_u^{-1}) + D(p_{vw}F_v^{-1})$. For the above example, we have $p_{AW}(i) = (i + (t - k_1))$ and $p_{BW}(i) = (i + (t-k_2))$. (The arithmetic on the indices in this paper are all modulo N.)

The above result can be easily generalized to a parallel expression S. Let $v_1, v_2 \ldots, v_k$ be the variables in S and let E denote an expression tree for S. Let $w_1, w_2 \ldots, w_t$ be the internal nodes of E where each w_i represents the partial result of some binary operation on variables and/or other internal nodes. For a given expression tree E, which specifies the order of computation of S, the alignment of operands for every binary operation must be determined from which the logical transfer functions can be specified. Thus, for a statement S, the communication cost is

$$\sum_{\substack{v_i:\ \text{variable (leaf node)}\\ w_j:\ \text{partial result (internal node)}\\ w_j=\text{PARENT}(v_i)}} D(p_{v_i w_j} F^{-1}_{v_i}) + \sum_{\substack{w_i, w_j:\text{internal nodes}\\ w_j=\text{PARENT}(w_i)}} D(p_{w_i w_j}) \qquad (1)$$

where $\text{PARENT}(v_i)$ denotes the parent node of v_i in tree E.

Notice that for a given statement and a given interconnection network, the total cost depends on the mappings F_{v_i}'s and the transfer functions $p_{v_i w_j}$'s and $p_{w_i w_j}$'s. The transfer functions used in the cost calculation depend on the expression tree itself (for an expression involving commutative operations the expression tree is not unique) as well as on the alignments of operands.

The complete minimization problem for a complete algorithm can therefore be stated as follows:

Given a parallel algorithm with $\{0,1,\ldots,N-1\}_m$, an interconnection network with $\{0,1,\ldots,N-1\}_M$, and D, determine the following:

1. an expression tree, or, a computation ordering, for every parallel assignment statement,

2. alignments of operands for binary operations,

3. a mapping function F_v for every variable v, such that

$$\sum_{\substack{\text{statements of}\\ \text{given algorithm}}} \Big(\sum_{\substack{v_i:\ \text{variable (leaf node)}\\ w_j:\ \text{partial result (internal node)}\\ w_j=\text{PARENT}(v_i)}} D(p_{v_i w_j} F^{-1}_{v_i}) + \sum_{\substack{w_i, w_j:\text{internal nodes}\\ w_j=\text{PARENT}(w_i)}} D(p_{w_i w_j}) \Big) \qquad (2)$$

is minimized.

The general minimization problem is very complex, and we shall restrict ourselves to the following environment. The SIMD machine is assumed to have a two-way circular unit-shift network on N processing elements (PEs), where, for notational simplicity, N is assumed to be an even integer, i.e., for this network, $Q=\{q_1, q_2\}$ and $q_1(i)=(i+1)$, $q_2(i)=(i-1)$. For convenience, this network will be referred to in what follows as the circular network. Also assumed is the fact that the index functions of the array variables of the algorithms are all of the form: i + constant. In other words, we assume that the logical transfers for the variables are all of the uniform-shift type, i.e., for some variables u and v, $p_{uv}(i)=(i+k) \bmod N$, where k is an integer constant. If an algorithm uses both uniform-shift type and other types of transfers, then we shall consider the minimization problem for those varialbes which are involved in only uniform-shift type transfers. Special types of data permutations, such as perfect shuffle permutation and bit reversal in FFT, do exist. However, more often than not, the variables are of the type considered in this paper. From this and the fact that practically every interconnection network contains the circular connection, the techniques developed here are widely applicable to many problems.

3. OPERAND ALIGNMENT AND COMPUTATION REORDERING

In this section, we shall develop techniques for determining, for a given expression, a sequence of logical data transfers that satisfies the communication requirement of an algorithm with minimum cost. It is assumed that the mapping F_v for every variable v is an identity mapping. Thus the cost function (1) for a parallel expression becomes

$$\sum_{\substack{v_i:\ \text{variable (leaf node)}\\ w_j:\ \text{partial result (internal node)}\\ w_j=\text{PARENT}(v_i)}} D(p_{v_i w_j}) + \sum_{\substack{w_i, w_j:\text{internal nodes}\\ w_j=\text{PARENT}(w_i)}} D(p_{w_i w_j}).$$

To minimize this cost, therefore, the following parameters must be determined:

i. alignment of operands for every binary operation in a parallel assignment statement (PAS), and

ii. an ordering of computation of a PAS.

We shall assume that a compiler is available which generates for PAS's, as intermediate code, a sequence of three-address triples (see [3], for example) on which our minimization is performed. A typical PAS, for example, has the following form:

$A(i) = B(i+2) - C(i-3) * D(i+1) \quad (0 \le i \le N-1)$

Our algorithm will determine, for example, where the operations * and - should take place such that the communication cost of evaluating this PAS is minimal. The logical data transfers, for example,

P_{CW} and P_{DW}, where W denotes the partial result, can then be easily specified.

Some basic definitions are needed.

Definition 1. Let $A(i+k)$ be a variable term in some parallel assignment statement, where k is an integer constant and $-(N/2-1) \le k \le N/2$. Then the displacement associated with this occurrence of variable A is k.

On a circular network of N PEs, the displacement k then refers to the PE that contains the first element, $A(k)$, of the vector $A(i+k)$. The displacement associated with the target variable of a PAS is assumed to be 0, for convenience. If it is not originally zero, an adjustment can be made to the indices of the variables in the PAS such that it becomes zero. We shall use disp(v) to denote the displacement associated with the operand v.

Definition 2. The alignment point (AP): For a binary operation on two operands $A(i+k_1)$ and $B(i+k_2)$, for all i, $0 \le i \le N-1$, if the operation on the $(i+k_1)^{th}$ element of A and $(i+k_2)^{th}$ element of B takes place in the $(i+AP)^{th}$ PE, then AP is the alignment point of that operation.

This value of AP is then the displacement associated with the partial result. The AP of such an operation node n in an expression tree E is denoted by AP(n).

Example 1. For PAS $A(i)=B(i+2)-C(i-3)*D(i+1)$, if AP for the multiplication is 0, then elements of C must be moved from i^{th} PE to $(i+3)^{rd}$ PE and those of D from i^{th} PE to $(i-1)^{st}$ PE. ■

Let k_1 and k_2 be two displacements. Then k_1 and k_2 are the boundary of two intervals on the circular network. Let $I(k_1,k_2)$ denote the interval with shorter length. Then it is easy to see that for a single operation such as $A(i+k_1)+B(i+k_2)$, the communication cost is minimal and equal to $|I(k_1,k_2)|$ if $AP \in I(k_1,k_2)$, where $|I(k_1,k_2)|$ denotes the length of $I(k_1,k_2)$. If, however, this operation is a node in some expression tree E containing more than one operation, then the following result is more general.

Theorem 1. Let $A(i+k_1) + B(i+k_2)$ be an operation in an expression tree E which is not the root of E, and let x denote the alignment point for this operation. Then a necessary condition for the communication cost of evaluating E to be minimal is

$$x \in I(k_1,k_2) \qquad \text{if } 3|I(k_1,k_2)|<N,$$
$$\text{or } x \in I(k_1,k_2) \cup I(k_1-t,k_2+t) \quad \text{otherwise,}$$

where it is assumed that $k_1<k_2$ and $t=N-2|I(k_1,k_2)|$.

Outline of Proof. What needs to be shown is that if x is not in the above interval, then we can always find another alignment point x', which lies in the interval, such that the cost of aligning A and B at x' plus the cost of moving the partial result from x' to x is less than the cost of aligning A and B at x. ■

Theorem 1 simplifies the minimization process greatly because the alignment points are now confined to lie in a small interval. Also, in real problems the displacements are usually small compared with N, i.e., the value $3|I(k_1,k_2)|$ is less than N in most cases. We shall further assume in what follows that $3|I(k_1,k_2)|<N$ for every pair of displacements k_1 and k_2. The necessary condition in this case thus becomes $AP(n) \in [k_1,k_2]$ for every operation node n. (The treatment of the general problem can be found in [5].)

Based on Theorem 1 and the above assumption, we are now able to further reduce the cost function. Suppose o_1 and o_2 are the operands of some binary operation which may be either variables or partial results. Let w denote the result of this operation and let $k_1 = disp(o_1)$ and $k_2 = disp(o_2)$. Then since $AP(w) \in [k_1,k_2]$, on a circular network, the communication cost is $D(p_{o_1w}) + D(p_{o_2w}) = |k_1 - k_2|$, and the cost function for a PAS becomes

$$\sum_{\substack{R,L:\text{brother}\\ \text{nodes in } E}} |AP(R) - AP(L)| + |AP(\text{root of } E)|, \qquad (3)$$

where the term AP(root of E) is needed if E corresponds to a PAS; and what is to be determined is the alignment point for every internal node of E.

This problem is solved using the following algorithm, where, for convenience, we define the alignment point of a leaf node n with displacement k to be $AP(n) = k$, and write $AP(n) \in [k,k]$.

Algorithm 1

```
/* Given E and disp(v) for every variable v in E,
this algorithm determines AP(w) for every internal
node w such that (3) is minimized. */
Traverse the expression tree E in postorder
At each node z,
/* Basic step */
   if z is a leaf node with disp(z)=k,
         i.e., $AP(z) \in [k,k]$
      then $AP(z) = k$;
/* Recursion step */
   if z is an internal node with child nodes x and
         y and $AP(x) \in [x1,x2]$ and $AP(y) \in [y1,y2]$,
         and $W = [x1,x2] \cap [y1,y2]$
      then
         case1. $W = \emptyset$
           /* AP(z) is in the interval "between"
           the two intervals */
             $AP(x) = x2$, $AP(y) = y1$, and
                $AP(z) \in [x2,y1]$ if $x2<y1$
             $AP(x) = x1$, $AP(y) = y2$, and
                $AP(z) \in [y2,x1]$ if $y2<x1$
         case2. $W \ne \emptyset$
           subcase1. $|W|=1$, i.e., $W=\{w\}$ where
                        $w=x1=y2$ or $w=x2=y1$
             $AP(x) = AP(y) = AP(z) = w$;
           subcase2. $|W| >1$
             $AP(x) = AP(y) = AP(z) \in W$;
   if z is the root of E and $AP(z) \in [z1,z2]$,
```

then AP(z) = t where $t \in [z1,z2]$ and the absolute value of t is minimum.

<u>End of Algorithm 1</u>

<u>Example 2.</u> We apply Algorithm 1 to the following PAS

$$A(i) = B(i) + B(i+4) - C(i+3) - D(i+3).$$

The alignment points obtained using Algorithm 1 are shown in Figure 1(a). The communication cost is 4 + 3 = 7. Compared with cost = 10 in Figure 1(b), which is obtained using a compiler without such optimization, this shows a 30% improvement.■

Now note that in (3), the structure of the expression tree, or the order of computation, is to be preserved. This restriction is sometimes necessary, especially in problems where computation reordering may cause loss of significant digits, overflow, or underflow, etc. However, if computation reordering is permitted, then further improvement in the communication cost may be possible. This is discussed below.

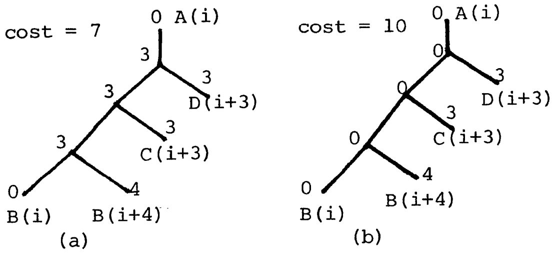


Figure 1. Alignment points for a PAS.

We shall consider computation reordering based on the commutativity and associativity of binary operators. In other words, we shall perform computation reordering only on subexpressions in which all operations have the same precedence.

Let E denote the expression tree for a parallel assignment statement. The precedence of an internal node of E is the precedence of the operator associated with that node.

<u>Definition 3.</u> A subtree T of E is called a p_subtree if all the internal nodes of T have the same precedence and every leaf node of T is either a variable, or an internal node of E which has a different precedence.

The precedence of a p_subtree is the precedence of the internal nodes of the p_subtree.

<u>Definition 4.</u> A maximal p_subtree of E is a p_subtree T of E such that the root of T is either the root of E or the child of a node of different precedence.

Thus a subexpression that corresponds to some maximal p_subtree of E is the basic unit for the consideration of computation reordering.

<u>Example 3.</u> In Figure 2, T_1 and T_3 are maximal p_subtrees of the same precedence. T_2 and T_4 are also maximal p_subtrees of the same precedence. ■

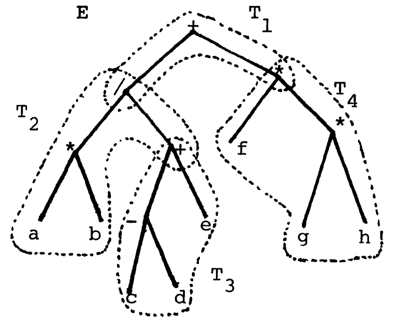


Figure 2. An expression tree and its p_subtrees.

For any expression tree E, the reordering process consists of two phases, the analysis and the synthesis. In the analysis phase, the following tasks are performed:

1. Identify all maximal p_subtrees T in E.

2. For each T, determine the interval in which the optimal alignment point for the root of T, after reordering, must lie.

3. Fix the AP for every node of E.

4. Determine the effective operation to be applied to each variable in E (see below).

The maximal p_subtrees of E can be identified by traversing E in the order of levels. During the traversal, if the node being visited has a precedence different from those already visited, then it is the root of some new maximal p_subtree, which is yet to be identified. For example, the "divide" node in Figure 2 has precedence different from that of addition, and is the root of p_subtree T_2. This can be done using a recursive routine, which,, in addition to identifying the maximal p_subtrees of E, shall also determine the effective operation to be applied to each operand on a maximal p_subtree. For example, for a subexpression A(i) - (B(i+2) - C(i-1)), the effective operations for A, B, and C are, respectively, +, -, and -.

After a maximal p_subtree T is identified, the next task is to determine for T the new order of computation for which the communication cost is minimal. This is done by first determining the alignment point for the root of T. Suppose d_1, $d_2,\ldots,d_k$ are the displacements for the operands $v_1, v_2, \ldots, v_k$, respectively, in T and $d_1 \geq d_2 \geq \ldots \geq d_k$. Then it can be shown [5] that the alignment point for the root, AP(root of T) must lie in the interval $[d_k,d_1]$ to ensure that the communication cost is minimal. This continues until AP(root of E) is obtained. Tree E will then be traversed again for fixing the exact value of the AP for the root of every maximal p_subtree T of E.

In the synthesis phase, new expressions are generated for every maximal p_subtree T of E. Let AP(root of E) = $h \in [d_k,d_1]$, then the new order of computation can be easily obtained. Following is the rule for obtaining the new computation order for which the communication cost is always equal to $|d_1 - d_k|$.

Case 1. $d_k = h$. The new expression is $((\ldots(v_1 \oplus v_2)\oplus\ldots)\oplus v_k)$, where $\oplus$ denotes the effective operation. The AP's of these operations are, respectively, $d_2,\ldots,d_k$.

Case 2. $d_1 = h$. The new expression is $((\ldots(v_k \oplus v_{k-1})\oplus)\ldots)\oplus v_1)$. The AP's are $d_{k-1}, d_{k-2},\ldots,d_1$.

Case 3. $d_k < h < d_1$. Let t be such that
$d_{t+1} < h \le d_t$.
The new expression is
$(\ldots(v_1 \oplus v_2)\oplus\ldots\oplus v_t) \oplus ((\ldots(v_k \oplus v_{k-1})\oplus\ldots)\oplus v_{t+1})$.
The AP's for the left half are $d_2,d_3,\ldots,d_t$, and for the right half are $d_{k-1},\ldots,d_{t+1}$; and the AP for the root of the new expression tree is h.

As an example, the PAS of Example 2 can be reordered as

A(i) = ((B(i+4) - C(i+3)) - D(i+3)) + B(i),

for which the cost is only 4, 40% of the original cost and almost half of the cost obtained from using only Algorithm 1.

Having determined the alignment point for every operation in an expression, one can easily specify the logical transfer functions that should be applied to the operands, and the sequence of logical transfers for the entire algorithm is obtained. In the next section, this sequence will be the data for obtaining the optimal mapping and remapping for the variables in the algorithm.

4. DATA MAPPING AND REMAPPING

In [11], two kinds of parallel algorithms are considered for applying data mapping. In one, the algorithm is assumed to have only one logical transfer. For example, the matrix transposition may be realized using only the perfect shuffle permutation. For this type of algorithms, the optimal mapping for the data can easily be obtained [11, Theorem 1]. In the second kind of algorithms, the logical transfers of a given variable are not all the same; for example, the bitonic sort on a mesh-connected network uses several different transfers. The minimization problem for this type of algorithms is usually difficult. A technique used in [11] for the bitonic sort is to determine an optimal mapping for the most expensive transfer, which is certainly not always the least expensive mapping.

In this section, we shall first study the minimization problem for the second kind of algorithms with the restriction that logical transfers are all uniform shifts but may have different shift distances, and that the mapping is static. In section 4.2, we discuss how to improve the cost further by allowing remapping of data during the execution of the algorithm.

4.1 Static Mapping

The problem is the following: given a parallel algorithm and a circular network, determine for each variable of the algorithm a static mapping such that the communication cost of operations involving these variables is minimal. The logical transfers are assumed to have been obtained using the techniques of the previous section.

First formulate the cost function. Let F_i be the mapping function for a variable v_i and $F_i(k) = (k + x_i)$, where $0 \le k \le N-1$ and x_i is to be determined. The cost of evaluating a PAS can be derived as follows. Let E be the expression tree. Then, from section 3, the cost at the logical level is simply

$$\sum_{\substack{R,L:\ \text{brother}\\ \text{nodes in E}}} |AP(R) - AP(L)| + |AP(\text{root of } E)|$$

If non-identity mapping is allowed for variables, then the cost of any alignment operation involving a variable becomes dependent on the mapping function for the variable. For partial results, however, the cost is a function of only their associated alignment points.

The cost of an alignment operation involving a variable is illustrated in Figure 3. The elements of v_i, $v_i(k+AP(i))$, originally stored in memory $F_i(k+AP(i))$, will now be moved to memory $(k+AP(PARENT(i)))$ where the partial result shall reside. So the cost of this operation is

$$|F_i(k+AP(i)) - (k+AP(PARENT(i)))|$$
$$= |x_i - (AP(PARENT(i)) - AP(i))|.$$

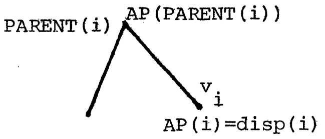


Figure 3. An alignment operation.

To determine the optimal mapping for v_i, all such terms involving x_i have to be collected. Let r_i denote the total number of logical transfers involving v_i, and let d_{ik} denote the value AP(PARENT(i)) -AP(i) in the kth transfer of v_i. Then the total cost due to variable v_i can be written as

$$\sum_{k=1}^{r_i} |x_i - d_{ik}| \qquad (4)$$

which is minimized by setting

$$x_i = \text{median of } \{d_{ik} \mid 1 \le k \le r_i\}.$$

Example 4. Suppose the displacements associated with variable v_i in an algorithm are

$AP(i) = disp(i)$: 2 0 -1 0 -1

and the alignment points for the operations involving v_i are

$AP(PARENT(i))$: 0 2 1 1 2,

then the shift distances for v_i are

d_{ik}: -2 2 2 1 3

and the median of $\{d_{ik}\}$ is 2.

Then the cost using the mapping $F_i(k) = (k + 2)$ is 6, while the cost using the identity mapping would be 10, indicating a 40% improvement. ■

It is easy to see that the improvement would be high if the sequence of transfers is long and the median is far from zero. If the median is equal to zero, then the mapping is simply the identity. It is also easy to note that if the sequence is long, perhaps one can perform several remapping during the execution of the algorithm so that the communication cost is further reduced. This data remapping is discussed below.

4.2 Remapping

In [10], remapping techniques are applied to a class of parallel algorithms which are assumed to be executed on an SIMD machine using a shuffle-shift interconnection network. Most of these algorithms have logrithmic computation but higher communication complexity. After remapping, many of the algorithms are balanced. This analysis again indicates that software techniques are a more flexible tool for providing better solution to the improvement of the performance of parallel algorithms. In this section, we study the remapping problem under the assumed environment.

In general, the major task in the use of data remapping is the determination of when and how to perform a remapping so that the new cost, the communication cost plus the cost of remapping itself, is less than the cost of using only static mapping.

In what follows, we shall use L to denote the sequence of all logical transfers of a variable v that are required by the given parallel algorithm, i.e., $L=d_1, d_2,\dots,d_k$ where d_i is the shift distance for v in the ith transfer of v. A remapping schedule for v is defined as a division of the sequence L into subsequences, and within a subsequence only a static mapping function for v is used, and a remapping of the data must be performed before a new subsequence which uses a different mapping function starts. Clearly, the mapping function for v within a subsequence should be the optimal static mapping for v with respect to this subsequence, which can be obtained using the result of section 4.1. A remapping schedule for v is said to be optimal if the total cost with respect to this schedule is the least among all possible schedules.

Let the interval defined by two consecutive shift distances d_i, d_{i+1} of L be denoted by I_i, i.e., $I_i = I(d_i,d_{i+1})$. Let I_L denote the intersection of all intervals I_i, $1 \le i \le |L|-1$, in L, i.e.,

$$I_L = \bigcap_{i=1}^{|L|-1} I_i,$$

where $|L|$ denotes the number of elements in L.

Following are some of the properties of a sequence L for which $I_L \neq \emptyset$.

Lemma 1. Let $I_L = [a,b]$. If $a < b$, then d_M, the median of L, is given by

$$d_M = \begin{cases} a & \text{if } d_1 \le a \text{ and } d_k \le a, \\ b & \text{if } d_1 \ge b \text{ and } d_k \ge b, \\ x, & \text{where } x \in [a,b], \text{ otherwise.} \end{cases}$$ ■

If $I_L = [a,a]$, then it is easy to see that $d_M = a$ if $|L|$ is odd, and, if $|L|$ is even, both d_M and a must lie in the interval defined by the two median numbers of L. In the latter case, we shall set $d_M = a$, so that in all cases $d_M \in I_L$.

In the following Lemmas, $C(L,d_M)$ denotes the communication cost resulting from the static mapping $F_v(i) = (i + d_M)$ for v with respect to L, i.e.,

$$C(L,d_M) = \sum_{i=1}^{k} |d_M - d_i|.$$

Let L' denote any subsequence of L having s elements. Let c_m denote the median of L'.

Lemma 2. Suppose $I_L \neq \emptyset$ and d_M has been fixed usign Lemma 1. If s is even, then it is possible to choose c_m such that

$$C(L',c_m) = C(L',d_M),$$

and if s is odd, then

$$C(L',d_M) = C(L',c_m) + |d_M - c_m|. \quad ■$$

Lemma 3. Let L_j and L_{j+1} be two consecutive subsequences of L, where $|L_j|=s_j$ and $|L_{j+1}|=s_{j+1}$. Let their medians be denoted d_m^j and d_m^{j+1}, respectively. Let L_j' denote the concatenation of L_j and L_{j+1}. If $I_L = [a,b] \neq \emptyset$, then

Case i. s_j, s_{j+1} both even.

d_m^j and d_m^{j+1} can be chosen such that

$$C(L_j,d_m^j) + C(L_{j+1},d_m^{j+1}) = C(L_j',d_M).$$

Case ii. s_j odd, s_{j+1} even.

d_m^{j+1} can be chosen such that

$$C(L_j,d_m^j)+C(L_{j+1},d_m^{j+1})+|d_M - d_m^j| = C(L_j',d_M).$$

Case iii. s_j, s_{j+1} both odd.

If $d_m^j \le d_M$ ($d_m^j \ge d_M$), then $d_m^{j+1} \ge d_M$ ($d_m^{j+1} \le d_M$). And

$$C(L_j, d_m^j) + C(L_{j+1}, d_m^{j+1}) + |d_m^{j+1} - d_m^j| = C(L_j^!, d_M). \blacksquare$$

In Lemma 4, $|d_M - d_m^j|$ and $|d_m^{j+1} - d_m^j|$ are the cost of remapping. This Lemma shows that if $I_L \neq \phi$, then no gain will be achieved by dividing L into two subsequences. A generalization of this Lemma is very useful and is given below.

Theorem 2. If $I_L \neq \phi$, then no remapping schedule $\mathscr{L}$ for v will result in a communication cost less than $C(L, d_M)$. ■

Theorem 2 implies that, as a first step, one should determine a sequence of subsequences of L, i.e., determine $\mathscr{L} = L_1, L_2, \ldots, L_t$, such that every L_i is maximal in the sense that $I_{L_i} \neq \phi$ and the inclusion of either two elements of L that precede L_i ($i \neq 1$) or the two elements that follow L_i ($i \neq t$) will result in $I_{L_i} = \phi$. This can be achieved using Algorithm 2 below.

Algorithm 2

```
/* given sequence of logical transfers (or shift
distances) in L(1:k).  F_X(1:k), L_X(1:k) store
start and end indices of subsequences.
L_MEET(1:k), H_MEET(1:k) low and high bounds of
intersection interval for Lj. Q_X: index for the
number of the subsequence Lj. [m1,m2]: current
intersection interval. [t1,t2] = [L(i), L(i+1)]:
next interval to be processed. */

if  k ≤ 2  then stop; /* no remapping necessary */
Q_X = 1;  /* for L1 */
F_X(1) = 1;  /* start with d1 */
m1 = min(L(1),L(2)); /*di = L(i) */
m2 = max(L(1),L(2)); /* [m1,m2] */
i = 2;  /* next interval is [L(2),L(3)] */
Flag = '1'B;  /* until [m1,m2] = ø */
do while (i < k);  /* L(2), L(3),..,L(k) */
   t1 = min(L(i),L(i+1)); t2 = max(L(i),L(i+1));
   /* test if intersection is empty */
   call CHK_MEET(m1,m2,t1,t2);
   if ¬Flag then do;  /*end of this sebsequence */
      L_X(Q_X) = i - 1;  /*index of last item */
      L_MEET(Q_X) = m1; /*intersection interval */
      H_MEET(Q_X) = m2;  /* of this subsequence */
      Q_X = Q_X + 1; /* next subsequence */
      F_X(Q_X) = i;  /* start of Lj+1 */
      j = m1;
      m1=min(m2,L(i)); /* new starting [m1,m2] */
      m2 = max(j,L(i));
      Flag = '1'B;
   end;  /* of if then do; */
   else  i = i + 1; /* just get next two items */
end;  /* of do while */
/* for last subsequence */
L_X(Q_X) = i; L_MEET(Q_X) = m1; H_MEET(Q_X) = m2;

CHK_MEET: Procedure (m1,m2,t1,t2);
 /* compute intersection of [m1,m2] & [t1,t2] */
 /* set Flag to '0'B if it is empty */
 if t1 > m2   |   t2 < m1
      then do; Flag = '0'B;  return;  end;
 if t1 = m2 then do;  m1 = t1;  return; end;
 if t2 = m1 then do;  m2 = t2;  return; end;
 m1 = max(m1,t1);    m2 = min(m2,t2);
end CHK_MEET;
```

End of Algorithm 2

After obtaining $\mathscr{L}$ using Algorithm 2, the next step is to fix the median of every subsequence L_i. If $|L_i|$ is odd, then d_m^i is simply the median of L_i; but if $|L_i|$ is even, then d_m^i can be any value in the interval defined by the two median elements of L_i. For convenience, we shall call such an interval the median interval. Following is the rule for determining the median of every L_i such that the resulting remapping schedule is optimal.

Median Selection Rule

Given $L = d_1, d_2, \ldots, d_k$, obtain $\mathscr{L} = L_1, L_2, \ldots, L_t$ using Algorithm 2.

i. if $|L_i|$ = odd, then d_m^i = median of L_i.

ii. if $|L_i|$ = even and if the median interval of L_i is of the form [w,w], then $d_m^i = w$.

iii. for every i, $1 \le i \le t$,
    if d_m^i is not fixed by rule i or ii and
        $d_m^i \in [lo,hi]$, then
            $d_m^i = lo$ if $d_m^{i+1} < lo$
            $d_m^i = hi$ if $d_m^{i+1} > hi$

End of Median Selection Rule

(Note that median intervals of any consecutive subsequences never overlap.)

Example 5. Suppose for some variable v, L = 1, 3, 4, 5, 6, 1, -2, -1, 0. Applying Algorithm 2 to L, we get $\mathscr{L} = L_1, L_2, L_3, L_4, L_5$, where L_1= 1,3, L_2= 4, L_3= 5,6, L_4= 1, L_5= -2,-1,0. We can first fix the following medians: $d_m^2=4$, $d_m^4=1$, and $d_m^5=-1$. Then using the Median Selection Rule, we obatin the medians: $d_m^1=3$, $d_m^3=5$. It is easy to compute the following communication costs:

$$C(L, d_M) = C(L, 1) = 20, \text{ and}$$

$$\sum_{i=1}^{5} C(L_i, d_m^i) + \sum_{i=2}^{5} |d_m^i - d_m^{i+1}|$$

$$= [(2+0)+0+(0+1)+0+(1+0+1)] + [1+1+4+2] = 13.$$

Thus, with remapping, the cost due to variable v is further improved by more than 30%. ■

Algorithm 2, along with the Median Selection Rule, generates an optimal remapping schedule for any sequence L. The optimality proof can be found in [5].

Theorem 3. Algorithm 2 and Median Selection Rule constitute an optimal algorithm for generating the remapping schedule for any logical transfer sequence. ■

As another more practical example, consider the Jacobi algorithm for computing the eigenvalues of real symmetric matrices [13]. In the classical Jacobi algorithm, a real symmetric matrix is reduced to the diagonal form by a sequence of elementary orthogonal transformations. In [9], the algorithm is modified for parallel computation on Illiac IV machine. In this implementation, the data broadcasting capability of the network is an essential requirement. Otherwise, the data communication cost would become very high and outweigh the gain obtained from parallel computation. In [4], the matrix multiplication by diagonal scheme [12] is used to compute the transformation, which requires no data broadcasting, so the algorithm can be implemented on a circular network. A further modification to the algorithm is given in [7], which also uses the multiplication by diagonal scheme. In both [4] and [7], the transformation has the form $\phi A\phi^t$, where $A = [a_{ij}]$ is the given NxN matrix and $\phi = [t_{ij}]$ is the transformation matrix which causes elements $a_{i,i+1}$ and $a_{i+1,i}$, i even, to be eliminated. From the multiplication scheme, it is easy to see that for both t_{ii} and $t_{i-1,i}$, i odd, the sequence of logical data transfers required to complete every computation of $\phi A\phi^t$ is $L = 0,0,1,2,3,\ldots,N-2$. Therefore, we can apply our algorithm to L and obtain the following remapping schedule:

$$\mathscr{L} = (0,0), (1), (2), (3), \ldots, (N-3,N-2).$$

In other words, these two data vectors are remapped one PE down the circular network every computation step (except for the first and last subsequences). The communication cost using this remapping is clearly $O(N)$ per transformation, while if only static mapping is used, the cost would be $O(N^2)$.

For this particualr example, although the above remapping schedule may have been obtained by carefully examining the multiplication scheme, or by using the data buffering technique of [10], with our method and Theorem 3, the optimality of the schedule is guaranteed.

5. CONCLUSION

The problem of minimizing the communication cost in the implementation of a parallel algorithm on an SIMD computer is discussed. For a given parallel algorithm, techniques have been developed for determining the order of computation of an expression, the alignment of operands for every binary operation, the mapping of data to the physical memories, and data remapping such that the communication time is minimized. This analysis, as well as other previous work, indicate that software techniques are sometimes a more flexible tool for providing better solution to the improvement of performance of parallel algorithms than hardware, which is often limited by the cost and complexity.

REFERENCES

[1] T. Agerwala, "Communication, computation, and computer architecture," Int'l Conf. on Communication, June 1977, 209-215.

[2] T. Agerwala, B. Lint, "Communication in parallel algorithms for Boolean matrix multiplication," Int'l Conf. on parallel Processing, 1978.

[3] A. V. Aho, J. D. Ullman, Principles of Compiler Design, Addison-Wesley, 1977.

[4] W. H. Bernhard, Illiac IV codes for Jacobi and Jacobi-like algorithms, Center for Advanced Computation Doc. No. 19, Univ. of Illinois, Nov, 1971.

[5] K. Chen, Ph.D. dissertation, in preparation.

[6] K. Chen, K.B. Irani, "Mapping problem and graph numbering," in Proc. of the Workshop on Interconnection Networks for parallel and distributed processing, April 21-22, 1980, 41-46.

[7] K. Chen, K. B. Irani, "A Jacobi algorithm and its implementation on parallel computers," in Proc. of 18th Annual Allerton Conference on Comm., Control, Computing, Oct. 1980.

[8] W.M. Gentleman, "Some complexity results for matrix computation on parallel processors," JACM 25, 1(1978), 112-115.

[9] D.J. Kuck, A.H. Sameh, "Parallel computation of eigenvalues of real matrices," in Information Processing 71 Proc. IFIP Congress 71, VOL. II, North-Holland Publ. Co., Amsterdam, The Netherlands, 1972, pp. 1266-1272.

[10] R. H. Kuhn, Optimization and Interconnection Complexity For: Parallel Processors, Single-Stage Networks, and Decision Trees, Dept. of Comp. Sci., Univ. of Illinois, 1980.

[11] H.T. Kung, D. Stevenson, "A software technique for reducing the routing time on a parallel computer with a fixed interconnnection network," in High Speed Computer and Algorithm Organization," (D.J. Kuck et al. editors) Academic Press, N.Y. 1977.

[12] N. Madsen, G. Rodrigue and J. Karush, "Matrix multiplication by diagonals on a vector/parallel processor," Information Processing Letters, vol. 5, June 1976, 41-45.

[13] J. H. Wilkinson, The Algebraic Eigenvalue Problém, Claradon Press, Oxford, 1965.

PARALLEL HASHING HARDWARE FOR TEXT SCANNING APPLICATIONS

F. J. BURKOWSKI

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF MANITOBA
WINNIPEG, MANITOBA, CANADA

ABSTRACT

This paper discusses the hardware design of a term detection unit which is to be used in the scanning of text emanating from a serial source such as disk or bubble memory. This unit will provide the highly parallel activity necessary for the detection of any one of many terms (eg. 1024 terms) while accepting source text at a transfer rate typical of disk technology, for example, one megabyte per second. The design incorporates a hardware-based hashing scheme which allows the incoming text to be compared with selected terms in a RAM which contains all of the strings to be detected. Since the speed of operation is such that any lengthy probe sequence cannot be tolerated, the design involves mechanisms which strive to provide a perfect hash; perfect in the sense that a probe sequence need not be used to overcome the collision problem.

INTRODUCTION

In many text retrieval systems the data base is comprised of a huge collection of documents which are essentially unstructured in their organization. Indexing facilities may be rather limited in scope and hence not sufficient to meet the requirements of queries which incorporate arbitrary terms or keywords. Examples of such data bases include newspaper repositories, legal decisions, journal articles, military intelligence reports, and even smaller data bases for corporate offices (see [1]). Since an indexing procedure cannot effectively narrow down the area to be searched, the retrieval strategy may involve a serial scan of the entire data base or some significant portion of it.

Typically, user enquiries are analyzed by query translator software which extracts from the user queries all of the various terms which must be detected in the source text. For example, a typical query might be the concise version of a command such as: "Retrieve all documents which contain any 3 of the following 6 terms: BRASS, TRUMPET, SACKBUT, CORNETT, SERPENT, TENOR CURTAL". The query translator passes a list of such terms to the term detection unit which will scan the source text and determine the "document location" of each term if it exists in the text. This information is passed to query resolution software which will determine whether the scanned document meets the criteria imposed by any user query.

If a system is subject to enquiries from many users then a typical scan may involve a parallel search for hundreds of terms. This search is to be accomplished while the source text streams by at a rate determined by the disk transfer rate of approximately one megabyte per second. This type of search is best done by a term detection unit which is specifically designed for such highly parallel activity. This paper will discuss the hardware design of such a term detection unit. By using a cost effective and novel approach, the unit is capable of handling a great many terms. If the reader is interested in other approaches he may consult [2] or [3] wherein the author discusses the problems associated with these applications and presents a survey of some of the architectures which are of current interest.

TERM DETECTION HARDWARE

The term detector accepts a serial stream of characters which are shifted into a serial-in parallel-out shift register SR capable of holding k characters. Each shift operation causes the input sequence to be shifted one character position. The parallel-out lines of SR provide input to a set of comparators which are also fed by the data lines of a RAM with an organization of M=2**m words each 8*k bits wide. In a text retrieval system, one might expect values such as k=32 and M=256.

Using this approach a k character substring of the source text can be compared with any one of M words in the RAM. Each byte of RAM would contain a 7 bit ASCII code and an additional bit used to signify a "don't care" or unconditional match character.

Since we must effectively compare all the M terms with the current source substring of k characters in the time interval between shifts, it is tempting to use an associative memory in place of the RAM and comparator, but considering the current prices of associative memory such a solution is prohibitively expensive. Consequently, we retain the RAM and use in conjunction with it a "mapping module" which accepts a subset X of the data-out bits from the shift register and computes H(X) an m-bit address which will select from the RAM one term which is to be compared with the source text in the shift register (see Fig. 1). The mapping module will be a high speed random access memory with address inputs determined by the X-lines and data-out lines providing the m-bit H(X) value.

0190-3918/81/0000/0282$00.75 © 1981 IEEE

We now describe the strategy involved. If the shift register does not contain a required term then a comparison will be initiated with some selected term in the RAM (from a practical point of view, it does not matter which term is chosen) and since a match is not detected the shift activity simply resumes. If the shift register contains a target term which matches some term in the RAM, then we must ensure that the RAM address of the matching term is equal to the value H(X) generated by the mapping module when the target term in the shift register is in alignment with this selected term in the RAM. This will give an active level on the equality output of the comparator when the selected term is compared with the contents of the shift register. The generation of the appropriate H(X) value is relatively easy to derive unless the number of terms is very large compared to the number of X-lines. Note that a uniqueness property must be enforced. If we consider any target term so situated in the shift register that it is aligned with its matching term in the RAM, the binary value produced on the X-lines must be unique i.e. different from the X-line value of any other target term in its aligned position. In effect, H(X) must generate a perfect (no collision) hash function for the set of all target terms in their aligned positions. The crucial point of the design is that the attainment of such a perfect hash is greatly enhanced by the fact that there is often some freedom of choice in selecting the X-value which is generated by the aligned target term. This is true since most terms will not be the full k characters in length and consequently a term can usually adopt one of many positions relative to the character positions responsible for the definition of the X-lines. Note that the appearance of a term within its word of RAM is constrained to a position such that the aligned target must cover those positions of the shift register which define the X-lines.

In order to illustrate the above discussion with an example, consider the following simplified situation: Suppose a search is being done for the following four terms, "MARS", "MARTIAN", "STAR", "ARTIST". Also, suppose for the sake of illustration, that there are 14 X-lines defined as the middle two characters in the shift register. In actual practice, it is likely that we would use fewer lines defined by more characters, say the middle four character positions. With these assumptions the X-line outputs for "MARS" would be "MA", "AR" or "RS". The given terms and their corresponding X-value candidates may be portrayed by a bipartite graph as illustrated in fig. 2. If it is possible to define for this graph a maximal matching which covers all the points in the "term set" then it is possible to find the distinct X-line values that will be needed. In fig. 2 one of the many maximal matchings is represented by the heavy lines. The algorithm which derives such a maximal matching can be found in [4] (the Hungarian method) and a faster version has been recently presented in [5]. Note that the graph illustrates a certain amount of competition for particular X-values such as "AR". As one might expect, the overall competition for distinct X-values will be augmented if the number of terms is increased and/or the number of X-lines is decreased. Using the algorithms presented in [4] or [5] a maximal matching can be extracted rapidly since the graphs that arise in text scanning are rather easy to match. If the maximal matching does not completely cover the term set then the uncovered terms are rejected from the batch and must wait for the batch being accumulated for the next scan of the data base.

Simulation studies have been carried out in an effort to predict the severity of this rejection problem. The following table presents the results:

Bits Out h	Window Size n	Minimum Number Rejects	Maximum Number Rejects	Average Number Rejects
8	2	80	96	88.29
	3	46	70	57.46
	4	38	67	51.55
10	2	2	26	14.20
	3	0 (78)	3	0.36
	4	0 (20)	9	2.34
12	2	0 (8)	22	10.30
	3	0 (82)	3	0.26
	4	0 (54)	5	0.92

RAM Implementation M=256 100 trials

In this table h represents the number of X-lines taken from the n middle characters of the shift register. In each case 100 trials were run with each trial involving a batch of 256 terms. For example, when 12 X-lines were defined using the four low order bits from each of the three character positions at the middle of the shift register then for the 100 trials 82 ran with 0 rejection (perfect hashing for all terms) while the other 18 required rejections but not more than 3 terms were ever rejected. The average number of rejections was 0.26 term. Terms were randomly selected and varied in length between 4 and 32 characters. The lengths formed a normal distribution with an average of 8.5.

MINIMIZING THE REJECTION PROBLEM

In [6] simulation studies indicate that the ASCII encoding of the source text is not necessarily the best choice.

Since the previous design works with low order bits of each character, the characters responsible for defining the X-lines may have the same appearance since their X-line outputs are identical. For example, the ASCII codes for the four vowels "A", "E", "I", and "U" all end with the bits "01" and hence, if the X-lines use only two low order bits these frequently occurring characters are not distinguished from one another. The selection algorithms for the previous design would have a better performance (on the average, fewer rejections per batch) if the binary codes representing the characters were distributed in such a way that the codes for characters used most frequently (in English language source text) presented the most variety in their low order bits. It can be demonstrated that for certain values of n and h the average rejection rate can be cut in half by using this more suitable encoding of the source text.

The above strategies are sufficient for term detection units which handle 256 or fewer terms. Unfortunately, if n and h are kept fixed, the rejection rate does not rise in a linear fashion as the number of terms increases. For example, a unit designed to handle 1024 terms with h=12 and n=4 must suffer an average of 85.58 term rejections. This many rejections could adversely affect many users on such a system. Various design techniques may be used in an effort to minimize term rejection. The most obvious approach is to increase h the number of X-lines. However, since each additional bit doubles the amount of RAM used in the mapping module it may be more profitable to investigate other approaches to the problem.

Perhaps the most effective alternative is a design which tends to alleviate the constraint imposed by the uniqueness requirement discussed earlier. In fig. 3 and fig. 4 designs are presented which allow two terms to share a common X-value. In the first case an extra field in the string RAM is used as a pointer to a second term which may also be compared with the contents of the shift register. Thus, an extra probe is done in a sequential fashion. Note that in the time between shifts (one microsecond) the following delay times are experienced: the delay in the shift register (40 ns.), the access time for the mapping module (100 ns.), the access time for two string RAM reads (400 ns.) and the propagation delay through the comparators (60 ns.).

In fig. 4 an alternative scheme effectively permits two parallel probes to be done simultaneously. There is extra hardware, consisting of two mapping modules and two sets of comparators but the performance is faster.

In both of these schemes the rejection problem is greatly alleviated as demonstrated by the table below which presents the results of 50 trials each working with 1024 terms.

		Perfect Hashing		Extra Probe Allowed	
h	n	min:max	avg	min:max	avg
12	2	537:682	632.6	236:423	360.0
12	3	49:215	153.7	0:17	2.4
12	4	39:140	85.6	0:16	4.0
14	2	537:682	632.6	236:423	360.0
14	3	36:178	117.1	0:16	1.4
14	4	15:101	49.2	0:9	1.6

Term Rejections for Batches with 1024 Terms

FUTURE DIRECTIONS

The derivation of the required maximal matching has been accomplished using graph theoretic algorithms which are guaranteed to produce results in all cases even though the general problem has a very high "worst case" complexity. However, text scanning graphs are easy to match and hence it would be well worthwhile to look for alternate algorithms in an effort to satisfy other objectives more important to the application, for example:

1. Since a user is effectively rejected if any of his or her terms are rejected, it is more important to have an algorithm which rejects the *minimum number of users*.

2. In the design discussed above, all user terms are collected and presented in a batch to the term detector just prior to a complete scan of the data base, which may take hours in some applications. Response time for a user would be improved if his or her terms could be presented to the term detection unit very soon after query translation, perhaps when the disk head moves from the current track to the next track. This dynamic addition and deletion of user terms would have a significant impact on the design of the algorithm used to extract the matching.

REFERENCES

1. R. V. Dickinson, "The SDC Records Manager: A VLSI Based Text Retrieval and Communications System", *Compcon 81*, (Feb. 1981), pp.115-118.

2. M. J. Foster and H. T. Kung, *Design of Special-Purpose VLSI Chips: Example and Opinions*, Tech. Rep. CMU-CS-79-147, Department of Computer Science, Carnegie-Mellon University.

3. L. A. Hollaar, "Text Retrieval Computers," *Computer*, (Mar., 1979), pp.40-50.

4. J. A. Bondy and U. S. Murty, *Graph Theory With Applications*, MacMillan Press Ltd., (1976), pp. 70-90.

5. J. Hopcroft and A. Karp, "An n**(5/2) Algorithm for Maximum Matchings in Bipartite Graphs", *SIAM J. Computing*, (Feb., 1973), pp. 225-231.

6. F. J. Burkowski, "Text Scanning Via Maximal Matchings on Bipartite Graphs", To appear.

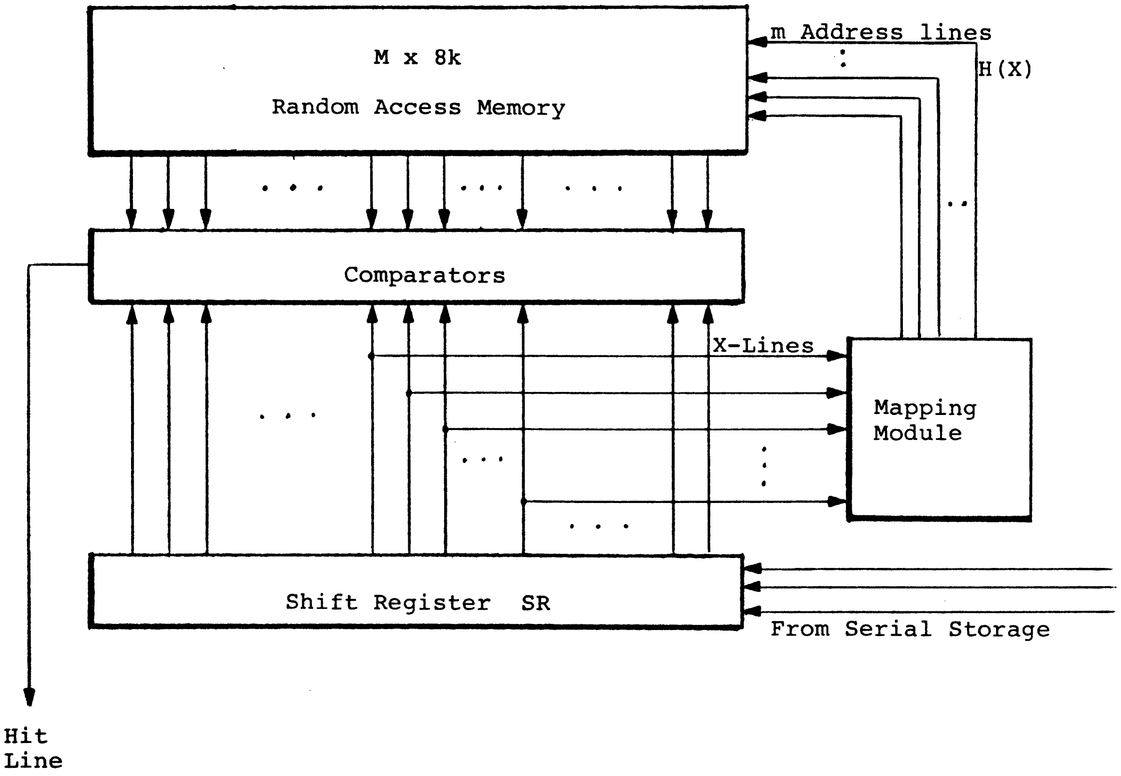


Figure 1. Term Detector with RAM Storage

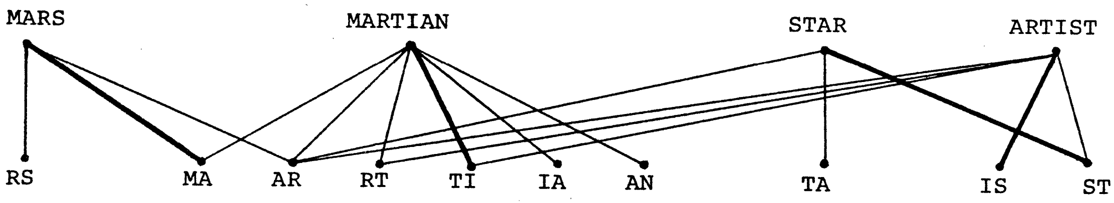


Figure 2. Four Terms and the Corresponding Bipartite Graph for n=2

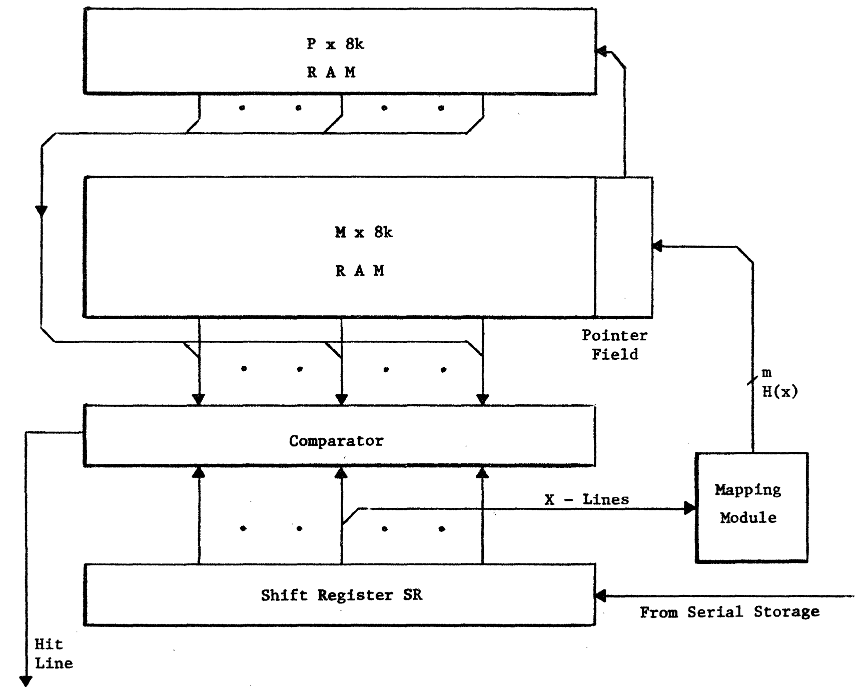


Figure 3. Term Detector with Auxiliary String RAM

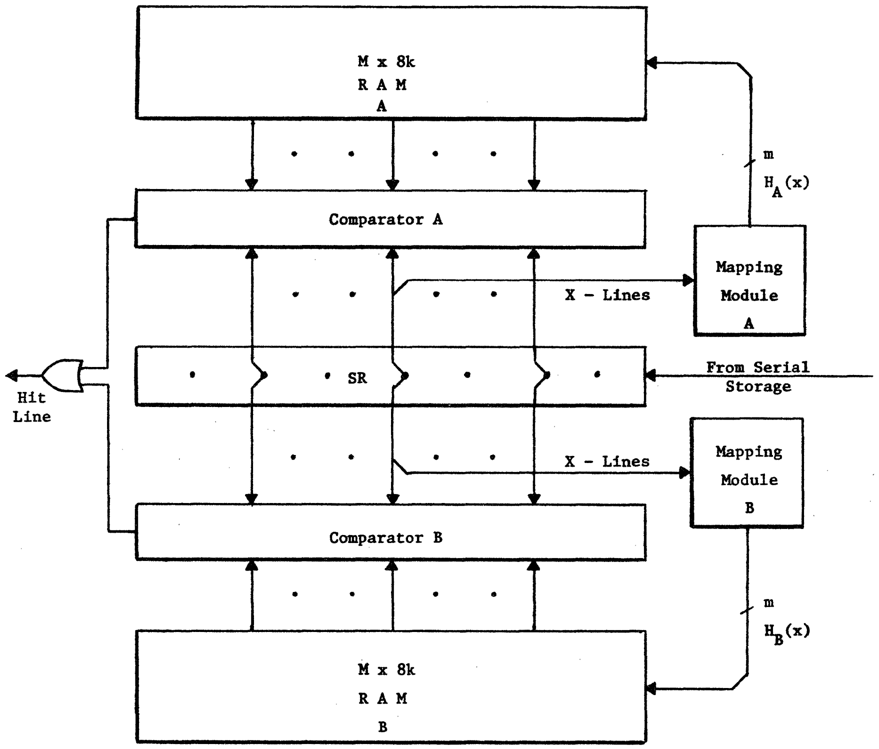


Figure 4. Term Detector with Parallel Activity

A PARALLEL PROCESSOR ELECTRONIC TARGET SIGNAL GENERATOR FOR ELECTRO-OPTICAL SEEKERS

T. N. Long, J. T. Randolph, and M. J. Sinclair
Engineering Experiment Station
Georgia Institute of Technology
Atlanta, Georgia 30332

Summary

The first phase of implementation of a flexible, programmable, computer controlled Electronic Target Signal Generator (ETSG) has been completed. This scene generator will be incorporated into hybrid closed-loop simulations performed at the United States Army Missile Command's Advanced Simulation Center in Huntsville, Alabama. Using parallel processing techniques, the ETSG can now generate real-time signals equivalent to the two spectral channel detector outputs of a Passive Optical Seeker Technique (POST) seeker such as Stinger/POST. A future phase of implementation will permit simulation of reticle type seekers such as Redeye or Stinger. A total of six targets within the field of view can now be simulated. Expansion capabilities allow for a total of nineteen (19) targets.

The Electronic Target Signal Generator (ETSG) is a self-contained parallel processing computer which, when given the proper initial and dynamic inputs, generates an analog voltage that simulates the detector output or outputs of an electro-optical seeker. Simulated sources of specified shape, size, spatial orientation, and intensity gradients can be created and controlled within the missile's field of view. Simulation of a target/background/countermeasure scenario can then be accomplished by selecting various sources and combining them to represent the various parts of the target signature. A high level block diagram of the system is illustrated in Figure 1.

During initialization, constants which describe a particular seeker type and desired source properties are entered through a color graphics terminal. Using these inputs (Table 1), a reference Random Access Memory (RAM) block is loaded for each source to avoid unnecessary calculations during real-time source updates. This loading is performed by an initialization processor which is a commercially available Motorola EXORcisor® that contains a 6800 micro-processor, 48K-bytes of RAM, dual floppy disks, and a serial interface. Source reference RAM's are 64 x 64 blocks of eight-bit bipolar memory which contain the highest resolution "map" that the source can have during a mission. Presently, two of the sources may be designated as "complex", with three orthogonal views being stored in RAM. This configuration, which can be expanded to include all sources, permits fly-around during a simulation.

For each real-time update of an engagement, a high-speed data transfer is initiated by the digital computer which dictates source spatial orientations. Upon completion of the transfer to the ETSG, source central processing units (CPUs) are halted and loaded with their respective dynamic data (Table 1). These CPUs (one per source) are also built around the 6800 microprocessor and use a 16 x 16-bit multiplier to reduce computation time.

INITIALIZATION PARAMETERS

SEEKER
Type (POST or reticle)
Field of View Size
Blur Size
NEFD
SNR for Tracking
Scan Rate (reticle only)

FLARE
Intensity vs Time

ENVIRONMENT
Background
Atmospheric Attenuation

SOURCES
Shape
Size
Aspect Ratio
Intensity Gradient
Spectral Band
Intensity Polarity
Intensity Program
Maximum Range
Minimum Range

PULSE JAMMER
Repetition Rate
Sweep Time
Duty Cycle
Period

DYNAMIC PARAMETERS

Block Transfer Code Word
Range
Azimuth
Elevation

Aspect
Rotation
Flare and Jammer Control

Table 1. Simulation Parameters

Each 64 x 64-byte reference RAM represents a source at the range where it exactly fills the field of view in at least one dimension. Also, the RAM represents the source viewed with no projection. Accordingly, a source with a non-zero aspect cosine will be smaller than its reference. Because of these principles, any source can be created simply by skipping points in reference RAM as it is loaded into a memory array which represents the image plane. The source CPUs calculate these skip factors.

Upon completion of the parallel calculations of the CPUs for each update, loader circuitry begins loading one of two image-plane "maps" (64 x 64-byte blocks of twelve-bit bipolar memory) in each channel. The maps are loaded from the reference RAMs using the values calculated and stored in the latches by the source

0190-3918/81/0000/0287$00.75 © 1981 IEEE

CPUs. As the sources are loaded into an image-plane map, the intensities from the reference RAMs are contrasted with the background and scaled for range (with atmospheric effects included) by a floating-point multiplication.

As sources are being loaded into one of the image-plane maps of a channel, points previously loaded in the other map of that channel are being summed for output in a method which depends on the type of seeker being modeled. In the POST (flying spot) system, a relatively small window (3 x 3, 2 x 2, or 1 x 1 bytes) is scanned in a pattern through the field of view. The intensities of the points which fall within the window are summed and output as a single time sample during the sequential sampling. The functions of the two image-plane maps in each channel are swapped for each update so that one map is always being filled while the other is being scanned and sampled.

After the points have been summed, the background level and detector noise are added with the result being converted to an analog signal and filtered to remove sampling noise. This analog signal represents simulated detector output.

Off-line diagnostics are performed by the initialization processor. A significant problem is encountered when using a 64K-byte machine to address over 200K bytes of memory. This problem is solved in the ETSG by defining memory locations D000 through DFFF as a 4K-byte window to the rest of the system. The initialization processor first writes an extended address word to a latch which is decoded by each memory module in the system. Being able to address all memory provides an inherent diagnostic capability which can be realized with software to eventually develop a completely self-diagnostic machine.

When the ETSG begins dynamic processing, an independent dedicated CPU takes control of the display. This "display" CPU, which is also a 6800-based design, has three functions: 1) process incoming source coordinates with missile roll angle to de-roll coordinates, 2) display all sources at the de-rolled coordinates with a different color for each source, and 3) perform dynamic error checking on source ranges and dynamic roll rate.

Present configuration of the ETSG provides for two three-dimensional sources and four, two-dimensional sources. Future phases of implementation will provide for nineteen three-dimensional sources. Present ETSG configuration provides rectilinear scanning of the field of view. Future phases of implementation will provide for reticle scanning which will require an additional section of parallel processing due to the increased number of bytes that must be convolved from the image-plane map to generate detector output.

References

(1) G. E. Riley and M. J. Sinclair, Electronic Target Signal Generator (ETSG) Design and Analysis, Engineering Experiment Station, Georgia Institute of Technology, Final Report No. A-2186, May 1980, 54 pp.

(2) C. E. Barnett and T. N. Long, Integration of a Hybrid Simulation for a Small Air Defense Missile, Engineering Experiment Station, Georgia Institute of Technology, Final Report No. A-2333, April 1980, 23 pp.

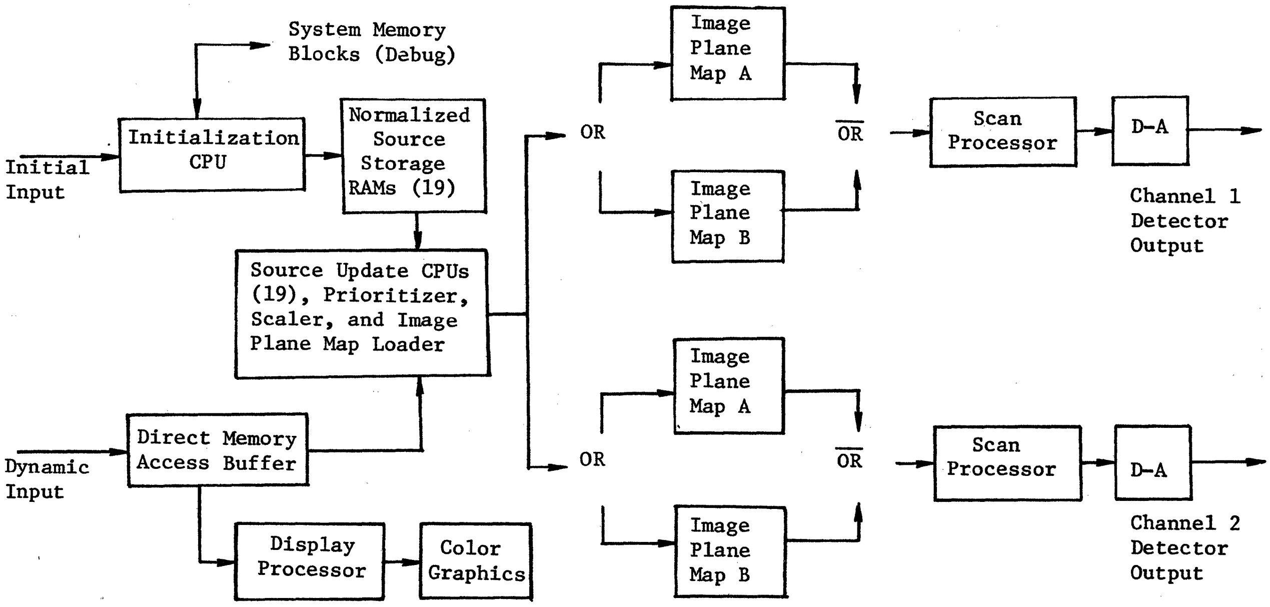


Figure 1. ETSG Block Diagram

DESIGN OF A MIXED VOICE/DATA COMPUTER NETWORK FOR PACKET-SWITCHING COMMUNICATION

Jien-Deng Kao, Jin-Tuu Wang, Te-Son Kuo, Ger-Chih Chow
Department of Electrical Engineering, National Taiwan University, Taipei, ROC

SUMMARY

In a four-wire telephone system, the channel utilization is activated by voice only 38% in one direction during conversation interval. It is therefore possible to use the remaining capacity of silent period for inserting and extracting data in packets by using a microcomputer-based controller called Mixed Voice and Data Processor (MVDP), which is one of the nodes of MVD network (MVDNET), that plays as a value added service of existing voice-grade telephone network.

The voice signal remains its analog form as well as its real-time property and is assigned with higher priority than data traffic, which can besstored-and-forwarded by MVDP in MVDNET.

The protocol, which is based on the ISO defined open system interconnection and CCITT recommendation X.25, is adopted to cope with the recently growing up public packet-switching data communication.

This paper describes the design consideration of packet switching MVDNET, laying emphasis on details of special hardware components, adaptive routing algorithm and hierachical flow control structure.

I. INTRODUCTION

Wang and Liu have proposed a queueing model of a Mixed Voice and Data (MVD) transmission system [1]. The concept arose on the basis of low efficiency of the sampled measurment of the activity records in satellite circuits for telephone conversation, in which a channel is activated in one direction for only 38.8% of the time when it is busy [2]. Thus the rest of the silent periods can be used for data communication.

Some analysis and simulation work have been done for the proposed system [3], [4]. An improved hardware realization of Mixed Voice and Data Processor (MVDP) has been implemented by G.C. Chow [5].

This paper describes the considerations in the design of a Mixed Voice and Data Network (MVDNET), which consists of MVDP nodes, as the communication processors to provide the value added service for the existing telephone network.

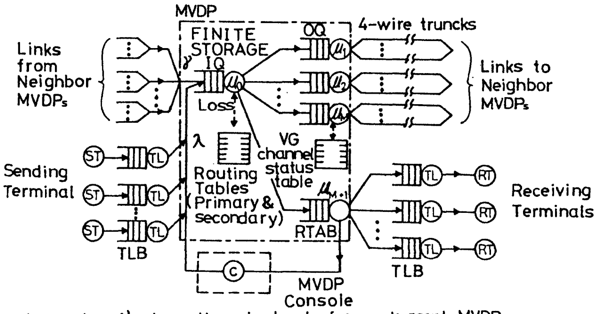
In the MVDNET, the data flow in an MVDP node is kept in a store-and-forward operation, while the voice remains in its real-time nature with higher priority to occupy the channels than data packets. In this way, voice subscribers does not know the existence of data packets and is not disturbed by data packet either. The interrupted data packets then wait for the next silent period or find another available channel by most recently realeased (MRR) rule to reduce the chance of being interrupted again. The methods to process the preempted data packet may be preemtive-repeat or preemtive-resume.

II. MVDNET Configuration

The data traffic following through the MVDNET is generated by data terminal equipments (DTEs). The sources and sinks of information (sending and receiving terminals) are attached to the MVDPs via local transmission lines (local loops), as shown in Fig.1.

In each MVDP, the process which decides to which following MVDPs or DTEs should be sent is called the switch module. The switch module makes the routing decision based on information it has on the status of the network. This routing information may be divided into a local and a global part. The local part is readily available at the current MVDP, whereas the global part has to be collected from all the other nodes. Collection of status information and preparation and distribution of the global routing information is performed by partial centralized adaptive routing algorithm.

III. Hardware Description.



Legends : γ : transit arrival rate from adj acent MVDPs
λ : source arrival arrival rate from local DTEs
μ_0: service rate of dispatcher (CPU)
$\mu_1, \mu_2 \cdots \mu_M$: link output rates
μ_{M+1} : MVDP to local DTEs output rate

Fig.1 MODEL OF A STORE-AND-FORWARD MVDP

We first summarize the important features of the MVDP design:

(1) It keeps existing voice-grade (VG) telephone network in analog form and considers voice signals as real-time through traffic having higher priority than data packets;

(2) It retransmits the data packet (preemtive-repeat mode) or residual part (preemtive-resume mode) whenever an interference occurred due to the arrival of voice signals.

(3) It uses voice detectors to sense the channel activity status and to interrupt the MVDP whenever there is a change of status (voice arrival or termination).

(4) It uses data recognizers to distinguish data from voice at the receiving side, therefore, they can be switched accordingly by an analog switch that feeds the data signal to modem.

(5) It uses multiple voice-grade channel to achieve higher data throughput and provide flexible choice of data-packet routing.

Among the hardware components, the important devices different from other existing data networks are voice detector, data recognizer and modified Modem. We will describe the feature of such dèvices in the following sections.

3-1 Voice Detector

The voice detector is a very important device and is difficult to be implemented due to its sensitivity. Its noise rejection figures are directly related to the utilization of voice-grade channels.

Fig. 2, shows the block diagram of the voice detector. The Zero Crossing Rate (ZCR) detector is based on important criteria to determine voice starting point from noise condition. Usually, an unvoiced sound has high zero crossing rate and low signal level, while white noise has both low ZCR and signal level [7]. A low level detector is used to decrease the voice dector activity caused by the background noice with high frequency and very low level. The Level Detector/Comparator is a substitution of the rms detector to avoid the 10 ms delay time of rms detector, and it has higher rejection of impulse noise than rms detector.

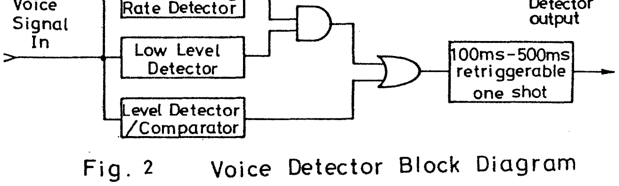

Voice Signal In
Zero Crossing Rate Detector
Low Level Detector
Level Detector /Comparator
100ms-500ms retriggerable one shot
Detector output

Fig. 2 Voice Detector Block Diagram

0190-3918/81/0000/0289$00.75 © 1981 IEEE

3-2 Data Recognizer

The purpose of this data recognizer is to distinguish data from voice signal by means of "signal pattern recognition method" according to the frequency spectrum and amplitude distribution of voice and data carrier.

There are four criteria to recognize data signal from unregular analog voice signal:

(1) Data carrier is one frequency component of voice signal which may trigger a normal voice detector.
(2) Data signal is transmitted in high level amplitude and with much more energy than most of voice signals.
(3) The pure data carrier is a signal tone signal. Its hamonic distortion should be less than 5 percent.
(4) The frequency components outside the carrier bandwidth, during the transmission of the pure carrier are 20 db to 40 db lower than inband signal. This criterion is the major role to recognize the existence of data signal. Fig.3 shows the implementation of Data Recognizer using the above mentioned four criteria.

3-3 Modified Modem

To achieve higher data rate, the modem is designed using the 4 phase PSK technique. An advantage of PSK modem is its ultra-low error rate.

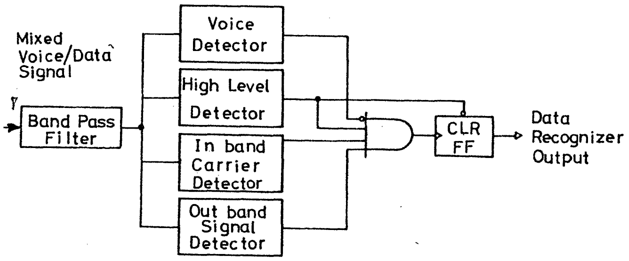


Fig.3 Data Recognizer

IV. Communication protocols for MVDNET

Most important among the functions of the communication protocols are error, flow and congestion control, and routing strategies. In the MVDNET, there is a protocol hierarchy, as shown in Table 1, its structure is based on the seven levels open system interconnection defined by ISO [8]. For the sake of public data packet transfer application in existing analog telephone networks, Level 1 is CCITT X.21 bis [9], while Levels 2 and 3 are the corresponding levels of CCITT X.25 [6]. Transport level is supposed to ensure reliable source DTE-to-sink DTE sequenced delivery; one candidate is TSS25 [10]. The Session Layer deals with the establishment of logical relationships between application entities, the maintenance of this relationship, and the handling of dialogue. The Presentation Level deals with data structuring. It contains a set of protocols which depend on the type of data to be exchanged. The Application layer deals with the user interface functions which include all service facilities such as security, multi-address working, service type, service offering, etc.

Table 1. MVDNET Protocol Hierarchy

Protocol level	Function
7. Application Level	The user/service interface.
6. Presentation Level	Data formatting features.
5. Session Layer	Logical relationships between application entities, etc.
4. Transport Service	TSS 25, DTE-TO-DTE control.
3. Network Access	Permanent or switched virtual circuit
2. Link Level	Asynchronous Balance Mode of HDLC
1. Physical Level	CCITT X.21 bis.

V. Buffer Management Policy

Buffer management policy in subnet of MVDNET is based on two rules:

(1) Output queue length limit.
(2) "Transit" traffic is assigned a higher priority than "New" traffic.

Two-priority scheme is proposed whereby transit traffic (consists of packets that have traveled over longer one ormore hops) has a higher priority than new traffic as regards to the buffer allocation. This is done by setting a limit on the number of occupied buffers, beyond which new traffic is rejected and transit traffic may be accepted. Let

B = the total number of buffers in an MVDP.

L = a threshold allocated value.

N = the number of outgoing links in a MVDP and b_{max} be the maximun length of an output queue (where $b_{max} > B/N$).

M = the number of allocated buffers, n_i of which are utilized by link i, upon arrival of a packet route to link i.

Under the constraints (a) $0 \leq n_i \leq b_{max}$; (b) $\sum_i n_i \leq B$, the following decision rule is applied:

(1) If $L \leq M < B$ and $n_i < b_{max}$, than accept thetransit packet, reject the new packet.
(2) If $M < L$ and $n_i < b_{max}$, then accept the arriving packet.

VI. Adaptive Routing Algorithm

There are two-level routing policies in the subnet of MVDNET i.e. path routing level and channel routing level under selected link. Fig. 4 shows the routing levels. The path routing level in MVDNET uses a database describing the network to generate a tree representing the minimum delaypaths from a given root MVDP to every other network MVDPs. The VG channel routing level of MVDNET selects a silent VG channel from the provided several Voice-grade (VG)channels between MVDPs.

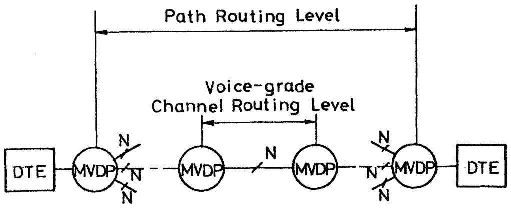


Fig.4 Routing levels in MVDNET

6-1 Path routing level

The adaptive routing procedures [11] for path routing level of MVDNET are described as following:

Routing computation—The First Two Shortest Paths (FTSP) Algorithm

(1) The basic FTSP algorithm uses a database describing the network to generate two tree representing the first two minimum delay paths from a given root node to every other network node.
(2) To implement the primary and secondary tables according to the first and secondary minimum path trees respectively.

Routes in MVDNET are assigned on a single-path-per Virtual Circuit (VC) basis. The assignment of a route to a switched virtual circuit is established temporarily or permanently.

6-2 VG channel routing level

Between hop levels (MVDP-to-MVDP), several four-wire voice grade channel are provided to improve the network throughput and lower the delay time through the MVDP hop.

In order to embed digital data packets into the silent of VG channel, we present the voice detectors at transmitting side to monitoring the VG channel status (talking or silent period)

(1) The interrupt events of voice arrival or terminating are recorded into the VG channel status table.
(2) The channel number, in which the voice is terminating, is push into the most recently released (MRR) stack.
(3) If any packet is to be sent, then it takes a VG channel number from MRR stack.
(4) After the packet has been sent, then check the VG channel is still in silent state or not? If true, then push the number of VG channel into MRR stack.

VII. Flow Control Policy

In MVDNET, the flow control is designed in hierachical multilevel structure, and these levels are actually embedded into corresponding levels of protocols.

7-1 Link level flow control

Link level flow control in MVDNET is carried out by the HDLC protocol [6,8], in asynchronous balanced mode (ABM). It operates in a local way that it monitors local queues and buffer occupancies at each node and rejects transit (from adjacent MVDPs) traffic at the node when some predefined thresholds (e.g. maximum queue limits) are exceeds.

A physical network path is set up for each user session and is released when the session is terminated. Sequeneing and error control are provided at each step along the path. In addition, it permits the application of selective flow control to each individual VC stream. Packet buffers in MVDP are dynamically allocated to VC's based on demand (complete sharing), but thresholds are set on individual VC allocations as well as on overall buffer pool utilization.

7-2 Entry-to-exist Level flow control

In MVDNET, in which a fixed route is assigned to each user session during setup time, the entry-to-exit flow control is applied individually on each virtual circuit.

When the source DTE transmitting rate exceeds the sink receiving rate, the flow control mechanism intervenes to slow down inputs from the source DTE into the entry MVDP. The window size W must be large enough to permit each virtual circuit to efficiently utilize the bandwidth available on the path.

7-3 Network access flow control

The network access scheme in MVDNET is similar to input buffer limit proposed by Lam [12]. The different point is that in the former case an new input packet from external souce DTE is discarded if the total number of the packets in the entry MVDP exceeds a given threshold, while in the Lam's scheme an input packet is discarded when the number of input packets exceeds a given threshold. Transit packets can freely claim all the buffers. It is clear that the network access control in MVDNET prevent congestion by favoring transit traffic over input traffic.

7-4 Transport level flow control

The transport flow control is based on a window mechanism as in LL, ETE levels. Namely, the receiver grants transmission credits to the sender as soon as reassembly buffers become free. Upon receiving a credit, the sender is authorized to transmit a message of an agreed-upon length. When reassembly buffer become full, no credits are returned to the sender, thus temporarily stopping message transmissions.

VIII. Conclusion

In this paper we have proposed a mixed voice and data communication network (MVDNET) as a value added service of existing VG telephone network by using the MVD processors (MVDP) as packet switching nodes.

The analog voice signal, due to its real-time property, is assigned with higher priority than data traffic. The latter can be stored-and-forwarded by the MVDP. Therefore most of conventional computer network protocols, buffer management policy, adaptive routing algorithm and flow control scheme are suited for data transfer in MVDNET. The ISO defined hierachical structure of protocols is adopted so as to cope with the growing-up public packet-switching data communication.

Routing and flow control procedures have traditionally been developed independently in packet networks, however, both are brought together into useful cooperation in MVDNET, which is a virtual call network, where a path must be selected before data transfer on a user connection begins. The routing algorithm is invoked first to determine whether primary route of sufficient residual bandwidth is available, else test secondary route next. If both paths are congested, the virtual circuit connection is blocked immediately at the entry node by the network access flow control level, thus preventing congestion rather than allowing it to occur and then attempting to recover from it.

In the future design, the modem using coherent detection may be used to obtain higher speed. Also, the CPU set may use the bipolar microcomputer with memory that quick access time. Thus, the MVDNET will become a high throughput, inexpensive and standardized network which provides improved efficiency and reduced cost public packet-switching data communication service from existing VG telephone network.

REFERENCE

[1] Jin-Tuu Wang, Ming T. Liu, "A Novel Model for a Mixed Voice/Data (MVD) Transmission System for Computer Communications", Proc. of International Computer Symposium, Vol.I, Taipei, ROC, (Aug. 1975) pp.458-468.

[2] J.T. Wang, M.T. Liu, "Analysis and Simulation of the Mixed Voice/Data Transmission System for Computer Communication" Conf. Record, 1976 NTC, Dalas, Texas. USA. Vol.III, pp.42-3-1 to 42-3-5, (Dec. 1976).

[3] J.T. Wang, J.L. Lin "Queueing Model for the Mixed Voice/Data Transmission System for Computer Communication", Proc. NCS, ROC, 1976, Taipei, Taiwan, ROC, pp.3-12 to 3-19. (Dec. 1976).

[4] J.T. Wang, L.C. Tsai, T.S. Kuo, "The Optimum Channel Selection Strategy of the Mixed Voice/Data Transmission System" Tech. Report. Ee Inst, National Taiwan University, Taipei, Taiwan, ROC, (May 1977).

[5] G.C. Chow, "Advanced Realization of Mixed Voice and Data Processor" Master Thesis, EE Inst, National Taiwan University, Taipei, Taiwan, ROC, (May, 1980).

[6] CCITT, "Provisional Recommendation X.3, X.25, X.28, and X.29 on Packed-Switched Data Transmission Services" ITU Geneva, Switzerland, (1978).

[7] B.S. Atal, L.R. Rabiner, "A Pattern Recognition Approach to Voiced-Unvoiced-Silence Recognition", IEEE. Trans on ASSP, (June, 1976).

[8] ISO/TC 97/SC16/N230, "Reference Model on Open Systems Interconnection" International Standards Organization, Pairs, (July 1979).

[9] CCITT, "Recommendation X.21 bis: Use on Public Data Networks of Data Terminal Equipments (DTEs) which are Designed for Interfacing to V-series Modems, Public Data Networks", Orange Book, Vol. VIII. 2, 6th Planning Assembly Int. Telecom. Union, Geneva, (1977) pp.57-69.

[10] UK PO Study Group 3, "A Transport Service, Data Communication Protocol Unit", National Physical Laboratory, London, (May 1979).

[11] J.M. McQuillan, I. Richer, E.C. Rosen "The New Routing Algorithm for the ARPANET" IEEE Trans. on Commu. Vol. Com-28, No.5, (May, 1980), pp.711-719.

[12] S. Lam and M. Reiser, "Congestion control of store and forward networks by buffer input limits," IEEE Trans. on Commn., Vol. COM-27, No.1, (Jan. 1979). PP.127-134.

A NEW TYPE OF MIMD- ORGANIZED MULTIPROCESSOR HANDLING TWO- STAGE PARALLELISM BY MEANS OF A DYNAMICALLY. CONFIGURABLE ARCHITECTURE

R. Bührer

ETH (Swiss Federal Institute of Technology)
Zurich, Switzerland

Summary

At ETH, a new simulation package PSCSP (power- series continuous simulation program) for the simulation of continuous systems is currently in the final stage of development [1], [2]. The main attributes of its integration technique can be summarized as follows:

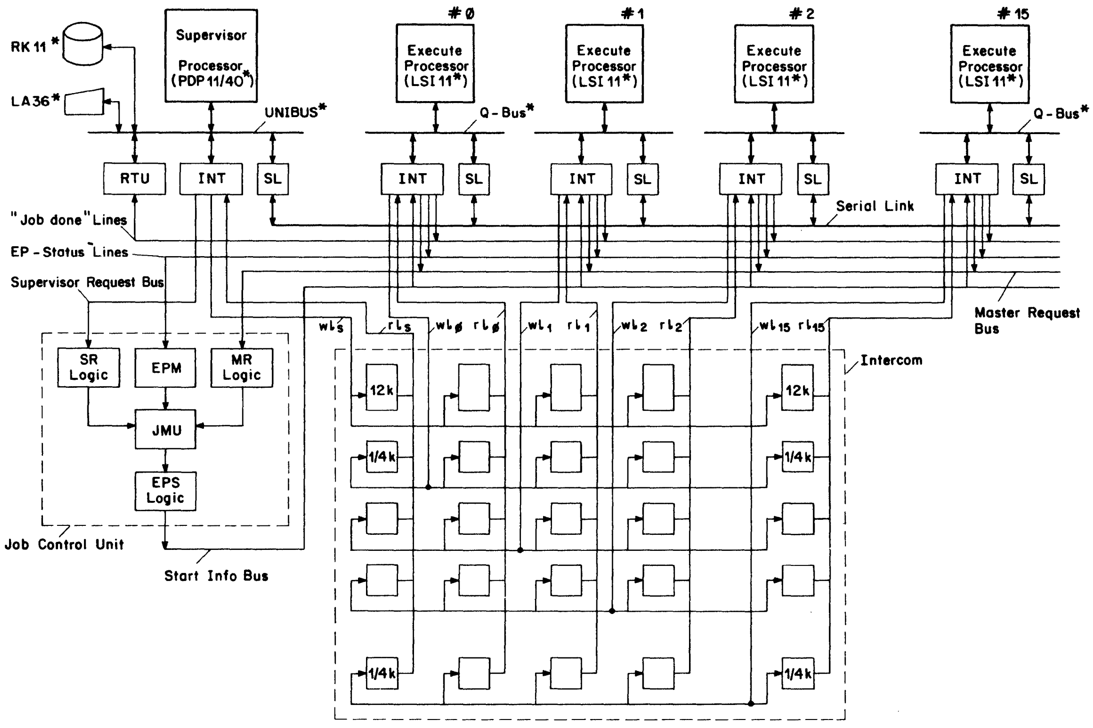
> When applying the method of power series expansions to the integration of ordinary differential equations, the right-hand sides of the equations have to be decomposed into elementary expressions. Using a set of library routines (FORMULAS) for these expressions, the higher derivatives can be evaluated analytically when calling all FORMULAS repeatedly with increasing orders of expansion. The parallelism has a two- stage nature: several independent FORMULAS can be evaluated concurrently (first level of parallelism) while most of them feature an internal parallel structure (second level of parallelism) that can be exploited simultaneously by a number of processors.

In order to acquire reliable information about the maximum gain in speed of a parallel PSCSP compared with the sequential version, a multiprocessor was built whose design and programming is consistent with the special nature of the encountered parallelism. This processor, whose main parts are outlined in figure 1, is based primarily on an MIMD (multiple- instruction stream - multiple- data stream) parallel computer concept [3] improved by a new dynamically configurable architecture principle [4]. In addition to a supervisor processor, whose activities are input/ output, compilation tasks and supervision of FORMULA- executions, a set of 16 execute processors (EP), loaded with identical software, are used for execution of the FORMULAS. Due to the individual activities of these EPs within the same or different FORMULAS, the execution of every FORMULA necessitates the creation of cooperating EP- groups whose members are distributed arbitrarily. This is done by the job control unit which generally serves as the EP- dispatching logic. In order to handle the numerous data transfers between cooperating processors in an appropriate way, a new interconnection memory intercom was developed, whose great advantage is the fact that a result provided by any of the EPs is made immediately available to all other processors; i.e. between any two processors data can be exchanged simultaneously without any delay. This intercom consists of a quadratic organized memory- matrix $C = (c_{ij})$ $(i,j = 1,\ldots 17)$ whereby processor k $(1 \leq k \leq 17)$ duplicates its data into all c_{kj} $(j = 1,\ldots,17)$ elements of its associated row. Reading is possible in all c_{ik} $(i = 1,\ldots,17)$ elements of its associated column. It can be shown that possible data protection problems can be eliminated if - in a restricted sense - data are only duplicated into the matrix elements of those processors which are working on the same FORMULA.

Whenever the job control unit starts a group of EPs, a configuration information is transmitted to the intercom interfaces of each EP of that group, allowing a transformation of job- specific address modes as "relative left (right)- or absolute within a group" into physical counterparts by means of hardware. Additional logics such as the result transfer unit (for direct- memory- access result transfers from the EP region to the supervisor processor region of the intercom) and a serial link for deadstart and maintenance tasks complete the system hardware.

Extensive performance measurements of the system (which is fully operational since the spring of this year) proved that the multiprocessor allows an exact determination of the gain in speed achieved by the parallel PSCSP. This is due to the fact that the system overheads can be separated precisely into technological overheads (resulting from hardware compromises such as the use of standard LSI- 11 EPs instead of fast, flexible microprocessors, or from the use of conventional RAM- memories instead of dual- access memory chips in the intercom, etc.) and principal overheads based on the specific system architecture. In the worst case these overheads turn out to be 27% and 10% respectively in relation to an optimally programmed sequential PSCSP- version. As shown in [4], present- day hardware technology allows the realization of a system whose technological overhead is almost zero.

0190-3918/81/0000/0292$00.75 © 1981 IEEE



*)	product of Digital Equipment Corporation	MR:	Master Request
RTU:	Result Transfer Unit	JMU:	Job Mangement Unit
INT:	Intercom Interface	EPS:	Execute Processor Start
SL:	Serial Link Interface	wl:	write lines
SR:	Supervisor Request	rl:	read lines
EPM:	Execute Processor Monitoring	k:	1024 words/ 16 bits

Figure 1 - Hardware of the Multiprocessor

References

[1] H.J. Halin, R. Bührer, W. Hälg, H. Benz, B. Bron, H.J. Brundiers, A. Isacson and M. Tadian, "The ETH multiprocessor project: parallel simulation of continuous systems", Simulation (October, 1980) pp. 109 - 123

[2] H.J. Halin, R. Bührer, W. Hälg, "Software Development for the ETH- Multiprocessor Project: Parallel Integration of Ordinary and Partial Differential Equations", Proceedings of the Second IMACS (AICA) International Symposium on Computer Methods for Partial Differential Equations, (Lehigh University, Bethlehem, Pennsylvania, 1977)

[3] M.J. Flynn, "Some Computer Organizations and Their Effectiveness", IEEE Transactions on Computers, vol. C- 21, No. 9, (September 1972) pp. 948 - 960

[4] R. Bührer, "Hardware eines dynamisch konfigurierbaren Multiprozessors", PhD thesis, Swiss Federal Institute of Technology, Zurich, (to be published)

PARALLEL PROCESSING IN COMPUTER COMMUNICATIONS

A. Faro and G. Messina
Istituto Elettrotecnico
Università di Catania
CATANIA, ITALY

Summary

The aim of this paper is to analyze, by simulation, the advantages of parallel processing in computer communications. In such a study the parameters of both the protocol and the hardware configuration characterizing the communication system are taken into consideration. A simulation model is presented which is to perform both the analysis and the synthesis of the processes in such a communication system. Finally, using this model, some results on the parallelism obtainable inside a transport protocol are shown.

Introduction

With the grand development of distributed processing systems, considerable efforts have been made in the experimental and theoretical field of computer networks in order to obtain reliable and correct interaction between users and applications through a Communication Device (CD) (fig.1). As is well known, such an interaction arises by means of a message exchange following a suitable set of rules (protocol). Because of modularity and flexibility reasons CD offers the users its service by a set of protocols which are hierarchically organized in layers; each protocol being performed by a pair of Communication Processes (CPs) generally running on remote computers. So besides the protocol at the application layer, other protocols at the lower levels exist in modern networks |1| (fig.2).

Since each CP can perform its functions at the same time as the functions performed by all the other CPs, we are faced by the problem of determining under what conditions parallel processing can be useful in computer communications. Two main solutions are possible to implement such CPs : implementation inside main frames or minicomputers and implementation in a multimicrocomputer environment. Usually in the first solution different CPs are implemented in the same machine thus determining a sequential processing of the functions performed in the above multilayer structure. On the contrary the second solution makes the implementation of each CP on a single machine possible and at reasonable costs, thus obtaining a parallel processing which uses a multimicrocomputer structure.

The aim of this paper is to analyze, by simulation, the advantages of parallel processing in computer communications. In such a study the parameters of both the protocol and the hardware configuration characterizing the communication system are taken into consideration. In particular sect.2 discusses the possibility of parallel processing in computer networks. In sect.2 a simulation model is presented which is to perform both the analysis and the synthesis of the CPs in the above multilayer structure. Finally sect.4 first analyzes how the hardware configurations can influence the performance of a transport protocol, then it shows some results on the parallelism obtainable inside such a layer.

Parallel processing

As is well known the communication structure of the computer networks is organized in a multilayer hierarchical architecture. Each layer consists of CPs and offers suitable services concerning the data transfer from one process to another of the upper layer. The service offered by each layer is obtained by means of messages exchanged between the CPs inside the layer depending on suitable communication protocols. In such networks, the decomposition is not only applied to partition the functions of the network into subfunctions to be performed by each layer |1|, but it is also used to study the internal structure of the CPs inside the layers |2|. For example a CP can be partitioned into three main subprocesses which manage respectively the information exchange with the upper layer (upper interface), with the lower layer (lower interface) and with the remote partner (protocol unit). Of course the degree of decomposition depends on the purpose of the study. For example in defining and validating protocols and services, suitable decomposition of the CPs may be useful in simpifying the problem. Similarly in implementing the CPs, one can apply decomposition techniques within certain limits in order to obtain a physical realization of such CPs with the desired modularity, flexibility and speed characteristics. This paper deals with parallel processing and then we are interested in decomposition techniques and hardware/software architectures which allows us to obtain a suitable degree of parallelism. So we take into special consideration the implementation of such CPs and, possibly, of their parallel subprocesses in multimicrocomputer environment.

0190-3918/81/0000/0294$00.75 © 1981 IEEE

The multimicrocomputer structures for networking have been widely analyzed by the authors in preceding papers |3|,|4|. Generally two solutions are possible : the first consists of a structure in which a microboard performs a CP, on the other hand in the second different CPs are implemented in the same microboard. In addition the microboards can interact by means of a shared memory area which can be obtained either from various memory chips or a single common memory chip. Of course a suitable analytical or simulation study is necessary to evaluate in detail the influence of the kind of implementation on the communication structure performance (as for example the throughput, the delivery delay or the reliability). Due to the complexity of such a problem a simulation based approach is proposed in the next section.

A simulation model

A computer network consists of CPs which are variously interconnected through communication channels. A CP can be schematized as a device with two inputs, two outputs and a set of internal states |2|. The inputs and the outputs of a CP at the layer N regard respectively the services offered by the layer N and by the layer N-1. Messages concerning respectively the protocol at the layer N+1 and at the layer N are embodied in such inputs and outputs. The set of rules a CP follows in producing all its outputs is called global procedure and depends on all its inputs and/or its internal states. The global procedure can be subdivided in three main subprocedures : P concerning the message exchange with a remote CP at the same layer, Q and U concerning respectively the command exchange with the adjacent CPs at the upper and lower layers. In addition such subprocedures can be partitioned into two other subprocedures :

- P_T,Q_T and U_T which are involved in the transmission of the messages inside P,Q and U;
- P_R,Q_R and U_R which are involved in the reception of the message inside P,Q and U.

Two fundamental problems arise in managing such an internal structure : the first concerns the environment in which the subprocedures of a CP are implemented, the second regards the environment in which the internal and interface buffers of a CP is implemented. Thus we introduce a CP supervisor which follows suitable rules in order to solve such problems |5|.

It is easy to understand that the simulation of the communication structure is very complicated because of the numerous interacting CPs. Therefore we propose a down-top approach which allows the network simulation, layer by layer, by using a recurrent structure. Such a structure consists of two CPs interacting through a lower level CD. In this way it is possible to characterize the whole communication structure of the CD connecting the end-user processes. Such an approach allows us not only the analysis but also the synthesis by solving the following general problem step by step :

- Given the communication device CD at the layer N-1 (that is for example the capacity, the delivery delay, the error probabilities etc.) and the input traffic of CD at the layer N;
- Optimize CD at the layer N (that is minimize the delivery delay in interactive systems or maximize the capacity in batch systems or both in general purpose systems etc.);
- Over all the possible protocols, interfaces and hardware configurations.

An high interdependence exists between two adjacent layers : the first concerns the non linearity of the parameters relative to a CD which depend on the working load coming from the upper layer; the second arises when the CPs of different layers are implemented in the same environment. For this reason it is necessary to characterize each layer by determining the performance depending on the working load. In our model starting from the knowledge of the delivery delay, the capacity and the error probabilities at the first layer, we obtain the same parameters for the CD at the second layer and the processing rate and memory acces time depending on the working load coming from the layer 3 and so on. The processing rate is used to simulate the critical regions relative to the CPU while the memory access time is used to simulate the critical regions relative to the data structure. In fact a CP runs according with the constraints with the lower level CD relative to such critical regions.

Results

A simulation program written in SIMULA has been derived from the above model. The first results regard an and-to-end layer provided with window flow control and recovery implemented at the upper level of a link layer. Three hardware configurations are examined in this paper as shown in fig.3. For each of the configurations, the throughput, the delivery delay and the memory occupation depending on the window size are shown in fig.4. Similar diagrams can be drawn for all the other protocol parameters (as for example the time outs and the message repetition number in the recovery phase). This allows us to optimize such parameters and to choose the best recovery and flow control strategies and hardware configuration. Further improvments are obtainable with a parallel transmission and reception of the protocol as shown in fig.5. However a cost analysis must be performed in this case to evaluate the advantage of such a parallelism. The use of this program is now planned to study the structure of a local net.

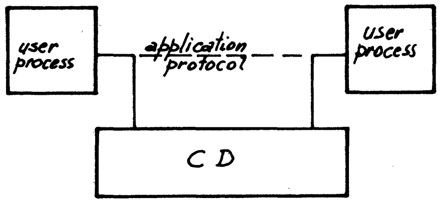
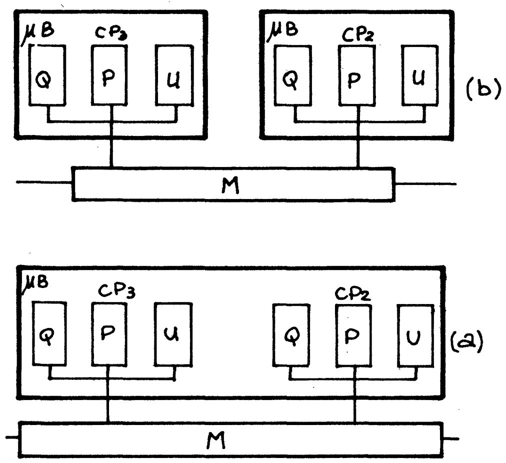


Fig.1 - Interaction between user processes



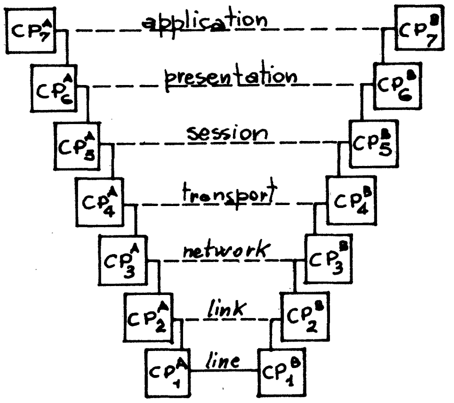


Fig.2 - Protocol layers in computer networks

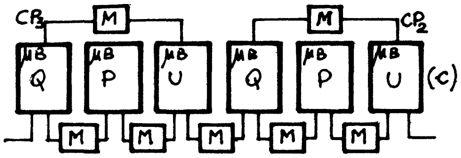


Fig.3 - Various hardware configurations to implement transport and link processes (i.e. CP3 and CP2).

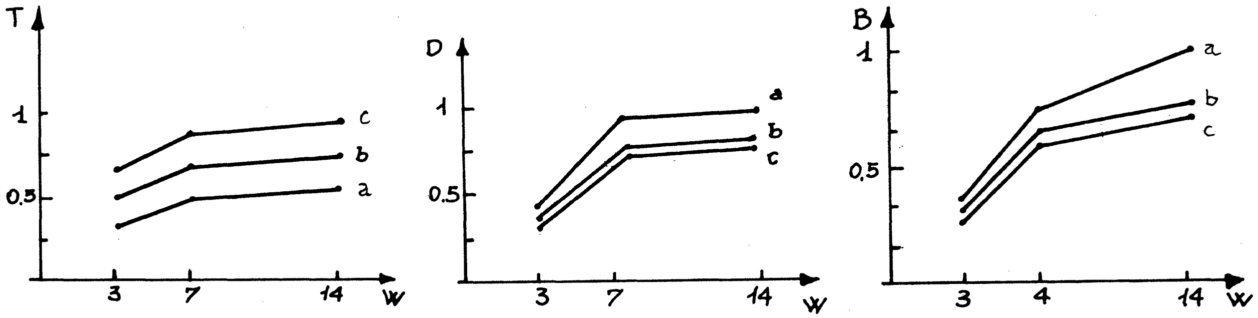


Fig.4 - Throughput (T), Delivery delay (D) and Buffer occupation versus window size relative to the above hardware configurations. The diagrams are opportunely normalized.

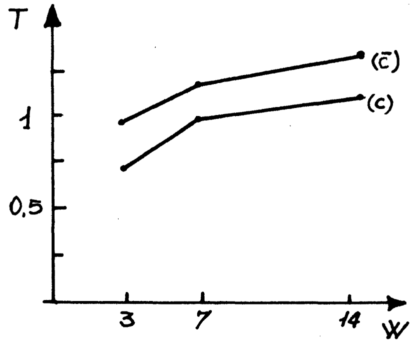


Fig.5 - Throughput (T) versus window size relative to an hardware configuration with parallel transmission and reception ($\bar{c}$).

References

|1| ISO : Open System Architecture—TC97SC16,80

|2| G.Le Moli : The second theory of colloquies-Submitted to Computer Networks, 1980.

|3| A.Faro,G.Messina : A microcomputer based gateway between computer networks - MIMI 81 Proc. San Diego 1981.

|4| G. Messina : Microcomputer based technologies for internetworking - IEEE ELECTRO Proc.New York 1981.

|5| A. Faro : Simulation based design of protocols and interfaces in computer communications - Summer Simulation Conf. Proc. Washington 1981.

PROCESS SYNCHRONIZATION IN THE PARALLEL SIMULA MACHINE

M.P.Papazoglou,P.I.Georgiadis and D.G.Maritsas
Digital Systems Laboratory,Department of Computers
N.R.C."Democritos"
Aghia Paraskevi Attikis
Athens - Greece

Abstract

A multiprocessor machine approach for parallely executing SIMULA programs has been recently proposed by the authors. As the constituent parts of the SIMULA programs at execution level are "processes", parallelism in the SIMULA machine is considered at process level. The aim of this paper is to present the problems of processes synchronization which arise during the parallel evolution of processes within the SIMULA multiprocessor machine. The issue of mutual exclusion in critical sections in such a system is discussed, and, according to the type of interprocess communication, corresponding modes of process synchronization are described. It is argued that semaphores and semaphore-queues rather than monitors are the most effective tools for coping with the synchronization phenomena in the Parallel SIMULA Machine.

Introduction

In a recent paper [1] we have presented the basic principles concerning a parallel SIMULA machine architecture. The SIMULA language was analyzed, constructs were proposed for detecting parallelism in SIMULA programs and rules were established to permit parallel execution of such programs in a multiprocessor environment.

SIMULA is a specially designed language which is particularly offered for describing and efficiently simulating large scale systems. These systems reveal significant potential parallelism, which although reflected in the SIMULA programs, remains unexploited by the implementation of the language within a uniprocessor environment. In contrast, a parallel SIMULA machine will provide the capability of achieving faster processing rates since any SIMULA program can proceed in parallel within the host multiprocessor system. Parallelism in the SIMULA machine is considered at process level.

In this paper we discuss the process synchronization problems which arise during parallel evolution of processes within the SIMULA multiprocessor machine. For reasons of presentation we outline the main features of the parallel SIMULA scheme which has been presented in [1].

The constituent parts of any SIMULA program are called "classes" at program definition level, and "processes" at execution level. SIMULA processes might be disjoint, but generally they are interactive. Process interaction is either affected by means of various "communication commands" ((re) activate, wait, hold, passivate, and cancel) issued by the individual processes, or by sharing relationships that extend over global data called "system variables". System variables include procedures, arrays, simple variables, and references.

The parallel SIMULA scheme is based upon a particular structure called the SIMULA Process Interaction Structure (SPIS). The SPIS caters to the parallel evolution of SIMULA processes by providing suitable information obtained both at compile and run-time levels. The SPIS employs a recognition mechanism, called the "SIMULA Parallel Process Recognizer" (SPPR).This recognition mechanism scans the SIMULA source program text and produces two kinds of table structures, the "System-Variables" -Table (SV-Table) and the "Class Templates" (CT's). The SV-Table contains all system variables which are accessible in the various SIMULA classes, and hence are manipulated by their dynamic instances known as processes. A CT provides a detailed record of actions of each particular class. Within each CT there appear such information as the system variables accessed by the corresponding class, the communication commands issued on behalf of this class, and the classes possibly affected by these commands. A CT also denotes identification of classes on a producer/consumer basis, (P/C-classes). Since each SIMULA process originates from a particular class, the information contained within a Class-Template, potentially reflect the process' interaction pattern and the P/C-process classification accordingly. It is important to establish the interprocess communication pattern before deciding parallelism at execution time. To support and maintain run-time information the "System Sequencing Set" (SQS) has been appropriately extended (E-SQS) so as to accomodate additional records on top of these used by the sequential SIMULA environment.

In order to ensure the correct evolution of processes within the multiprocessor environment an "Executive Algorithm" has been developed.This executive algorithm imposes a proper communication link between SPIS and the "Extended Run-time" system (E-RTS) needed to support SIMULA programs during their parallel execution. The executive algorithm is implemented by means of a "controller processor" whose main functions are to dispatch processes to processors, to deal with synchronization phenomena and prevent the occurence of deadblocks.

The structure of the SPIS and the dispatching rules that should apply for an efficient allocation of SIMULA processes to processors have been already presented in [1]. In the following we consider process synchronization within the parallel SIMULA machine.

0190-3918/81/0000/0297$00.75 © 1981 IEEE

SIMULA Process Synchronization

An attempt towards intervening within a SIMULA program structure so as to enable it to be executed in a parallel fashion should be focused:

(i) on successfully manipulating the pattern of possible transfers of control through the constituent SIMULA processes, and (ii) on implementing proper sharing relationships on system variables accessed by these SIMULA processes. This approach should be implemented in a well defined manner so as to allow deterministic behaviour on behalf of the program and preserve integrity of system variables.

When SIMULA processes are executed in parallel the outcome of their actions depends on their relative speed of execution. The speed of SIMULA-processes that run asynchronously is affected by their frequency of interaction. To achieve successful cooperation there are specific interaction points at which SIMULA processes must synchronize their actions. Therefore, a SIMULA process should be prevented from proceeding beyond certain interaction points that require some activity by other processes.

In the parallel SIMULA scheme there exist two types of syncronization problems: critical sections and deadlock These problems affect the evolution of processes and are due to the accessing of system variables and to the issuing of SIMULA communications commands. A specific procedure of the "Executive Algorithm" called "SIMULA synchronizer" receives information from the SPIS concerning process interaction points, and applies the required solution.

In a multiprocessor environment, SIMULA processes may refer to, and modify, system variables within blocks of statements known as "critical sections". In the parallel SIMULA structure, critical sections consist of system variables instead of physical resources, which is the usual approach in the operating systems concept. Applying mutual exclusion in critical sections achieves synchronization of SIMULA processes, preserves integrity and consistency of system variables and quarantees program determinacy. Furthermore, deadlocks occur when nested critical sections of system variables are encountered. As stated by Hansen in [2], an absolute hierarchical pattern of communication established among processes will assure deadlock elimination. In the E-SQS of the parallel SIMULA scheme, hierarchies are established according to the unique event time associated with each process. As a consequence, this hierarchical order prevents deadlock occurence during the parallel evolution of processes.

Mutual Exclusion in SIMULA Processes

In this section we consider the necessary criteria for an efficient solution of the synchronization problems, and we investigate the fundamental synchronizing modes which are handled by the "SIMULA Synchronizing Procedure" (SSP). The well known principles for an efficient solution of synchronization problems have been adapted so as to suit the parallel SIMULA machine requirements. These synchronizing principles are as follows;

(a) Only one process is permitted to be inside a critical section at any time instance.

(b) When a process is inside a critical section other processes trying to enter this section will be delayed.

(c) The priority rule used to determine which process will enter a critical section is based upon the time hierarchy among processes. This time hierarchy results from the event times associated with each particular process as it appears within the E-SQS. The process associated with the least event time will proceed first.

Synchronizing modes

The synchronization phenomena evoke runtime action that is provided by the SIMULA synchronizing procedure of the "Executive" algorithm. This algorithm deals with two fundamental synchronization modes:
(i) the V-mode, specifically related to the synchronization imposed by the critical sections,and

(ii) the C-mode, related to the synchronization imposed by the SIMULA communication commands within the critical sections.

In the following we critically examine synchronizing cases representative of each synchronization mode. All generalizations of these cases are solved by employing identical synchronization techniques. It is assumed that all of the processes being considered have been allocated to, and run on, a processor according to the dispatching rules introduced in [1].

V-mode

Case (a): Let $\{P_1[x_1,t_{p_1}]//P_2[x_1,t_{p_2}]\}$denote any two SIMULA processes running in parallel,where x_1 is their common system variable (i.e. a common critical section), and t_{p_1}, t_{p_2} their associated event times respectively. Assuming that $t_{p_1} \leq t_{p_2}$,P_1 establishes a higher priority of execution over P_2. Regardless of the relative execution speeds concerning P_1 and P_2, P_2 should suspend its execution action prior to entering its section, while P_1 is allowed to proceed inside its associated critical section. When P_1 leaves its critical section then "SIMULA Synchronizing" procedure (SSP) of the executive will immediately signal the processor associated with P_2 to resume its execution.

Case (b): Let two SIMULA processes $\{P_1[x_1, x_2;t_{p_1}]//P_2[x_1,x_2,t_{p_2}]\}$, run in parallel. Fig.1(a) shows a possible arrangement of the processes'"system variables" x_1, x_2 and their associated critical sections.

P_2 is delayed (blocked) at the critical statement which contains x_2. Process P_1 continues executing its critical section statements and leaves its critical section at x_2. At this point P_2 resumes execution. It can be shown that the general case of n-processes accessing K-system variables can be approached in a similar fashion.

The synchronization between processes becomes more complicated when processes start issuing SIMULA communication commands from inside their critical sections. In such situations a more elaborate solution is required.

C-mode

Let $\{P_1[x_1, \uparrow, t_{p_1}]//P_2[x_1, t_{p_2}]\}$, where ↑ indicates any possible SIMULA communication command. It is necessary to investigate the nature and the effects caused by each individual command.

It is evident that P_2 is delayed until P_1 deals with the issued communication command. According to the specific command, action is taken as follows:

(a) Hold (T)-command: It advances the event time of P_1 by T, i.e. $t_{p_1}=t_{p_1}+T$. It follows that:

(i) If $t_{p_1} \leq t_{p_2}$, then P_1 continues execution while P_2 still remains blocked expecting P_1 to leave its critical section.

(ii) If $t_{p_1} > t_{p_2}$, then P_2 resumes execution while P_1 is delayed until P_2 permits resumption of P_1 at the proper time.

(b) Passivate, Wait (S), Cancel (P_1)-commands:

These commands cause removal of P_1 from inside the ESQS with a subsequent loss of its allocated processor [1]. Therefore, P_2 resumes execution and P_1 will not be effectively involved in the computation unless it is explicitly called by some other active process at a later simulation time instance.

(c) (Re) activate (P_3)-commands: Fig.1(b) shows the situation where a process P_3 is (re) called back into the ESQS. In [1] it is stated that if a process P_1 issues a "(re) activate" command, process P_3 is always a Consumer (C) -process in relation to P_1 which is a producer (P)- process. In such a case process P_3 is replacing process P_1 in its processor, because processes belonging to the C-set of a P-process all run under the same processor. The following subcases can emerge:

(i) Let $t_{p_1} < t_{p_2} \leq t_{p_3}$ or $t_{p_1} \leq t_{p_3} < t_{p_2}$: in this case these relations result from any non-immediate activation command [3]. As a result P_1 contunues being active, P_2 is temporarily delayed (blocked) while P_3 does not claim P_1's processor as it exists in a suspended state inside the ESQS. P_3 will obtain P_1's processor only when P_1 is removed from inside the E-SQS. At that instance synchronization problems might accrue between P_3 and P_2. They will naturally fall into one of the described cases.

(ii) Let $t_{p_3}=t_{p_1} < t_{p_2}$: this relation is the outcome of the execution of any immediate activation command. As a result P_1 still remains inside the ESQS but its processor is allocated to P_3 which commences execution. P_2 still remains blocked until a later instance at which P_1 leaves its critical section, thus permitting unblocking of P_2. This is expected to happen after a possible loss of control on behalf of P_3 (e.g. P_3 will be either terminated or passivated, or suspended through the appropriate command).

The C-mode cases which were examined above, can also be generalized so as to include processes accessing more than one system variable and issuing more than one SIMULA communication command from statements inside their critical sections. All these generalized cases are approached in an analogous way.

Synchronization problems are normally solved by implementing various indivisible operations on such structures as semaphores [4,5] and monitors [6,7].

There exist several reasons that discourage the implementation of monitors in the synchronizing procedure of the "Executive" in the parallel SIMULA machine. Each monitor in the operating system theory is implemented to manipulate a physical resource according to some scheduling rules. In conventional systems the limited number of physical resources usually makes possible the choice of monitors. On the contrary the high and unpredictable rate of appearance of system variables within SIMULA programs is not encouraging an efficient implementation based on the use of monitors. Therefore semaphores and semaphore-queues are considered to be an effective tool to cater to the solution of synchronization phenomena within the parallel SIMULA Machine.

References

[1] P.I.Georgiadis,M.P.Papazoglou,D.G.Maritsas, "Towards a Parallel SIMULA Machine",Procs.of the 8th Annual Symp.Comp.Arch.,(May,1981),263-278pp.

[2] P.B.Hansen, Operating System Principles, Prentice-Hall, (1973), 366 pp.

[3] W.R.Franta, The Process View of Simulation, North Holland, (1979), 244 pp.

[4] E.W.Dijkstra, Cooperating Sequential Processes, Dept.of Maths, Technological University of Eidhoven, The Netherlands,EDW123,(1965),84 pp.

[5] A.N.Habermann, "Synchronization of Communicating Processes", CACM, (April,1972),pp.171-176.

[6] C.A.R.Hoare, "Monitors: an Operating System structuring concept", CACM, (October, 1974), pp.549-557.

[7] P.B.Hansen, The Architecture of Concurrent Programs, Prentice-Hall, (1977), 317 pp.

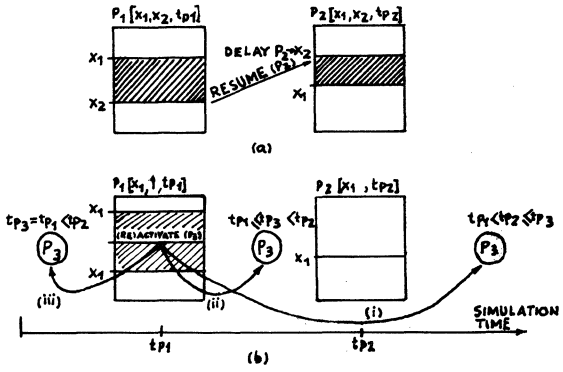


FIG. 1(a) V-mode synchronization,(case b), 1(b) C-mode synchronization,(case c).

Architecture of the First Vector Computer of China

Gao Qing-Shi   Zhang Xiang
Institute of Computing Technology,Academia Sinica
Peking, China

The first vector computer of China will soon be examined with computational test programs. It is a large scale, high-speed, pipeline machine. Most of devices were designed and produced in China. The main units of the system are all made of domestic products. Architecture of the system was proposed in 1973[1,2]. The designs of architecture and system function were completed in 1974, and were examined by a national meeting in 1975[3]. In this system vector registers are adopted, and the vectors are treated in vertical--horizontal processing fashion (i.e. segment by segment fashion). Fig. 1 shows the vector computer system organization. The system consists of three parts: the vector computer, the peripheral computer, and the peripheral devices.

Various units and their functions of the vector computer are as follows:

1. ALU. It is a pipelined executing unit. It performs various scalar and vector operations. For most of the operations, it can accept a set of new operands each clock period, meanwhile issue one computation result.

2. $\vec{R}$--Vector Registers. There are twelve of them:$\vec{R}_0 \sim \vec{R}_{11}$. Every $\vec{R}_i$ has 16 elements of 64 bits each. They supply the ALU with vector operands and temporarily store the intermediate computation results at high speed. They are the basic tools that vector operation can be performed in segment by segment fashion. $\vec{R}_i \theta \vec{R}_j \Rightarrow \vec{R}_k$ is a vector instruction, θ represents one of the arithmetic-logic operations, $\vec{R}_i$ and $\vec{R}_j$ supply the ALU with two operand-vectors, the result-vector of operation will be stored into $\vec{R}_k$.

Any $\vec{R}_i$ can be used as a scalar register when it is not being used as a vector register. In this case only a particular element of $\vec{R}_i$ is used as the scalar register R_i.

3. S--High-speed Scalar Memory. It has 32 words of 64 bits each: $S_0 \sim S_{31}$. In scalar program, S serves as the backup of scalar registers. In vector program, S holds the scalar values or constants which participate in vector operations with vectors.

4. Lookahead-Fetch-Vector-Buffers. There are four of them. Each has 16 64-bit elements. They are used to temporarily hold vectors which are fetched in advance from main memory.

Post-Store-Vector-Buffers. There are two of them. Each has 16 64-bit elements. They are used to temporarily hold result-vectors which are produced by ALU and waiting for being stored into main memory.

Just as $\vec{R}$, lookahead and post vector buffers may also be used as scalar buffers.

Lookahead and post buffers are generally allocated automatically by hardware.

5. $\vec{\alpha}$--Operation-Control-Bit-Vectors. There are eight of them: $\vec{\alpha}_0 \sim \vec{\alpha}_7$. Every $\vec{\alpha}_i$ has 16 elements of 1 bit each. They are used to record the status or result-characteristic of 16 element-operations of vectors. They are used to control 16 element-operations of vectors.

α--Operation-Control-Bit-Scalars. There are eight of them:$\alpha_0 \sim \alpha_7$. Every α_i has only 1 bit. They are used to record the status or result--characteristic of scalar operation. They are used to control operations.

$\vec{\alpha}$ and α are very useful for raising the flexibility of vector processing and for enlarging the range of parallel computation. Because of them, the operations of global dependency "conditional branch according to the results of ALU operation", which are very harmful to pipeline processing, may be appreciably reduced.

6. MM--Main Memory. It consists of 16 magnetic core memory modules. Memory addresses are assigned mod 16. Every module has capacity of 32K 64-bit words. Access cycle time is less than 16 clock periods.

7. $\vec{\beta}$--Indirect-Address-Control-Vector-Registers. There are four of them:$\vec{\beta}_0 \sim \vec{\beta}_3$. Every $\vec{\beta}_i$ has 16 elements of 22 bits each.

8. $\vec{\gamma}$--Compress and Spread-Control-Bit-Vectors. There are two of them:$\vec{\gamma}_0$, $\vec{\gamma}_1$. Every $\vec{\gamma}_i$ has 16 elements of 1 bit each.

9. b--Index Memory. It has capacity of 32 22-bit words: $b_0 \sim b_{31}$.

Δ--Address Increments. There are twelve of them:$\Delta_0 \sim \Delta_{11}$. Every Δ_i has 22 bits. They are used to store address increment (or skip distance) between two adjacent elements of a vector in MM.

l--Length-Store-Eelments. There are twenty of them: $l_{12} \sim l_{31}$. Every l_i has 22 bits. l are used to store vector length (the number of elements of a vector), loop count number, etc. . l and Δ are the different parts of the same semiconductor memory unit of 32 words.

Basic word size: 64 bits.
Main Memory capacity: 512K 64-bit words
MM Vector access maximum rate: 1 word every clock tick.
ALU speed (clock cycle number needed for producing each computation result):
floating point add: 1 cycle (dependent operation: 4 cycles)
floating point multiply: 2 cycles (dependent operation: 5 cycles)
floating point divide: 8 cycles (dependent operation: 12 cycles)
most of the other operation: 1 cycle

Main features (which, except 1,2 and 8, are not available in CRAY-1 [4]):

1. Multi-Vector-Registers.
2. Vectors are treated in vertical-horizontal processing fashion (i.e. segment by segment fashion).
3. When vectors are being treated in segment by segment fashion, dividing vectors of arbitrary length N into segments of 16 (or less) elements each is automatically done by hardware.

There is a Vector-Segment-Length-Register l', it controls the execution of vector instruction. Every vector instruction executes the operations of l' elements ($l' \leq 16$). Besides, we have two special control-type instructions: vector loop starting instruction " $\ulcorner^{*}_{N \Rightarrow l_i}$ " and vector loop

0190-3918/81/0000/0300$00.75 © 1981 IEEE

ending instruction " $]^{*}_{l_i}$ ". Loop body, which is between " $[^{*}_{N\Rightarrow l_i}$ " and " $]^{*}_{l_i}$ ", will be executed $\lceil N/16\rceil$ times. The loop ends as soon as N elements of vector have been processed.

The function of instruction " $[^{*}_{N\Rightarrow l_i}$ " is:

(1)$N\Rightarrow l_i$; (2)$\min(l_i,16)\Rightarrow l'$; (3)$l_i-l'\Rightarrow l_i$

The function of instruction " $]^{*}_{l_i}$ " is:

if $l_i=0$, then loop ends;

if $l_i\neq 0$, then do:(1)$\min(l_i,16)\Rightarrow l'$; (2)$l_i-l'\Rightarrow l_i$; (3)jump back to the beginning of the loop body for continuing execution.

For example, program is:

$$[^{*}_{50\Rightarrow l_i};\ F(\vec{u},\vec{v},\cdots)\Rightarrow\vec{f};\ ]^{*}_{l_i}$$

It will be processed in the 4 ($\lceil 50/16\rceil=4$) passes through the loop as follows:

the first pass,compute $F(u_{0\sim15},v_{0\sim15},\cdots)\Rightarrow f_{0\sim15}$

the second pass,compute $F(u_{16\sim31},v_{16\sim31},\cdots)\Rightarrow f_{16\sim31}$

the third pass,compute $F(u_{32\sim47},v_{32\sim47},\cdots)\Rightarrow f_{32\sim47}$

the fourth pass,compute $F(u_{48\sim49},v_{48\sim49},\cdots)\Rightarrow f_{48\sim49}$

where $u_{i\sim j}$ represents i-th element to j-th element of vector $\vec{u}$.

4. In this system Δ-type vector, as the basic object of vector operation, can have an arbitrary beginning address D and an arbitrary address increment Δ . That is, we can have instruction $\vec{u}\theta\vec{R}_j\Rightarrow\vec{R}_k$ or $\vec{R}_i\theta\vec{R}_j\Rightarrow\vec{u}$, and $\vec{u}$ is a Δ--type vector, its address vector is $(D,D+\Delta,D+2\Delta,D+3\Delta,\cdots)$, both D and Δ may be arbitrary.

When the row vector and the column vector of a matrix are frequently used in the alternative way, there is no need to transpose the matrix many times, we only need to use the different Δ-values, we can directly process the two kinds of directive vectors.

Multi-dimention array has more directive vectors. They also can be directly processed.

As memory space allocation is concerned, it is best to select odd Δ for the most frequently and the second most frequently used directive vectors so as to attain the maximum access rate for MM.

5. Infirect vector $\overrightarrow{\beta u}$ (i.e. $\vec{u}[\vec{\beta}]$) is also a basic object of vector operation. Every instruction can directly use $\overrightarrow{\beta u}$ as its operand, or store computation result into $\overrightarrow{\beta u}$. That is, we can have instruction $\overrightarrow{\beta u}\theta\vec{R}_j\Rightarrow\vec{R}_k$ or $\vec{R}_i\theta\vec{R}_j\Rightarrow\overrightarrow{\beta u}$, where, $\vec{\beta}$ is an Indirect-Address-Control-Vector--Register, $\vec{u}$ is a common MM vector, if $\vec{\beta}=(i_0, i_1,i_2,\cdots)$, then $\overrightarrow{\beta u}$ represents a MM vector $(u_{i_0},u_{i_1},u_{i_2},\cdots)$.

For example: If in computational expression $F(\vec{u},\vec{v},\cdots)\Rightarrow\vec{f}$, we do not need to compute all the elements, we only need to compute a few elements which satisfy a particular condition. Let $\vec{\beta}$ be a vector consisting of the subscripts of those elements that satisfy the particular condition, then we only need to directly compute the following expression: $F(\overrightarrow{\beta u},\overrightarrow{\beta v},\cdots)\Rightarrow\overrightarrow{\beta f}$.

This feature is helpful for enlarging the range of parallel computation.

6. Every instruction has the function of compressing or spreading.

Spreading vector $\overrightarrow{\gamma u}$ is also a basic object of vector operation. Every instruction can use $\overrightarrow{\gamma u}$ as its operand, that is, we can have instruction $\overrightarrow{\gamma u}\theta\vec{R}_j\Rightarrow\vec{R}_k$.

For example: Let $\vec{\gamma}=(1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0)$, $\vec{u}=(u_0,u_1,u_2,\cdots)$. Then as an operand, $\overrightarrow{\gamma u}=(u_0,0,0,0,u_1,0,0,0,u_2,0,0,0,u_3,0,0,0)$.

Every instruction can compress the computation result-vector under the control of $\vec{\gamma}$ and store the compressed vector into MM.

For instance: Let $\vec{\gamma}$ be the same as before, and $(x_0,x_1,\cdots,x_{15})$ is the computation result of $\vec{R}_i\theta\vec{R}_j$, then instruction $\vec{R}_i\theta\vec{R}_j\Rightarrow\overrightarrow{\gamma u}$ is executed as follows: $x_0\Rightarrow u_0, x_4\Rightarrow u_1, x_8\Rightarrow u_2, x_{12}\Rightarrow u_3$

7. For the three register addresses i,j,k of vector instruction $\vec{R}_i\theta\vec{R}_j\Rightarrow\vec{R}_k$, no limit is set to them. i,j can differ from k, and i,j can also be equal to k.

8. In adition to the features of highly pipelined ALU and interleaved 16 modules MM, the operations of the three main units--Instruction Unit, MM Unit and AL Unit--are asynchronously and highly overlapped. Because of overlapping, the vector operation start-up time may be ignored. In general, there is no time delay between two successive vector-operations in ALU.

References

1 Gao Qing-Shi, Zhang Xiang, Wang Jia-Mo, "Principles of Pipline Vector Computer of Vertical-Horizontal Processing", Chinese Journal of Computers, No.1,1978.

2 Gao Qing-Shi, Zhang Xiang, et al., "The Main Shortcomings of ILLIAC-IV and Their Improvements", technical report, Nov., 1973.

3 Gao Qing-Shi, Zhang Xiang, et al., "Vector Computer of Vertical-Horizontal Processing", technical report, July, 1975

4 F.Baskett and T.Keller, "An Evaluation of the CRAY-1 Computer", High Speed Computer and Algorithm Organization, D.J.Kuck et al (eds.), Academic Press, 1977.

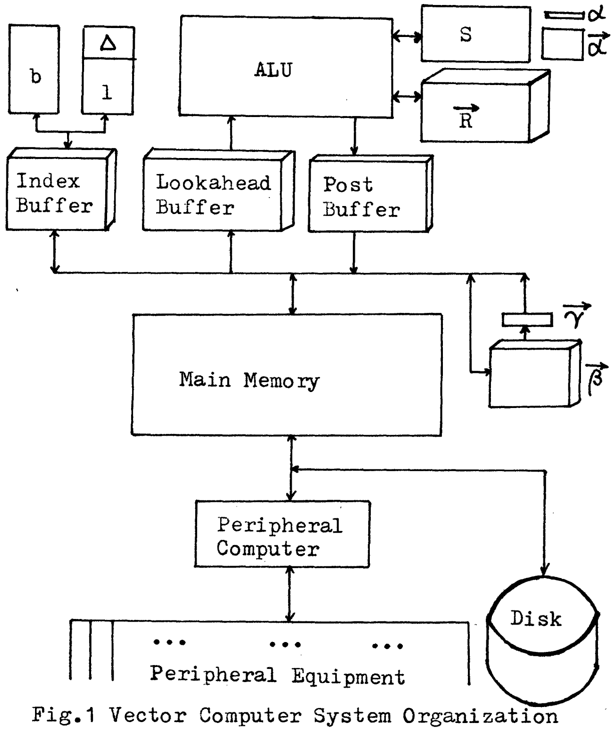


Fig.1 Vector Computer System Organization

MAX: AN ALGORITHM FOR FINDING MAXIMUM IN AN ARRAY PROCESSOR WITH A GLOBAL BUS

Shahid H. Bokhari
Department of Electrical Engineering
University of Engineering & Technology, Lahore-31
Pakistan

Summary

The problem of finding the maximum of a set of values stored one per processor on an array of processors is analyzed. The array has a time-shared global bus in addition to conventional processor-processor links. It is shown that the problem may be solved on an n X n array in $O(n^{2/3})$ time using a two-phase algorithm that uses conventional links during the first phase and the global bus during the second phase. Without a global bus the problem takes O(n) time. Two types of array interconnection patterns are considered: the eight nearest-neighbor pattern and the four-nearest neighbor pattern. The analysis is shown to apply to both cases. Extensions to q-dimensional arrays result in an $O(n^{q/q+1})$ algorithm.

Global broadcast buses provide an attractive method of increasing connectivity of array processors. Although past applications of broadcast buses have mostly been in the area of local computer networks, where they have been used to interconnect processors that have occasional bursty communications [7], improvements in technology now make it viable to use them to interconnect a number of processors that are cooperating in the parallel solution of one problem. In particular, a global bus may be combined with conventional processor-processor links in an array of processors to permit non-adjacent processors to communicate without passing messages through intermediate processors. This type of structure blurs the traditional distinction between "tightly coupled" and "loosely coupled" computers.

A practical example of such an arrangement is the Finite Element Machine (FEM), [1] being developed at NASA Langley Research Center. The prototype array is made up of 36 microcomputers arranged in a 6 X 6 grid. It is proposed to ultimately construct a 1024 processor array. Each processor is connected to its eight "nearest-neighbors" through direct processor-processor links. In addition there is a time-shared global bus to which all processors are connected.

The direct links allow pairs of adjacent processors to communicate with each other and also allow each processor to transmit a value to its eight neighbors. These operations can be carried out in parallel. In contrast, the global bus permits any two non-adjacent processors to communicate and for any one processor to broadcast its value to all other processors.

Since the bus is time-shared, only one processor may transmit a value on it at one time. Thus, although the bus improves connectivity, this is "time-shared" connectivity.

Efficient algorithms for array processors must take into account the two types of connectivity available. Challenging problems arise when developing such algorithms; consider for example the Mapping Problem [5] in which the objective is to minimize the usage of the time-shared global bus.

This paper discusses the prblem of finding the maximum of a set of values stored one per processor [2]. At the outset each processor p has a value V(p) and the problem is to find the maximum of V(p) over all processors and to transmit this maximum value to every processor. This problem can be solved for an n X n array in O(n) time using a simple algorithm that uses only the direct links. If only the global bus is used the time required is $O(n^2)$. We have developed a two-phase algorithm that first uses the direct links for a certain number of steps and then switches over to the global bus. The overall time required is $O(n^{2/3})$ which is below the time required by either of the constituent algorithms. This seemingly paradoxical result is due to the optimal selection of the switchover point. This algorithm is a refinement of an algorithm proposed in [2].

Two types of arrays are analyzed in this paper. The eight-nearest neighbor array is the FEM discussed above. The four-nearest neighbor array has an interconnection pattern similar to that of Illiac-IV.

The purely local algorithm utilizes only direct links and proceeds as follows. Each processor local broadcasts its own value to its 'a' neighbors (a=8 for the FEM, 4 for the Illiac). This causes the broadcasting processor's value to be placed in the input registers of all its neighbors [3], [4]. Each processor then reads all of its input registers, updating its own value if the received value is greater. The maximum value spreads throughout the array in time proportional to the diameter of the array, i.e. O(n).

The purely global algorithm utilizes only the time-shared global bus. At the start of this algorithm each processor attempts to acquire the

This research was supported in part by NASA Contract NAS1-14472 while the author was resident at the Institute for Computer Applications in Science & Engineering (ICASE), NASA Langley Research Center, Hampton, Virginia 23665, USA.

0190-3918/81/0000/0302$00.75 © 1981 IEEE

bus. The bus is granted to an arbitrary processor which uses it to broadcast its own value to all n^2 processors in the array. At subsequent steps only those processors that have not yet had a chance to do a global broadcast attempt to acquire the bus. In the worst case this algorithm takes $O(n^2)$ time since the processor containing the maximum value may not get the bus until the very end.

We assume that a global broadcast takes t units of time while a local broadcast takes a·t units of time (where a is the number of neighbors). This closely approximates the behavior of the FEM's hardware [1]- [4].

The two-phase algorithm that we have developed first utilizes the purely local algorithm for k steps and then switches over to the global algorithm. Processors that have never had their values updated during the local phase are called survivors. During the global phase only survivors contend for the bus. If we denote by S_k the maximum number of survivors possible in an n X n array after k steps of the purely local algorithm, then the time required by the two-phase algorithm is $T = k \cdot a \cdot t + S_k \cdot t$.

The program that executes in each processor is given below.

```
begin
   survivor:=true;
   (* begin local phase *)
   for j:=1 to k do
      begin
         local_broadcast(own_value);
         for neighbor:=1 to a do
            begin
               local_receive(neighbor,n_value);
               if own_value < n_value then
                  begin
                     own_value:=n_value;
                     survivor:=false;
                  end
               else (* do nothing *)
            end;
      end;
   (* end local phase *)

   (* begin global phase *)
   while survivor do
      begin
         attempt_global_broadcast(own_value);
         if successful then survivor:=false
         else
            begin
               global_receive(n_value);
               if own_value < n_value then
                  begin
                     own_value:=n_value;
                     survivor:=false;
                  end
               else (* do nothing *)
            end;
      end;
   (* end global phase *)
end.
```

The function local_broadcast(x) broadcasts the value x to all neighbor's input registers. local_receive(x,y) reads the input register from processor x into the variable y. Attempt_global_broadcast(x) will initiate an attempt to acquire the bus and broadcast value x. If this attempt is successful, the boolean variable "successful" will be set true. global_receive(x) causes the contents of the global input register to be moved to x.

Expressions for S_k for both types of arrays have been obtained in [6] . The optimal value of k can be derived using these expressions to obtain the minimum time to find maximum using the two-phase algorithm. These are

$T_{min} = (6n^{2/3}-4)t$ for the Illiac type array and

$T_{min} = (12(n/2)^{2/3}-8)t$ for the FEM.

For n processors connected in a ring (plus a global bus) the two phase algorithm finds maximum in $O(n^{\frac{1}{2}})$ time. For a q-dimensional array the time is $O(n^{q/q+1})$. These cases are discussed further in [6].

References

[1] H. F. Jordan. "A Special Purpose Architecture for Finite Element Analysis," Proc. 1978 Int. Conf. on Parallel Proc., (August, 1978), pp. 263-266.

[2] H. F. Jordan, M. Scalabrin, and W. Calvert, "A Comparison of Three Types of Multiprocessor Algorithms," Proc. 1979 Int. Conf. on Parallel Proc., (August, 1979), pp. 231-238.

[3] H. F. Jordan, ed., The Finite Element Machine Programmer's Reference Manual, Dept. of Elec. Eng., Univ. of Colorado, Boulder, Report No. CSDG-79-2, (August, 1979), 51 pp.

[4] D. A. Podsiadlo and H. F. Jordan, Operating System Support for the Finite Element Machine, Dept. of Elec. Eng., Univ of Colorado, Boulder, Report No. CSDG-81-2, (March,1981), 28 pp.

[5] S. H. Bokhari, "On the Mapping Problem," IEEE Trans. Computers, (March, 1981), pp. 207-214.

[6] S. H. Bokhari, MAX: An Algorithm for finding Maximum in an Array Processor with a Global Bus, Dept. of Elec. Eng., Univ. of Eng. & Tech., Lahore, Pakistan, in preparation.

[7] R. M. Metcalfe and D. P. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," C.ACM, (July, 1976), pp. 395-404.

A PRACTICAL PARALLEL ALGORITHM FOR REPORTING INTERSECTIONS OF RECTANGLES

Anita L. Chow
Communication Products Technology Center
GTE Laboratories, Incorporated
Waltham, Massachusetts 02154

Summary

Given a set of N rectangles with their sides parallel to the coordinates axes, we are asked to report all pairs of rectangles which intersect. This problem has important applications in VLSI circuitry design rule checking [1,4]. There exists O(N log N + k) time algorithms [2] for reporting all k intersecting pairs on a uniprocessor machine. However, for large input size, these results are not satisfactory. This concern has motivated our investigation of the rectangle problem using parallel computing machines to produce a faster algorithm.

Two models of computation are used in this paper. They are the shared memory model (SMM) [5] and the cube-connected-cycles model (CCC) [6] which can emulate a cube model. The validity of the SMM resides in uncovering the inherent data-dependence of a problem while the validity of the CCC, which complies with the VLSI technological constraints, is the development of practical algorithms.

We can say that two rectangles intersect if their edges intersect or if one rectangle encloses the other entirely. Thus, we can solve the problem in two intermediate steps: (1) reporting the intersections of horizontal and vertical line segments, and (2) two-dimensional range searching.

First, we study the intersection problem of a set V of n vertical line segments and a set H of m horizontal line segments. Let T(v) and B(v) denote the y-coordinates of the top and bottom endpoints of a vertical line segment v in set V. We assume, for simplicity, the following: all T(v) and B(v) are distinct integers in the interval [0, N - 1], where N = 2n is the number of distinct y-coordinates of the endpoints, and N is a power of 2. For the general case, the readers are referred to [3].

A binary search tree $\mathscr{T}$ of height log N can be produced for the set V. At each level in $\mathscr{T}$, the nodes are indexed from left to right, starting with the integer 0. The node $N_i(j)$, at height i, represents an interval $[j \cdot 2^i, (j + 1) 2^i]$ and contains a list of edges v sorted in the positive x-direction, where $B(v) \leqq j \cdot 2^i$ and $(j + 1) 2^i \leqq T(v)$, and v does not belong to any ancestor of $N_i(j)$ in $\mathscr{T}$. This data structure is suitable for implementation on the SMM but not on the CCC. Thus, we have to transform $\mathscr{T}$ into a data structure $\mathscr{T}'$ which can be implemented on the CCC. $\mathscr{T}'$ is a binary tree similar to $\mathscr{T}$, except with respect to the indexing of nodes. The left-to-right sequence of node indexing at any level of $\mathscr{T}'$ is the bit-reversal permutation of the node indices at the corresponding level of $\mathscr{T}$. Note that node $N_i'(j)$ in $\mathscr{T}'$ contains the same list of vertical line segments that are in $N_i(2 (j \bmod 2^{i-1}) + \lfloor j/2^{i-1} \rfloor)$ in $\mathscr{T}$. $\mathscr{T}'$ can be represented as a collection $\mathscr{E}$ of arrays $E_{\log N}$, $E_{\log N-1}$, ... E_0, where E_i is the concatenation of the sorted lists of vertical lines associated with node $N_i'(j)$, in the ascending order of j. Associate with a vertical line v in E_i a node number which is the node index j such that v belongs to $N_i'(j)$. Therefore, E_i is a selected list of vertical line segments sorted lexicographically by their nodes numbers and their x-coordinates.

$E_{\log N}$, ..., E_0 are determined one at a time, in the given order. Let C_i be a list of candidates for E_i, sorted by their potential node numbers and then their x-coordinates. Initially, potential node numbers of all segments are 0, and $C_{\log N}$ is the set V sorted by x-coordinates. From $C_{\log N}$, we extract segments which cover the vertical interval [0, N] to form $E_{\log N}$. In $C_{\log N-1}$, segment v has the potential node number 0, if $0 \leqq B(v) \leqq N/2$, and the potential node number 1 if $N/2 \leqq T(v) \leqq N$. Note that a segment may have both numbers. We extract from the remaining segments in $C_{\log N}$, the list C' of segments with potential node number 0 and the list C" of those with number 1. The concatenation of C' and C" forms $C_{\log N-1}$. We repeat this process for constructing successively $E_{\log N-1}$, ..., E_0.

To find intersecting pairs, we envision $\mathscr{E}$ as a binary search tree. At level i we associate with each horizontal line segment h a node number #(h) indicating that h may intersect some vertical segment in node N_i' (#(h)). It is obvious that at the $E_{\log N}$ (the root), #(h) = 0 for all h. We maintain the set of horizontal segments sorted by their node numbers and then x-coordinates of their left-most endpoints, the same manner as in E_i. We can use a one-dimensional range searching algorithm [3] to report all intersecting pairs at a level in $\mathscr{E}$. We then

This work was done at the University of Illinois at Urbana-Champaign and was supported in part by the National Science Foundation under Grant MCS 78-13642 and in part by the Joint Service Electronics Program (U.S. Army, U.S. Navy and U.S. Air Force) under Contract N00014-79-C-0424.

[a] Extracting a selected subset of elements from an ordered array means moving the subset to consecutive processors in an order preserving fashion.

0190-3918/81/0000/0304$00.75 © 1981 IEEE

determine which node number in the next level should be associated with each horizontal segment. We continue this process which geometrically traces a unique path, possibly two, from the root to a leaf.

The time complexity of this algorithm for reporting all intersecting pairs of n vertical and m horizontal line segments is $O((\log (n + m))^2 + k')$ with 4n + 2m processors, where k' is the maximum number of intersections per vertical line segment.

We now turn to the two-dimensional range searching problem. A point s in the set S of n points in the plane is represented by its coordinates X(s) and Y(s). A two-dimensional range search query q, in the set Q of m queries, asks for all the points s in set S such that $L(q) \leqq X(s) \leqq R(q)$ and $B(q) \leqq Y(s) \leqq T(q)$. Here, we assume the simple case: all X(s) are distinct integers in the interval [0, n - 1] and n is a power of 2. A binary search tree $\mathscr{H}$ similar to $\mathscr{T}$ can be produced for S. Node $N_i(j)$ represents the vertical interval $[j \cdot 2^i, (j + 1) 2^i - 1]$ and associated with it is a subset of points s, with $j \cdot 2_i \leqq Y(s) \leqq (j + 1) 2^i - 1$, sorted by X(s). In a manner similar to transforming $\mathscr{T}$ to $\mathscr{T}'$, $\mathscr{H}$ is transformed to a data structure $\mathscr{H}'$. We can then represent $\mathscr{H}'$ as a collection $\mathscr{P}$ of arrays, $P_{\log N}, \ldots, P_0$. The array P_i contains the set S sorted lexicographically by their node numbers and x-coordinates.

The construction of $\mathscr{P}$ is similar to $\mathscr{E}$: the set S is first sorted by their x-coordinates. The resulting array is $P_{\log N}$. We then determine the node numbers for the next level and rearrange the order of points according to their node numbers.

To answer the set Q of queries, we use $\mathscr{P}$ as a binary search tree. Initially, we sort Q by L(q). We extract, from Q, those q such that $B(q) \leqq 0$ and $N - 1 \leqq T(q)$. For those extracted queries q, a point s with $L(q) \leqq X(s) \leqq R(q)$, must satisfy q. Therefore, we can use one-dimensional range searching to find all the points in these extracted queries. For the remaining queries, we determine their node number in the similar manner as we determined those for the horizontal line segments. We proceed until all queries are answered.

The time complexity of two-dimensional range searching algorithm is $O((\log (n + m))^2 + k)$ with n + 4m processors, where k is the maximum number of inclusions per query.

The line segment intersection algorithm and the two-dimensional range searching algorithm are combined to give an $O((\log N)^2 + k')$ algorithm for reporting all k intersecting pairs of N rectangles with N processors, where k' ($\leqq$ k) is the maximum number of intersections per rectangle. With the best known serial algorithm requiring $O(N \log N + k)$ time, this algorithm yields a speedup of $O(N/\log N)$ and an efficiency of $O(1/\log N)$, which means that the algorithm is not only fast, but involves relatively little waste of processors. The formal descriptions and analysis of these algorithms are contained in [3].

In [3], it is shown that this method can be generalized when we have $N^{1+\alpha}$ number of processors, $0 < \alpha \leqq 1$, to improve the time complexity by a factor of $\alpha \log N$.

References

[1] H.S. Baird, "Fast Algorithms for LSI Artwork Analysis," Design Automation & Fault-Tolerant Computing (1978), pp. 179-209.

[2] J.L. Bentley and D. Wood, An Optimal Worst-Case Algorithm for Reporting Intersection of Rectangles, Computer Science Technical Report, McMaster University (1979).

[3] A.L. Chow, Parallel Algorithms for Geometric Problems, Ph.D. Thesis, Department of Computer Science, University of Illinois, Urbana, Illinois (October, 1980).

[4] U. Lauther, "4-Dimensional Binary Search Trees as a Means to Speed Up Associative Searches in Design Rule Verification of Integrated Circuits," Design Automation & Fault-Tolerant Computing (1978), pp. 241-247.

[5] D. Nassimi and S. Sahni, Parallel Permutation and Sorting Algorithms and a New Generalized-Connection-Network, Computer Science Department, Technical Report 79-8, University of Minnesota, Minneapolis, Minnesota (April, 1979).

[6] F.P. Preparata and J. Vuillemin, "The Cube-Connected-Cycles: A Versatile Network of Parallel Computation," Proc. 20th Annual IEEE Symp. on Foundations of Computer Science (October, 1979).

CACHE EFFECTIVENESS IN MULTIPROCESSOR SYSTEMS WITH PIPELINED PARALLEL MEMORIES[†]

Fäye A. Briggs and Michel Dubois
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907

Abstract -- A possible design alternative for improving the performance of a multiprocessor system is to insert a private cache between each processor and the shared memory. The caches act as high-speed buffers by reducing the memory access time, and they affect the delays caused by memory conflicts. In this paper, we study the effectiveness of caches in a multiprocessor system. The shared memory is pipelined and interleaved to improve the block transfer rate, and it assumes an L-M organization, previously studied under random word access. An approximate model is developed to estimate the processor utilization and the speedup improvement provided by the caches. These two parameters are essential to a cost-effective design. An example of a design is treated to illustrate the usefulness of this investigation.

1. Introduction

In this paper, we present simulation results and an approximate analytical model to evaluate the performance of cache-based multiprocessors with a shared memory (SM) as depicted in Figure 1. At the first level, each processor has a private cache (PC). The second memory level comprises the L-M memory organization, which consists of l lines and m memory modules per line [BRI 77]. A line is used to denote the address bus within the SM. Associated with each line is a Direct Memory Access (DMA) controller which receives a cache request for a block transfer of size b and issues b internal requests (IR) to consecutive modules on the line. In this paper, it is assumed that the interconnection network between the private caches and shared memory modules is a full crossbar.

In the architecture of Figure 1, there is a data coherence problem in which several copies of the same block may exist in different caches at any given time. When a processor attempts to write in a cache, all the copies in other caches must be updated before the process is allowed to proceed. Various stolutions to the cache coherence problem have been proposed [CEN 78, DUB 81a]. In summary, the cache coherence problem is solvable, and its impact on system performance can be minimized by efficient data-sharing mechanisms. In our study, the effect of the overhead caused by the enforcement of cache coherence is neglected. The program behavior in a processor will be characterized by its cache hit ratio, h, or miss ratio, 1-h. The determination of the hit ratio of a program as a function of cache size, set size and block size have been investigated by several authors [STR 76, RAO 78].

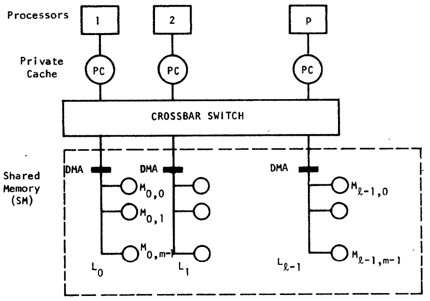


Figure 1. Cache-Based Multiprocessor System

Most studies to date evaluate the shared memory conflict problem for random word access [BRI 77]. We propose a model which is used to evaluate the degree of memory conflicts in a multiprocessor system with private caches. The model permits us to determine the processor utilization. Furthermore, we compare the effect of the cache on the speedup of the multiprocessing system. The performance of the system is a function of the cache miss ratio, cache organization, processor characteristics and the shared memory characteristics and configuration.

2. The Cache Model

The processor system consists of p identical and synchronized processors. In each of these processors, we assume that a machine cycle consists of an integer number, d, of cache cycles. An instruction cycle usually consists of an integer number of machine cycles. Typical machine cycles are instruction fetch, operand fetch and execution cycle, which may involve register-register or memory-register references. It is obvious that in some machine cycles of a processor, no cache memory references will occur. Therefore, let θ be the probability that a memory request is issued by a processor to the cache in a machine cycle. Thus, the fraction of references made by the processor to the cache in each cache cycle is $x = \theta/d$.

When the data requested by a processor is not in its private cache, a miss occurs that causes the cache controller to issue a shared memory request for a block transfer. In particular, if a read miss occurs, the block of shared memory words containing the location specified is transferred into the cache. We assume that no read-through strategy is implemented. If the cache is full, a cache replacement algorithm is invoked to decide on which block frame to free in order to create space for the new block containing the referenced data.

Cache management algorithms differ basically

0190-3918/81/0000/0306$00.75 © 1981 IEEE

in the method of resolving write misses. In a *write-through* strategy, a processor always writes directly in the shared memory, and possibly in the cache if the block is present. Consequently, a block is never copied to shared memory when a block frame is freed. However, such a policy requires buffering of the write requests. Moreover, the most efficient schemes to enforce cache consistency [CEN 78] are based on a *write-back-write-allocate* strategy. This policy is adopted in this paper. For a write hit, the data is written only in the cache. However, if a write miss occurs, the write-allocate policy is used to transfer the block containing the addressed word to the cache. Hence in our model, a read or write miss requires a block transfer to the cache.

If a cache block frame which has not been modified is to be replaced, it is overwritten with the new block of data. However, a modified block-frame that is to be replaced must be written to the shared memory (SM) before a block-read from the SM is initiated. In this case, two consecutive transfers are made between the cache and SM. We denote the probability of a cache hit by h and assume that each time a cache miss occurs, a block-write to SM is required with a probability w, followed by a block-read from SM.

Two methods of organizing the cache for block reads and writes are investigated. In one case, it is assumed that the two consecutive block transfers (one block-write followed by one block-read) are made between a processor and the same SM line. This assumption will be satisfied if a *set-associative* cache is used in which all the blocks that map to the same set are stored on the same SM line. Hence, in this method a cache miss requires the transfer of a 2b-word block with a probability w and the transfer of a b-word block with a probability 1-w.

A second method of organizing the cache assumes that the two consecutive block-write and block-read requests are considered independent and hence have equal probability of referencing any SM line (assuming independent reference model). This assumption may be valid in a *fully associative* cacne. The effect of making two consecutive and independent block requests from a processor is to increase the effective rate of requests to the SM.

In our models, the hit ratio is given. Generally, the hit ratio depends on the locality property of the program mix, the cache replacement policy, and the block, set and cache sizes. Various studies have addressed this relationship. In [LEH 80], a program is characterized by a simplified mathematical model based on its instruction mix and the model is used to estimate the hit ratio. Smith compares different cache replacement policies [SMI 78]. Under the assumption of a linear paging model, he shows that the ratio of the miss ratios between the set associative and the fully associative caches is

$$R(i,N) = \frac{i - 1/N}{i - 1} \quad \text{for } i \geq 3,$$

where i is the set size (number of blocks in a set) and N is the number of sets. This ratio is always ≥ 1. It tends to 1 when Ni, the cache capacity, increases without limit. This relationship should be considered when comparing the set associative and fully associative caches from our models. Finally, the hit ratio is also a function of the cache and block sizes. Strecker presents empirical results for the PDP-11 family computers [STR 76]. In [RAO 78], an analytical model is proposed. For a *given cache size*, the hit ratio improves as the block size increases from 1, due to the locality of the references to the cache. However, beyond a certain block size, the hit ratio decreases. This is due to the decrease in the usefulness of the extra words in a block as the block size increases. For a *given block size*, the hit ratio increases monotonically with the cache size. If the hit ratio can be determined empirically on a uniprocessor machine [STR 76] or theoretically [RAO 78, LEH 80], the models given in this paper can then be applied to evaluate the various performance indices. We will assume that the cache size is adapted to obtain a given hit ratio.

For practical purposes, the absolute size of the private cache would be expected to be large enough to accommodate at least the "working set" of the process so that the miss ratio, 1-h, is small (in the order of 0.1). Furthermore, we assume that the block transfer time is also small (less than 64 cache cycles). Under these conditions, it is not necessary to perform a task switch on a cache miss to another runable process. Therefore, in this paper we shall assume that, on a cache miss, the processor enters a *wait state* while waiting for service of the desired block request, and into a sequence of *transfer states* while the block is being serviced. If a processor is not in the wait or in the set of transfer states, it is said to be in the *active state*. Hence, the processor utilization can be computed from the fraction of time spent in the active state. Finally, associated with each DMA controller of a memory line is a buffer which queues the requests for block transfers. The DMA controller schedules these requests to the line, using a First-Come-First-Served (FCFS) policy.

3. The Shared Memory Organization

The shared memory configuration is derived from the L-M organization, which exploited the timing characteristics exhibited by semiconductor memories with address latches [BRI 77]. The address cycle or hold time, a_0, which is the minimum duration that the address is maintained on the address bus of the shared memory module for a successful memory operation, is usually less than the shared memory cycle, c_0. Throughout this paper, we assume that the basic unit of time is the cache cycle, whicn is equal to τ seconds. If the address and shared memory cycles are quantized so that they are expressed as an integer number of cache cycles, then

$$a = \lceil a_0/\tau \rceil \text{ and } c = \lceil c_0/\tau \rceil ,$$

so that a set of modules can be multiplexed on a line. In general, $1 \leq a \leq c$. When a memory operation is initiated in a module, it causes the associated line to be active for a units of time and the module to be active for c units of time. The shared memory, which consists of $N = 2^n$ interleaved identical memory modules, is organized in a matrix form in order to exploit the memory module characteristics (a,c). As shown in Figure 1, a particular memory configuration (l,m) consists of $l = 2^\beta$ lines and $m = 2^{n-\beta}$ lines and modules per line, such that $lm = N$, for integer $\beta \geq 0$. The blocks in the memory are interleaved on the lines so that block i is assigned to modules on line i mod l. It should be noted that this does not contradict the assumption made earlier that blocks of the same set are on the same line for the set associative cache model.

Since the shared memory is used in the block transfer mode in this paper, we will assume an address cycle of a = 1 for the shared memory, in order to effectively utilize the line. However, if a > 1 for a particular type of memory, the address cycle could be made equal to 1 by incorporating an appropriate address latch in each SM module. Since a = 1, the memory module will be characterized by c in the rest of the paper. The model developed in [BRI 77] is not applicable here, since it was for single-word transfers that are requested by pipelined processors. In order to effectively utilize the SM modules for block transfers, the modules on a line are interleaved in a particular fashion, so that the servicing of two SM requests could be overlapped on the same line. The SM modules on a line are interleaved so that a block of data of size $b = 2^\alpha$ is interleaved on consecutive modules on that line. Let line i and module j on that line be referred to as L_i and $M_{i,j}$ respectively for $0 \leq i \leq l-1$ and $0 \leq j \leq m-1$. Then, the k-th word of the block of data that exists on line i is in module k mod m on that line for $0 \leq k \leq b-1$. It is important to note that the first word of a block that exists on line i is in the first module, $M_{i,0}$, of that line. In this paper, we assume that $b \geq m$. If $b < m$, memory modules $M_{i,b}, M_{i,b+1}, \ldots, M_{i,m-1}$, will not be utilized, since a block starts in module $M_{i,0}$.

When an SM block request is accepted by a line i, the DMA controller at that line issues b successive internal requests (IR) to consecutive modules on line i, starting from module $M_{i,0}$. It is assumed that these internal requests are issued at the beginning of every time unit. Therefore, the internal request for the k-th word of the block will be issued to module $M_{i,j}$, where $j = k \bmod m$ for $0 \leq k \leq b-1$. It is obvious that this set of b internal requests is not preemptible. Note that if b > m or if the cache is set associative, the (m+1)st internal request is for module $M_{i,0}$. Consequently, the first request must be completed by the time the (m+1)st internal request issued. This constraint is satisfied if $c \leq m$.

4. Performance Analysis

In this section we present assumptions and develop the models that permit us to evaluate the various performance indicators of the cache-based multiprocessor system.

4.1 Hybrid Simulation

For analytical purposes, it is assumed that cache requests to SM are random and uniformly distributed over all l lines of the SM. This assumption is justified by the interleaving of the blocks across the lines. One inference that can be made directly from the above assumption is that the probability of a request addressing any module is $1/\ell$.

In order to understand the timing characteristics of the servicing of requests for block transfer, we define the time instants t^- and t^+ as $\lim_{\Delta t \to 0} (t-\Delta t)$ and $\lim_{\Delta t \to 0} (t+\Delta t)$, respectively, for $\Delta t > 0$. A time unit <t,t+1> may be thought of as beginning at time t^+ and ending at time $(t+1)^-$. Hence, since a = 1, the successive Internal Requests which are generated to a line in the servicing of an SM request, do not encounter any conflicts.

Recall that when an SM request for a block transfer is accepted, the DMA controller issues b successive IRs. If the request is accepted on line i at time t, then the IR for the k-th word of a block of size b is initiated at time t+k to module $M_{i,j}$, for $j = k \bmod m$ and $0 \leq k \leq b-1$. Since the SM module cycle time is c, module $M_{i,j}$ will be busy in the intervals <t+k, t+c+k> for the values of j.

Since $b = 2^\alpha \geq m = 2^{n-\beta}$, then $\frac{b}{m}$ is an integer ≥ 1. Therefore, each module on a line i which accepts an SM request for block transfer at time t receives $\frac{b}{m}$ internal memory requests. In particular, the last IR to module $M_{i,0}$ is made at time $t + (\frac{b}{m} -1)\, m = t + b-m$. Thus, the last interval that module $M_{i,0}$ is busy (during the current block transfer) is <t+b-m, t+b-m+c>. After this period, a new block transfer which addresses line i can be accepted. Since the current block transfer was initiated at time t, all block transfer requests arriving at t+1,t+2,...,t+b-m+c-1 will find line i busy. Note that to an SM request, the line is busy for b-m+c time units. We refer to this as the line service time. The actual service time of the SM request is b+c-1. This is the time taken to access and transfer a block of size b when the request is accepted. Since we do not implement a read-through policy in the cache model, the processor goes through a sequence of transfer states having total duration b+c-1 before returning to the active state. That is, the block transfer must be completed before the processor can become active again.

The cache-based multiprocessor system may be modeled as a closed queueing network shown in Figure 2. This network has been called the "central server model" [KLE 76]. The servers are the shared memory lines and the requests are issued by a set of p processing nodes, each of which

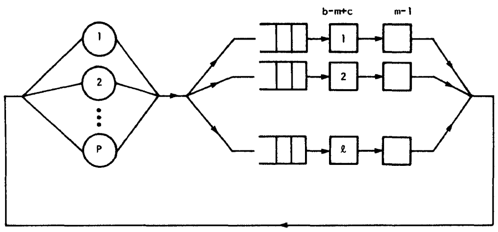


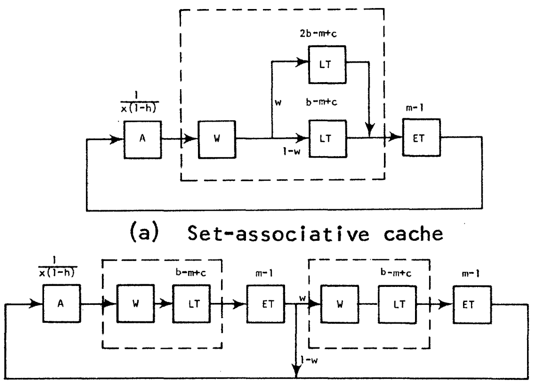
Figure 2. Central server model for the cache system.

lumps a processor with its local cache. The two segments of a server model each SM line and reflect the pipeline effect of the LM memory described above.

The behavior of each processor is illustrated in Figure 3 for both cache strategies. Node "A" denotes an active state of the processor and node "W," a waiting state. Node "LT" represents the state for the first part of a transfer during which the line is kept busy (line service time), and "ET," the state in which a transfer is completed without holding a line. These states have to be distinguished because of their different properties. Note that the state representations and their interconnections as shown in Figure 3 do not constitute Markov graphs, since each state has a different average duration. These average durations are indicated on the graphs. The state of each processor changes asynchronously in an SM request cycle. The SM request cycle is the total average time spent in the active state, wait state, and set of transfer states (LT and ET). According to the model assumptions, the visit time (expressed in units of cache cycles) in state A is geometrically distributed with mean 1/x(1-h), and the visit time in a state ET is constant with value m-1.

For the set-associative cache (Figure 3a), if a block-write is not required (with probability 1-w) on a cache miss, then the line which accepts the SM request is busy for b-m+c time units. However, if a block-write is required (with probability w) in addition to the block-read, then two consecutive block transfers (each of size b) are made uninterruptedly on the same SM line. In this case, the line that accepts the request is busy for 2b-m+c time units.

The case of the fully associative cache is simpler (Figure 3b): if a cache miss requires a block-write (with a probability w) followed by a block-read, the processor submits these requests as two successive and independent requests to transfer a block of size b in each case. Each of the two corresponding LT states thus have a constant duration b-m+c.



(a) Set-associative cache

(b) Fully associative cache

Figure 3. State representation for each cache implementation.

In both cases, each processor goes through "independent" states (states A and ET), followed by "interactive" states (states W and LT). When in an independent state, a processor can proceed freely and does not interfere with the progress of other processors. Interactive states are characterized by a potential for conflicts with other processors. The interactive states are framed in Figure 3. To estimate the average visit time in such states, simulations are required. Note that the foregoing analysis that leads to the state representation of Figure 3 simplifies the simulation significantly. Such an approach has been called "hybrid simulation" [SCH 78]. Table 1 is a compilation of some of the simulation results for a realistic case (w = 0.3, c = 4, x = 0.4, h = 0.95, m = 4, p = 16). The number of lines, l, and the block size, b, are variable. The performance index is the average processor utilization, defined as the average fraction of time spent by each processor in an active state. Both cache implementations have practically the same performance, for <u>the same value of the hit ratio</u>.

Table 1. Processor utilization for the set associative cache
(c=4, w=0.3, m=4, x=0.4, h=0.95, p=16).

l	b	Simulation	Model	Error (%)
1	4	0.593	0.601	+1.3
	8	0.299	0.300	+0.3
	16	0.150	0.150	0.0
	32	0.076	0.075	-1.3
	64	0.038	0.038	0.0
2	4	0.797	0.814	+2.1
	8	0.540	0.578	+7.0
	16	0.281	0.300	+6.8
	32	0.143	0.150	+4.9
	64	0.073	0.075	+2.7
4	4	0.839	0.842	+3.6
	8	0.711	0.729	+2.5
	16	0.474	0.514	+8.4
	32	0.257	0.286	+11.2
	64	0.133	0.147	+10.5
8	4	0.850	0.852	+0.2
	8	0.759	0.764	+0.7
	16	0.596	0.614	+3.0
	32	0.385	0.414	+7.5
	64	0.215	0.238	+10.6
16	4	0.855	0.856	+0.1
	8	0.776	0.777	+0.1
	16	0.644	0.650	+0.9
	32	0.463	0.475	+2.6
	64	0.288	0.301	+4.5

Since these simulations are still expensive, despite the simplification, we have developed an approximate analytical model to estimate the processor utilization.

4.2 Approximate Analytical Model

The processor's behavior shown in Figure 3 are quite complex to model exactly. We propose an approximate model based on a method applied in [HOO 77] for the modeling of random word accesses in multiprocessor memories. We number the processors from 1 to p and the memory line from 1 to l. Let

$$I_k(t) = \left[i_{k,1}(t), i_{k,2}(t), \ldots, i_{k,p}(t)\right]$$

for $k = 1,\ldots,l$,

with $i_{k,j}(t) = 1$ iff processor j <u>is not</u> waiting for or using line k,
and $i_{k,j}(t) = 0$ iff processor j <u>is</u> waiting for or using line k at time t.

$I_k(t)$ is called the <u>indicator vector</u> for line k at time t. Each component $i_{k,j}(t)$ indicates whether or not processor j is waiting for or holding line k. Note that a processor waits for or holds a line whenever it is in state W or LT (interactive states), respectively. Let Y be the average fraction of time a given processor is in an independent state. Y is also the probability of being in such a state by the ergodic property [KLE 76]. The symmetry of the system implies the same value of Y for all the processors. Similarly, let X_s be the probability that a given line is busy and S, the average line service time of a request. Then

$$\begin{aligned} X_s &= \text{Prob["at least one processor is waiting for or holding a given line k"]} \\ &= 1 - \text{Prob["no processor is waiting for or holding line k"]} \\ &= 1 - \text{Prob["}i_{k,1} \cdot i_{k,2} \cdots i_{k,p} = 1\text{"]} \qquad (1) \\ &= 1 - E[i_{k,1} \cdot i_{k,2} \cdots i_{k,p}]. \end{aligned}$$

This last equality results from the fact that the expectation of a random variable taking only the values 0 and 1 is equal to the probability of the variable being 1. The rate of <u>completed</u> requests by a line is

$$X_s/S \qquad (2)$$

In equilibrium, this rate can be equated to the rate of <u>submitted</u> requests to a line. To compute this second member of the equation, we have to distinguish between the two cache implementations.

<u>4.2.1 Set-Associative Cache (Figure 3a)</u>. A processor submits a request whenever it exists state A. This occurs, for each processor, whenever a cycle in the network of Figure 3a is completed. Let C be the average time taken by such a cycle. From the definition of Y, we have

$$Y = \frac{1/x(1-h) + m-1}{C} .$$

The rate of submitted requests to the SM by any one processor is 1/C. Since there are p requesting processors and each request is submitted randomly to any one of the l lines, the average rate of submitted requests to a given line k is

$$\frac{1}{C} \cdot \frac{p}{l} = \frac{Y}{\frac{1}{x(1-h)} + m-1} \cdot \frac{p}{l} . \qquad (3)$$

Equating (3) and (2), and since for the Set Associative Cache (see Figure 3a), S = b(1+w)-m+c, one finds that

$$X_s = \frac{S}{\frac{1}{x(1-h)} + m-1} \cdot \frac{p}{l} \cdot Y = \rho Y$$

$$\text{with } \rho = x(1-h) \cdot \frac{p}{l} \cdot \frac{b(1+w)-m+c}{1+(m-1)x(1-h)} . \qquad (4)$$

Combining equations (1) and (4) we have

$$E[i_{k,1} \cdot i_{k,2} \cdots i_{k,p}] + \rho Y = 1 . \qquad (5)$$

This equation is exact for the set-associative cache. However, the first term of the L.H.S. of the equation is very complex to estimate in general. The approximation consists in neglecting the interactions between processors. As a result of the approximation, the components of $I_k(t)$ are not correlated. This approximation performs best for a short and deterministic line service time. Indeed, large instances of the line service time are more likely to result in instantaneous longer queues and more interactions between the processors. Under the non-correlation conditions,

$$E[i_{k,1} \cdot i_{k,2} \cdots i_{k,p}] = E[i_{k,1}] \cdot E[i_{k,2}] \cdots E[i_{k,p}].$$

If we denote by Z the fraction of time spent by each processor waiting for or holding a given line k, equation (5) becomes

$$(1-Z)^p + \rho Y = 1 , \qquad (6)$$

because of the symmetry of the system.

On the other hand, since a processor is either in an independent state (A or ET), or in an interactive state (waiting or holding one of the lines), then by the law of total probability in a system with l lines we have

$$Y + l \cdot Z = 1. \qquad (7)$$

Using (6) with the condition that $1 - \rho Y > 0$,

$$Z = 1 - (1-\rho Y)^{1/p} .$$

Consequently, by the substitution for Z in (7) and rearranging, we obtain

$$Y = \frac{1}{\rho} \left[1 - \left(\frac{Y+l-1}{l}\right)^p\right] . \qquad (8)$$

Since Y is the average fraction of time spent in a state A or ET, the processor utilization, u, which is the fraction of time spent in state A is

$$u = \frac{1/x(1-h)}{C} = \frac{Y}{1+x(1-h)(m-1)} . \qquad (9)$$

Besides being a good approximation for short line service times with low coefficient of variation, the approximation (8) was proven in [DUB 81b] to have the following desirable properties.

<u>Property 1</u>: when p tends to ∞ (and all other parameters are kept constant), Y tends to $1/\rho$.

Property 2: equation (8) has one unique real solution between 0 and Min $(1/\rho, 1)$.

As a consequence of the first property, the approximation is correct asymptotically, when the traffic at the memory (and thus the interactions between processors) increases. This can be seen as follows. When the number of processors increases, the system of Figure 2 tends to saturate [KLE 76]. Under saturated conditions, each line is constantly busy, which means that X_s tends to 1 for all the lines. Equation (4) shows then that Y tends to $1/\rho$.

4.2.2 Fully-Associative Cache. A cycle through the network of Figure 3b may result in one request (with probability (1-w)) or two requests (with probability w). If C is the average cycle time, the rate of submitted requests by any one processor to the memory is (1+w)/C. For the case of Figure 3b,

$$Y = \frac{1/x(1-h) + (m-1)(1+2)}{C} .$$

Following the same reasoning as the one leading to equation (8), one finds that equation (8) is also applicable to the Fully Associative Cache Model with

$$\rho = \frac{p}{l} \cdot \frac{(b-m+c)(1+w)}{\frac{1}{x(1-h)} + (m-1)(1+w)} . \qquad (10)$$

To obtain the processor utilization, we note that

$$u = \frac{1/x(1-h)}{C} = \frac{Y}{1+(m-1)(1+w)x(1-h)} . \qquad (11)$$

The Newton iterative method converges rapidly with an initial value $Y_0 = 0.5$.

4.2.3 Accuracy of the Approximate Model. To check the accuracy of the approximate model, we have compared it with the hybrid simulation. Some results are shown in Table 1 that are typical. The model is adequate for parameter values corresponding to an effective design, and it is able to detect a poor design. In Figure 5, the results of the analytical model for the set associative cache was used to plot the processor utilization. The memory cycle time, c, was also varied; all the other parameters were kept the same as for Table 1. The simulation points are linked to their analytical estimate in Figure 5. The analytical model tends to slightly overestimate the utilization.

4.3 Speedup of the Cache-Based Multiprocessor

We can derive the speedup of the cache-based multiprocessor system (called system 1) over another system (called system 2) without caches. Of course, the two systems have identical parameters. That is, each of them consists of a p processor system that has a shared memory with l lines and m modules per line. However, in system 2, there is no block transfer, since it is not a cache-based system. The memory modules are interleaved for single word accesses, as discussed in [BRI 77]. Each memory reference requires a single word transfer. Since we assume the same memory parameters, the address cycle is a = 1, and the memory cycle is c. In this case the probability of acceptance of a memory request for system 2 is (see [BRI 78])

$$P_{A_2} = \frac{1 - P_1}{1 + \frac{r'(c-1)(1-P_1)}{N}} , \qquad (12)$$

where $1 - P_1 = [1 - (1 - \frac{r'}{l})^p] \frac{l}{r'p}$; and $N = l \cdot m$, and r' is the effective probability of a processor making a memory reference in a time unit.

The instruction mix parameter, θ, is the same for both systems. Recall that θ is the probability of a memory request being issued by a processor in a machine cycle. Hence for system 2, the fraction of references made by the processor to the SM in each time unit is $x' = \theta/T_2$, where T_2 is the machine cycle time for processors in system 2. Note that x' is the probability that a processor of system 2 makes a memory reference at any time t, when the processor is in the active state.

Since we assume that systems 1 and 2 have identical processors, the absence of the cache in system 2 and the service of each memory reference in the SM elongates the machine cycle time of each processor from T_1 (in system 1) to T_2 (in system 2). It can be easily seen that $T_2 = T_1 - 1 + c$ time units. Note that a cache cycle = 1 time unit. From section 2 of this paper, T_1 consists of an integer number, d, of cache cycles. In system 2, a memory reference to SM may encounter a delay in service due to memory conflicts. We denote by A the state in which the processor is active, and by W the state in which the processor is waiting. Furthermore, state B_i denotes the state of the service of the SM request for i = 1,2,..,c. Figure 4 depicts the simple Markov graph for the processor in system 2. It should be pointed out that the scheduling of requests for single-word transfers in this system is not FCFS as in the system with caches. Rejected requests are resubmitted one cache cycle later until they are accepted. These resubmitted requests are assumed to be independent and hence have equal probability of referencing any of the l lines. This schedule, which is also depicted by the Markov graph of Figure 4, would slightly overestimate the true performance of the system without caches. Simulations indicate that the overestimation is within about 5% [BRI 78]. A previous model in which the SM block requests are

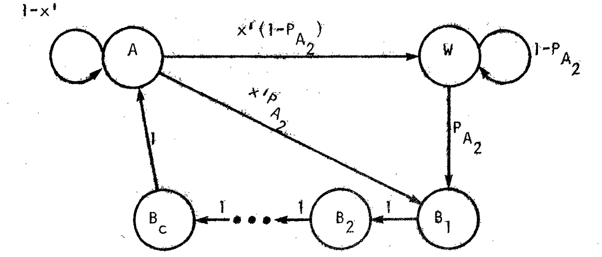


Figure 4. State diagram of a processor's transition in system without cache

not buffered was studied in [BRI 81]. The buffered model presented here is more accurate but not applicable to other interconnection networks.

Let q_A and q_W represent the probabilities of being in states A and W, respectively. A solution to this graph yields

$$q_A = \frac{P_{A_2}}{P_{A_2}+x'[(1-P_{A_2})+cP_{A_2}]},$$

$$q_W = \frac{x'(1-P_{A_2})}{P_{A_2}+x'[(1-P_{A_2})+cP_{A_2}]}. \qquad (13)$$

The processor utilization for system 2 is $u_2 = q_A$. It can be seen that the effective request rate is

$$r' = x'q_A + q_W = \frac{x'}{P_{A_2}+x'[(1-P_{A_2})+cP_{A_2}]}. \qquad (14)$$

Again, an iterative technique can be applied to obtain a solution to the utilization, $u_2 = q_A$.

Let us represent the utilization of system 1 by u_1. This can be obtained from equation (9) or (11), for the set associative or fully associative cache models, respectively. The effective machine cycles for a processor of systems 1 and 2 are T_1/u_1 and T_2/u_2, respectively. Since $T_2 = T_1 + c-1$, and $T_1 = d$, the speedup for the p processor system is

$$S_P = \frac{T_2/u_2}{T_1/u_1} = \frac{T_2}{T_1} \cdot \frac{u_1}{u_2} = (1 + \frac{c-1}{d}) \cdot \frac{u_1}{u_2}. \qquad (15)$$

The evaluation of the speedup permits us to compare the effectiveness of the cache in the multiprocessor system. Certainly, this speedup is a function of many parameters. The discussion of the results given in the next section exposes the effects of the variability of these parameters on the system performance.

5. Discussion and Conclusion

In the following discussion, we assume that the multiprocessor system consists of p = 16 processors with private caches. The machine cycle time of the cache based system is d = 2 (all times are expressed in units of cache cycles), and the instruction mix parameter, θ, is 0.8. The shared memory has an L-M configuration with m = 4, and l (a power of 2) is between 1 and 16. Thus the total number of modules is variable. Other parameters of the study are b, the block size, ($4 \leq b \leq 64$), c, the memory cycle time ($2 \leq c \leq 8$) and h, the cache hit ratio (h = .95). Note, however, that for a given cache size, the hit ratio and the block size are not independent, as observed in [STR 76]. In this study, we assume that, for a given block size, a cache with an appropriate size is selected so that the hit ratio is kept constant.

The processor utilization (u) is a performance index reflecting the degree of match between the processors and the memory organization. The throughput improvement provided by the introduction of caches is measured by the speedup (S_p), as defined in section 4.3. Note that u and S_p are not necessarily related: a system with high "S_p" may have an unacceptably low "u." In general, one desires a design with high processor utilization and speedup to justify the investment in faster processors and expensive cache memories, respectively. For the parameters chosen, there is little difference in the results of the set associative and fully associative cache models. Hence, only the results for the set associative model are given in the figures. To limit the cost, the analytical models are used to derivate the following curves.

A comparison of figures 5 and 6 shows that the

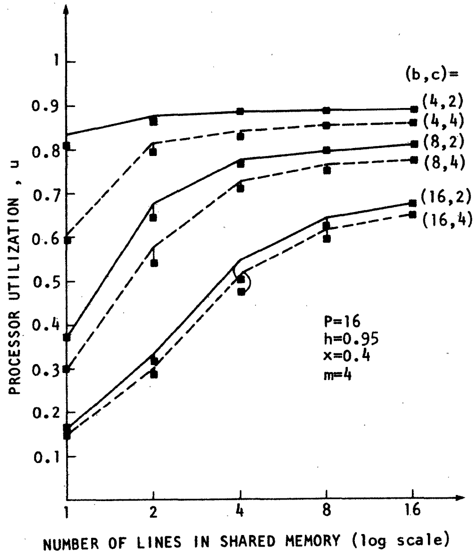


Figure 5. Processor utilization for the set-associative cache

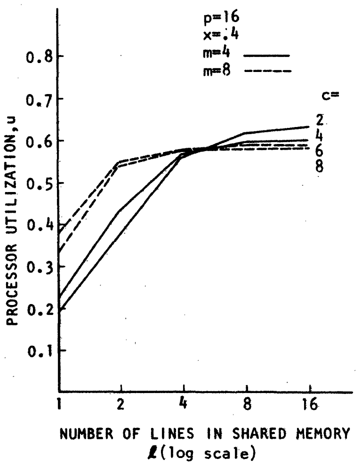


Figure 6. Processor utilization for multiprocessor without caches

caches can have a dramatic effect on processor utilization. In general, an increase in the block size causes a significant deterioration of the processor utilization. However, for a small block size (4), the processor utilization for the system with cache is much better than for the system without cache. This improvement is more dramatic when the memory cycle time, c, is large. Figure 5 also shows that increasing the number of memory lines, l, is not always cost-effective.

The following throughput comparisons between two systems emphasize the design alternatives offered by the use of private caches. Both systems consist of 16 processors. In system 1, a private cache is added to each processor; the memory configuration is characterized by m = 4 and $1 \leq l \leq 16$. Hence, the cache controllers access the SM via a 16 by l crossbar switch. In system 2, the processors are connected to an L-M memory with l = 16 and m = 4 through a 16 x 16 crossbar switch. All the other parameters are as described earlier in this section.

Decreasing l reduces the total number of memory modules and hence the cost of the decoder. The most significant effect of l in a system is the reduction in complexity of the processor-memory interconnection network and hence the cost. Figure 7 shows the effective speedup achieved by the inclusion of cache memories into system 2 and the simultaneous reduction in the number of lines l. It can be seen that a significant improvement in the system throughput is still achievable by the simultaneous reduction in l and the inclusion of cache memories. This performance improvement is even more pronounced for large values of c. A possible significant reduction in l gives the designer a choice. If for a small number of lines, l < 16, the incorporation of a per processor cache results in a speedup, $S_p \geq 1$, the designer can consider trading off low-cost multiport memories for the expensive 16 x 16 crossbar switch used in the system without caches. In fact, as shown in Figure 7, the incorporation of a per processor cache results in significant speedup in most cases, even for small l.

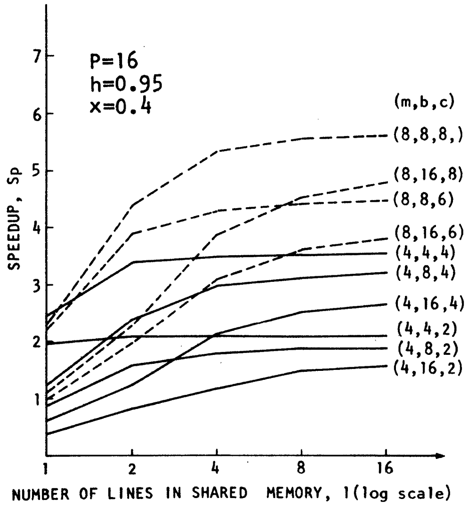


Figure 7. Effective speedup for cache-based system, where system without cache has ℓ=16.

REFERENCES

[BRI 77] F. A. Briggs, and E. S. Davidson, "Organization of Semiconductor Memories for Parallel Pipelined Processors," IEEE Transactions on Computers, Vol. C-26, February 1977, pp. 162-169.

[BRI 78] F. A. Briggs, "Performance of Memory Configurations for Parallel-Pipelined Computers," Proceedings of the 5th Annual Symposium on Computer Architecture, (1978), pp. 202-209.

[BRI 81] F. A. Briggs and M. Dubois, "Performance of Cache-based Multiprocessors," ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, September 14-16, 1981.

[CEN 78] C. M. Censier and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems," IEEE Transactions on Computers, Vol. C-27, No. 12, December 1978, pp. 1112-1118.

[DUB 81a] M. Dubois and F. A. Briggs, "Efficient Interprocessor Communication for MIMD Multiprocessor Systems," Proceedings of the 8th Annual Symposium on Computer Architecture May 1981.

[DUB 81b] M. Dubois and F. A. Briggs, "Modeling of MIMD Algorithms for Multiprocessor Systems," Purdue University, Technical Report TR-EE 81-13, 1981.

[HOO 77] 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.

[KLE 76] L. Kleinrock, Queueing Systems, Volume 2: Computer Applications, New York: Wiley and Sons, Inc., 1976.

[LEH 80] A. Lehman, "Performance Evaluation and Prediction of Storage Hierarchies," ACM SIGMETRICS Performance '80, Vol. 9, No. 2, pp. 43-54, May 1980.

[PAT 79] J. H. Patel, "Processor-Memory Interconnections for Multiprocessors," Proceedings of the 6th Annual Symposium on Computer Architecture, April 1979, pp. 168-177.

[RAO 78] G. S. Rao, "Performance Analysis of Cache Memories," Journal of the ACM, Vol. 25, No. 3, July 1978, pp. 378-395.

[SCH 78] H. G. Schwetman, "Hybrid Simulation Models of Computer Systems, "Comm. of the ACM, Vol. 21, No. 9, September 1978, pp. 718-723.

[SMI 78] A. J. Smith, "A Comparative Study of Set Associative Memory Mapping Algorithms and Their Use for Cache and Main Memory," IEEE Trans. Software Engr., Vol. SE-4, No. 2, March 1978, pp. 121-130.

[STR 76] W. D. Strecker, "Cache Memories for PDP-11 Family Computers," Proceedings of the 3rd Symposium on Computer Architecture, pp. 155-158, January 1976.

†This research was supported by NSF Grant MCS-78-18906.

A PERFORMANCE MODEL FOR MULTIPROCESSORS WITH PRIVATE CACHE MEMORIES

Janak H. Patel
Coordinated Science Laboratory
University of Illinois
Urbana, Illinois 61801

1. Introduction

This paper presents analytic and simulation results for multiprocessors with two-level memory hierarchy of the type shown in Fig. 1. The first level of memory is a private cache and the second level of the memory is the main memory shared by all processors. The two levels are connected through a switch. In this paper we shall restrict ourselves to switches which are full crossbars or delta networks [PAT79]. The approximate analytical model presented is simple but remarkably close to the extensive simulation results.

2. The Physical Model

2.1 The Cache: For the purposes of this paper a cache-main hierarchy will be assumed to have an acceptable miss ratio less than 0.1 and a block size small enough and/or memories fast enough so that a block transfer takes no greater than about 64 cache cycles. An implication of the small block transfer time is that in a cache-main system it is not profitable to switch processes on a cache miss, because a process switching time is comparable to a block transfer time and also because the cache is not large enough to hold more than one working set for an acceptable miss ratio. Therefore, we can assume that on a cache miss, the processor is idle while the desired block is being transferred. Thus the system throughput of cache-main multiprocessor can be computed directly from the total time spent in doing block transfers, if the processor execution is not overlapped with a block transfer.

2.2 Cache-Main Interconnection: Two interconnection networks that will be studied here are full crossbars and delta networks. Both networks will be used here in the circuit switching mode. Once a fault occurs in a cache, the fault handling hardware requests a block transfer from a particular main memory module and the network establishes a path between the cache and the main memory module. This path is held until the memory transaction is complete. The path cannot be preempted by any other requests coming from other cache modules. This description implicitly assumes that a block resides in a single memory module. However, a memory module itself may be interleaved to increase its bandwidth. The advantage of using circuit switching and storing the block in one memory module is the reduction in block transfer time. Both in the crossbar and delta network there is an initial delay in establishing a path due to arbitration, decoding and setting of appropriate switches. Once the path is established the data can be transferred at a high rate.

3. Analysis

3.1. Simple Cache Model: In this section we develop an analytical model for a very simple cache organization. This will be extended later to include more complex cache organizations. The following assumptions define the simple cache model.

1. Each cache fault involves one block-write to a main memory module followed by one block-read from the same memory module. As a consequence of this, we further assume that once a path is established between the faulting cache and the requested memory module, the transaction (read and write of a block) takes a constant time equal to t CPU cycles.
2. Cache requests to main memory are random and uniformly distributed over all main memory modules.

The miss ratio of a program as a function of cache size, block size and set size have been measured by several researchers [KAP73,STR76,PEU77 YEH81]. From such data it is possible to determine the request rate from a cache to the main memory. Let m be the probability that a cache makes a request to main memory in a given time unit, that is, m is the probability that the processor makes a request to the cache and that it is a fault. m typically would be less than the miss ratio, because not every CPU cycle is a memory reference.

To summarize the simple cache model, at each time unit a cache makes a request to the main memory with probability m, after some wait time a transaction between a main memory module and the faulting cache takes place, which lasts for t time units. Throughout this period the processor remains idle.

3.2 analysis of simple cache model: A processor in our multiprocessor is in one of two states. It is either busy doing useful work or it is idle waiting for a cache-fault service to be completed. The throughput of the system is directly proportional to the processor utilization. Therefore, we shall use the processor utilization as a measure of the system performance. This can be computed as follows.

Consider Fig. 2, which shows the effect of cache faults and wait times on the processor activity. Since each processor cycle generates a cache fault with probability m, there are on average mk faults for k units of useful computation. Let w be the average wait encountered at each request. Since a block transfer takes t time units, the k units of useful processor activity takes k + mk(w+t) time units. Assuming N processors and M main memory modules the following can be computed directly from Fig. 2c. The processor utilization,

$$U = k/[k+mk(w+t)] = 1/[1+m(w+t)] \qquad (1)$$

This research was supported in part by Joint Services Electronics Program (U.S. Army, U.S. Navy and U.S. Air Force) under Contract N00016-79-C-0242.

The average number of busy memory modules,

$$B = Nmkt/[k+mk(w+t)] = Nmt/[1+m(w+t)] \qquad (2)$$

In terms of utilization U, $B = NmtU \qquad (3)$

In the above expressions, the only unknown is the average wait time w. It is clear that the wait time depends on several factors, such as, request rate m, number of processors N, number of memory modules M, block transfer time t and the type of the interconnection network. Exact Markov analysis is always possible for specific numeric values of m,t,N and M, because of finite number of states. However, the state space is very large and therefore computationally very complex. In the absence of a reasonable analysis, simulation is the only other viable alternative. We have done extensive simulation of the cache model. One important outcome of the simulation was an observation that the processor utilization of given sized multiprocessor system with a specific network (crossbar or delta) can be approximated as a function of the product mt, where m is the probability of a cache-to-main request and t is the block transfer time.

Consider the Fig. 2c once again which shows the activity of a single processor. While the processor is waiting, the cache is resubmitting the block transfer request again and again until it is accepted by the network; on the average this happens for w time units. After the request is granted, the network holds a path to a memory module for t time units. One can view this as t consecutive requests to the same module, each request requiring one time unit of service. Thus on each cache fault, the network sees an average of w+t consecutive requests for unit service time. Refering to Fig. 2c, in k+mk(w+t) time units a total of mk(w+t) requests for unit service are made to the network. Therefore, the request rate (for unit service) from a cache module as seen by the network is

$$m' = m(w+t)/[1+m(w+t)] \qquad (4)$$

In terms of processor utilization U of equation (1) we have

$$m' = 1 - U \qquad (5)$$

The approximation that we introduce here is that w+t consecutive requests to a single memory module can be decomposed into w+t separate requests which are random, independent and uniformly distributed over all memory modules, without essentially changing the system behavior. The model that we will analyze is a system of N sources and M destinations, each source generates a request with probability m' in each time unit. The request is independent, random and uniformly distributed over all destinations. Each request is for one unit service time. Rejected requests are resubmitted as new independent requests and are made part of the new request rate m'. First we analyze the system with a crossbar and then with a delta network.

The crossbar: We already have the average number of busy memory modules B from equation (3), namely B = NmtU. We can compute the same quantity another way. Each main memory module is addressed with probability m'/M from a cache. The probability that none of the N cache modules make a request to a particular main memory module is $(1-m'/M)^N$. Therefore on the average $M[(1-m'/M)^N]$ modules are not doing any memory transfers. In other words, the average number of busy modules is

$$B = M[1 - (1 - m'/M)^N] \qquad (6)$$

substituting for B = NmtU from Eq. 3 and m'=1-U from eq. 5 we have,

$$NmtU - M[1 - (1 - (1-U)/M)^N] = 0 \qquad (7)$$

The above equation in U can be solved by standard numeric algorithms using iterative techniques. A good initial value for U is obtained by setting wait time w=0 in eq. (1), that is, setting U=1/(1+mt) which incidently corresponds to the maximum possible processor utilization.

The delta network: A delta network is an n stage network constructed from axb crossbar switches with a resulting size of $a^n x b^n$. Thus in our model, it is required that $N = a^n$ and $M = b^n$. For a more complete description see [PAT79]. Functionally, delta network is an interconnection network which allows any of the N cache modules to communicate with any one of the M main memory modules. However, unlike in a crossbar, two requests may collide in the delta network even if the requests were to two different memory modules. The average number of busy modules is computed recursively using the result of the axb crossbar as follows.

$$NmtU - Mm_n = 0 \qquad (8)$$

where, $m_{i+1} = 1 - (1 - m_i/b)^a \quad 0 <= i < n$

and $m_0 = 1 - U$

4. Discussion of the Results

In this section we present several results obtained using the above approximate analysis. The CPU utilization from approximate analysis was compared with the utilization obtained in simulation for a wide range of parameters. The comparison showed that our analysis overestimates the processor utilization in most cases. However, the error was less than 2% for mt<1 and less than 7% for mt<32. Thus the approximate analysis of the multiprocessor cache organization is quite accurate in the region where mt<1. As we shall see in the following discussion, it is this region which is of practical interest.

Consider Fig. 3 which is a graph of processor utilization over a broad range of parameters. The utilization plotted may be interpreted as simulation results or analytical results, since the differences are so small that they are not visible on the graph with the scale used. Since in the analysis, the processor utilization is a function of mt, the parameters m and t are not separated in this and other graphs. The graph shows three different systems, one is N=64, M=64 using 64x64 crossbar, second is N=64, M=64 using $2^6 x 2^6$ delta network and the third is the single processor system N=M=1. Since the wait time is zero in the case of M=M=1 system, the processor utilization from eq. (1) is 1/(1+mt), which serves as the upper bound on the processor utilization. It is clear from the graph that for mt>1 the processor utilization is less than 50%.

Therefore in a practical system one must have the product mt much smaller than 1 for an acceptable level of performance. Therefore the region of interest is mt<1. As pointed out earlier, it is in the region of interest that our approximate analysis is most accurate. Figures 4 and 5 show the processor utilization in the region of interest. Figure 4 shows a graph for a 32x32 crossbar network and a graph for 2^5x2^5 delta network. Figure 5 shows the processor utilization as a function of the network size NxN using a crossbar. Both figures are obtained from the analytical model of the previous section. Other measures of performance may also be evaluated from the analysis. For example the average traffic between the cache and the main and the average wait time of a request.

From a designer's perspective the above analysis shows that for optimum performance one must choose mt as small as possible. Recall that m is not the miss-ratio itself but it is directly proportional to the miss ratio and t is some function of the block size. A typical curve of miss ratio vs. block size for a fixed cache size might look like Fig. 6. From this, one can compute the graph of mt as a function of block size, where m is the probability that the CPU cycle is a memory reference and that it is a miss, and t is the block transfer time. From the graph, one can choose the optimum block size corresponding to the minimum mt. For a given cache size, the optimum block size for the maximum throughput, may or may not correspond to the minimum miss ratio of Fig. 6. To reduce mt below this value, one can either decrease the miss ratio by choosing a larger cache or reduce the block transfer time by using a faster main memory. Another alternative is to change the simple cache organization so that some of the block transfer time can be overlapped with the processor execution. This alternative is discussed in the next section.

5. Extensions of the Simple Cache Model

Three most common extensions of a simple cache organization are: (1) Buffered write back, (2) Store-through, and (3) Load-through. All of these achieve the same objective, namely, overlap of the CPU execution with a block transfer. Each extension is described below.

Buffered write back: On a cache miss, the block to be replaced, i.e., written back, is first stored in a high speed buffer. The desired block is then read into the cache module. Following this the buffer is written back to the main memory module. The writing back of the buffer is overlapped with the CPU execution.

Store-through (write-through): In this cache organization, every write command from the CPU results in the word being stored in the main memory, regardless of whether the corresponding block is present in the cache or not. If the block is present in cache then the word will also be written in the cache. As a consequence, on a read miss, the desired block is loaded in the cache and the block being replaced is not required to be written back. Here also the writing of each word in to the main memory is overlapped with the CPU execution.

Load-through: On a cache-miss for a read reference, the desired word is directly loaded into a CPU register from the main memory, after which the block containing that word is read into the cache module. This strategy tries to overlap the CPU execution with a block read. Load-through can be combined with either of the two previous strategies of write-back and write-through.

All of these organizations can be analyzed with the techniques presented in section 3. In each case the approximation used is to treat the memory traffic as consisting of independent single cycle requests. Furthermore, these requests are assumed to be uniformly distributed over all memory modules. Once the unit request rate is expressed in terms of known and unknown parameters, it can be substituted in eq. (6) or (8) to obtain the average number of busy modules, which is also expressed another way (similar to eqs. 2 and 3) to give a full set of equations for the solution of the CPU utilization.

6. Concluding Remarks

In this paper we have presented an approximate analytical model for multiprocessors with private cache memories. The accuracy of the model is remarkably good considering the complexity of the problem. In the region of practical interest the error of the analytical model is less than 2%. The same model is useful in computing several different measures of performance, such as processor utilization, average wait time of a request and memory traffic.

The central idea introduced in this paper is that of breaking up a request for a block transfer into several unit requests as well as treating waiting requests as several unit requests for the purpose of the analysis. This idea makes the analysis of more complex cache organizations like write-back, write-through and load-through as easy as the simple cache organization. As a side benefit, we now have a way to evaluate the bandwidths of crossbar and delta networks under asynchronous block transfer modes.

REFERENCES

[KAP73] K.R. Kaplan and R.O. Winder, "Cache based Computer Systems," Computer, pp. 30-36, March 1973.

[PAT79] J.H. Patel, "Processor-memory interconnection for multiprocessors," Proc. 6th Annual Symp. on Computer Architecture, pp. 168-177, 1979.

[PEU77] B.L. Peuto and L.J. Shustek, "An instruction timing model of CPU performance," Proc. 4th Symposium on Computer Architecture, pp. 165-178, 1977.

[STR76] W.D. Strecker, "Cache Memories for PDP-11 Family Computers," Proc. 3rd Symp. on Computer Architecture, pp. 155-158, Jan. 1976.

[YEH81] C-C. Yeh, "Shared Cache Organization for Multiple-Stream Computer Systems", Tech. Report No. R-904, Coordinated Science Lab, Univ. of Illinois, Urbana, IL, Jan. 1981.

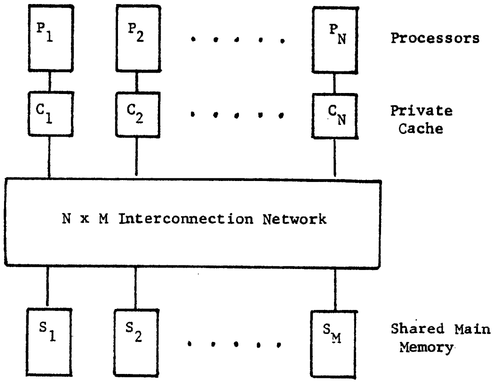


Fig. 1. Multiprocessor with Private Cache

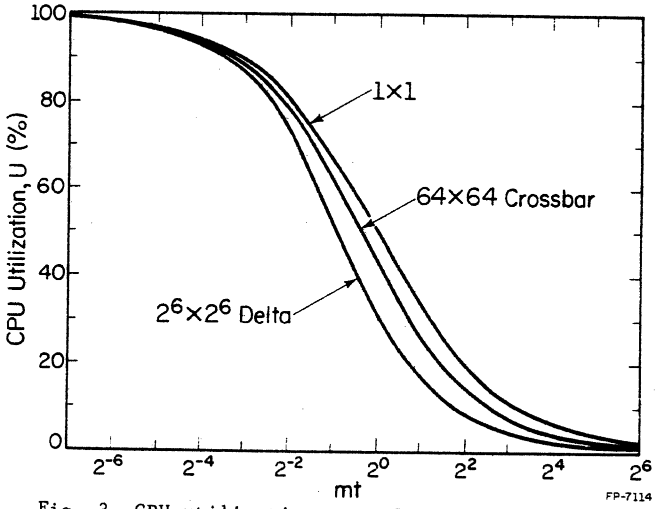


Fig. 3. CPU utilization as a function of mt

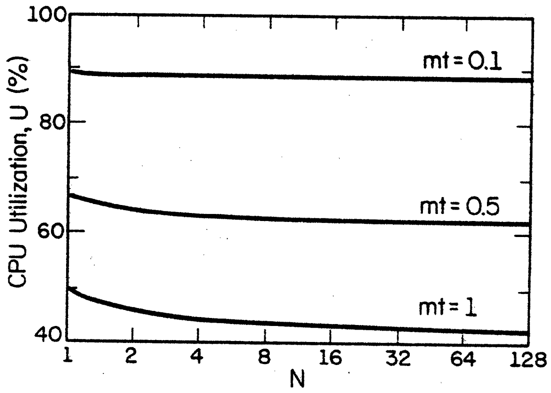
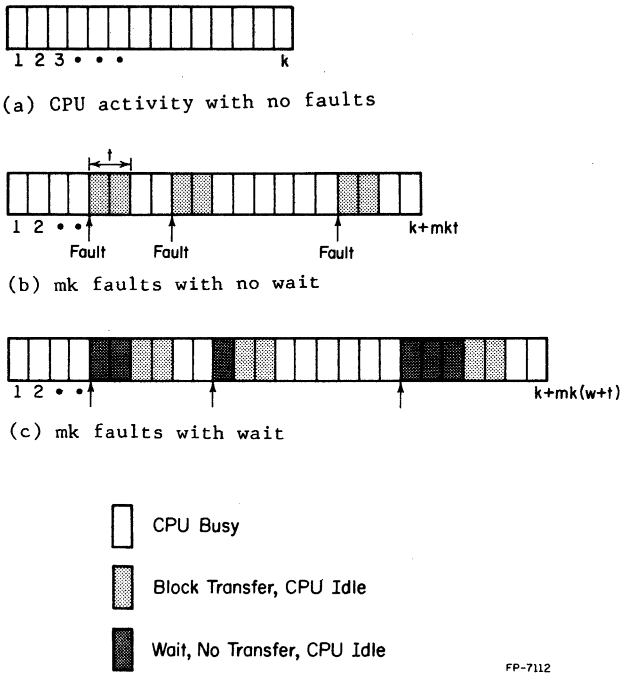


Fig. 5. CPU utilization vs. crossbar size



(a) CPU activity with no faults

(b) mk faults with no wait

(c) mk faults with wait

Fig. 2. CPU activity with faults and wait

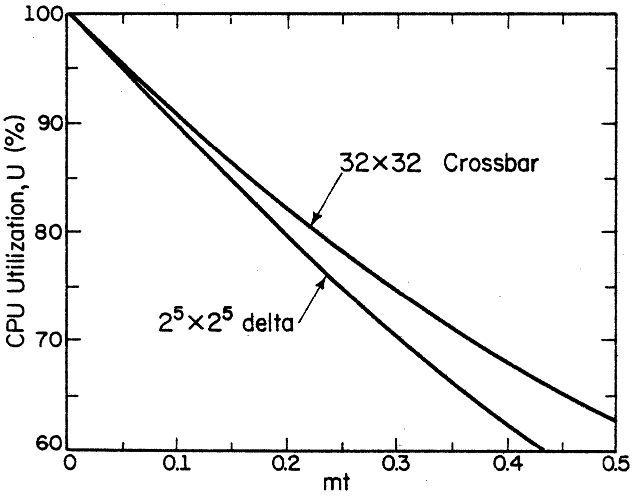


Fig. 4. CPU utilization as a function of mt

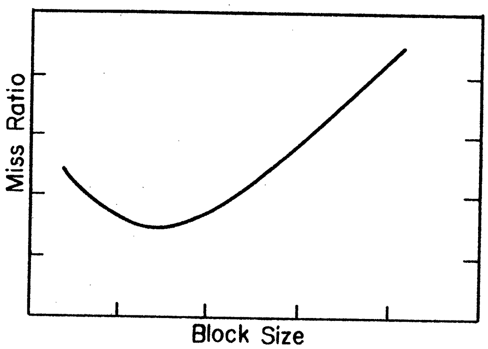


Fig. 6. Miss ratio vs. block size

AN ANALYSIS OF A NEW MEMORY SYSTEM FOR CONFLICT-FREE ACCESS

Tzu Yun-Kuei, Yang Shao-Tung, Yue Chang-Hai
Department of Computer Science
Changsha Institute of Technology
Hunan, China

Abstract--In this paper, the features of a new memory system are described. The process of conflict-free data array access is analyzed for interleaved memory system. The factors affecting the memory bandwidth and access speed are discussed, and the prime number of memory modules is determined for the AP-601 computer. The formulas for address transformation are derived and the corresponding networks are designed.

Introduction [1][4][5]

In interleaved memory systems, it is convenient to make m, the number of modules, a power of 2, say $m= 2^p$. Then the least significant p bits of every(binary) address immediately identify the module to which the address belongs. But the access conflicts, in this typical system, are very serious under some conditions. In order to generate conflict-free data array access, we make m, the number of modules, a prime number 31/17 for the AP-601 computer.

AP-601 is a high speed vectorized computer with ten multifunction processors of pipeline structure operating at 20 megacycles per second. The interconnection network of AP-601 is a double buss structure. The memory access time is 400 ns. The memory control unit consists of two pipelines for access, so that each cycle (50 ns) may read or write two words(2*72 bits) from or to the main memory system. Thus the highest frequency of conflict-free data array access is 40 MC/S. Each data array access instruction may read or write 128 words. A simple analysis on the memory bandwidth shows that the access speed of 17 modules (Mode 17) is 1.5 times that of 16 modules and that of 31 modules (Mode 31) is 1.4 times that of 32 modules. The hardware technique used to generate the conflict-free access is an effective address transformation unit to match the prime number of memory modules.

Determination of the Number of Modules for the AP-601 Computer

In principle, the number of modules is so determined as to obtain the maximum data array access speed required by the actual computer system.

For the interleaved main memory of m modules, the address distribution is shown in Table 1.

Table 1

Am \ Nm	0	1	2	...	m-1
0	0	1	2	...	m-1
1	m	m+1	m+2	...	m+m-1
2	2m	2m+1	2m+2	...	2m+m-1
•	•	•	•		•
•	•	•	•		•
•	•	•	•		•
a_i	$a_i m$	$a_i m+1$	$a_i m+2$	...	$a_i m+m-1$
•	•	•	•		•
•	•	•	•		•
•	•	•	•		•

Where Nm is module's number,
Am is the address in the module,
and the absolute physical address is

$$A = m * A_m + N_m \qquad (1)$$

The address sequence of a data array access instruction is

$$a_0, a_0+d, a_0+2d, \ldots, a_0+id, \ldots$$

Where a_0 is the first address, and d is the index distance.

Suppose, due to index distance d, we can only have access to m' memory modules among m memory modules. Here m' is called the effective number of modules for the interleaved memory. Since in the address sequence, the difference between any two neighbor addresses is d, while in a memory module the difference between any two neighbor addresses is m, so that in the address sequence, the difference of the two consecutive addresses, which fall into the same memory module, satisfies the equation $m'd = \{m,d\}$,

$$\text{i.e.}\quad m' = \frac{\{m,d\}}{d} = \frac{md}{(m,d)d} = \frac{m}{(m,d)} \qquad (2)$$

0190-3918/81/0000/0318$00.75 © 1981 IEEE

where $\{m,d\}$ is the least common multiple of m and d, (m,d) is the maximum common factor of m and d. Since m' equals m, is the ideal case, thus from(2) we should have $(m,d)=1$. Evidently, it is impossible for all values of d. But,when m is equal to a prime number m_p,we would have the case

$$m' = \begin{cases} m_p & (d \neq km_p) \\ 1 & (d=km_p) \end{cases} \qquad (k=1,2,3,\ldots)$$

In order to generate conflict-free data array access for AP-601's double buss structure, we should have

$$m' \geqslant \frac{2T_m}{t_{cp}} \qquad (3)$$

where Tm is the cycle time of a memory module, and t_{cp} is the cycle time of an instruction. For the AP-601 computer, $T_m = 400$ ns, $t_{cp} = 50$ ns.

i.e. $\qquad m' \geqslant 16.$

Hence, we select the prime number 31/17 for the AP-601 computer.

The average data array access speed $\overline{S}_{m17}$, or the average memory bandwidth $\overline{b}_{m17}$ of Mode 17 is compared with that of Mode 16 as follows: [3]

let

$$K_{17} = \frac{\overline{S}_{m17}}{\overline{S}_{m16}} = \frac{\overline{b}_{m17}}{\overline{b}_{m16}}$$

since

$$\frac{\overline{b}_{m17}}{\overline{b}_{m16}} = \frac{\overline{m'}_{17}}{\overline{m'}_{16}}$$

where $\overline{m'}_{17}(\overline{m'}_{16})$ is the average effective number of memory modules of Mode 17 (Mode 16).

i.e.

$$K_{17} = \frac{\overline{m'}_{17}}{\overline{m'}_{16}} \approx 1.5 \quad .$$

It is calculated in Table 2. Similarly, we have $K_{31} \approx 1.4$(relative to Mode 32).

In the mean while, there are inevitably some random scalar data access instructions in between or just after a data array access. In Mode 17, the probability of immediate execution of such instruction is only 0.18 to 0.6. But it increases with the number of memory modules. So we select 31 memory modules (Mode 31) as the normal operating mode and Mode 17 as the auxiliary mode for the AP-601 computer.

Table 2

d	$m'_{16} = \frac{16}{(16,d)}$	$m'_{17} = \frac{17}{(17,d)}$
1	16	17
2	8	17
3	16	17
4	4	17
5	16	17
6	8	17
7	16	17
8	2	17
9	16	17
10	8	17
11	16	17
12	4	17
13	16	17
14	8	17
15	16	17
16	1	17
17	16	1
18	8	17
⋮	⋮	⋮
	$\overline{m'}_{16}=10.6825$	$\overline{m'}_{17}=16.0625$

The Mothod of Address Transformation

When $m_p \neq 2^p(p=0,1,2,\ldots)$, it is necessary to find the module's number (or address) Nm and the address in the memory module Am by means of address transformation. In the AP-601 computer, this is done in only two stages. The execution time of each stage should not exceed 27 ns for matching the pipeline processing speed.

Address Transformation for Mode31

1.The Formulas for Calculating N_{m31} and A_{m31}

In the AP-601 computer, the total capacity of the main memory is 1984k words for $m_p = 31$. So the absolute physical address A = 21 bits (binary), i.e.

$$A = a_0\, a_1 \,\ldots\ldots a_{20}$$

Let
$$\begin{aligned} A_1 &= a_0 \\ A_2 &= a_1\ a_2\ a_3\ a_4\ a_5 \\ A_3 &= a_6\ a_7\ a_8\ a_9\ a_{10} \\ A_4 &= a_{11}a_{12}a_{13}a_{14}a_{15} \\ A_5 &= a_{16}a_{17}a_{18}a_{19}a_{20} \end{aligned}$$

In order to find N_{m31} and A_{m31}, it is required to calculate

$$\begin{aligned} \widetilde{A} &= \frac{A_1A_2A_3A_4A_5}{31} \\ &= \frac{A_1A_2A_3A_4A_5}{2^5-1} \\ &= A_1A_2A_3A_4A_5(2^{-5}+2^{-10}+2^{-15}+\ldots) \\ &= A_1A_2A_3A_4+A_1A_2A_3+A_1A_2+A_1 \\ &\quad+\frac{A_1+A_2+A_3+A_4+A_5}{31} \end{aligned}$$

From (1) we have

$$A_{m31}= A_1A_2A_3A_4 +A_1A_2A_3 +A_1A_2 +A_1 +\left[\frac{A_1+A_2+A_3+A_4+A_5}{31}\right] \qquad (4)$$

where $\left[\frac{A_1+A_2+A_3+A_4+A_5}{31}\right]$ is the quotient of $\frac{A_1+A_2+A_3+A_4+A_5}{31}$;

$$N_{m31}=\left\{\frac{A_1+A_2+A_3+A_4+A_5}{31}\right\},$$

where $\left\{\frac{A_1+A_2+A_3+A_4+A_5}{31}\right\}$ is the remander of $\frac{A_1+A_2+A_3+A_4+A_5}{31}$.

2.The Practical Procedure for Calculating N_{m31}

The first stage is to calculate $S''=A_1+A_2+A_3+A_4+A_5=\alpha_1\alpha_2S_1''S_2''S_3''S_4''S_5''$.

The second stage performs two kinds of modifications for S" to obtain N_{m31} finally.

Firstly, substracting "$\alpha_1\alpha_2$" *31 from S", we get

$$S'=S_1''S_2''S_3''S_4''S_5''+000\alpha_1\alpha_2=\alpha S_1'S_2'S_3'S_4'S_5' \quad .$$

Because max($A_1+A_2+A_3+A_4+A_5$)= 1111011, S' has only four kinds of different states from which the corresponding result of N_{m31} is obtained as shown in Table 3.

Table 3

State	$S'=\alpha S_1'S_2'S_3'S_4'S_5'$	$N_{m31}=S_1S_2S_3S_4S_5$
1	$0\ S_1'S_2'S_3'S_4'S_5'$ $(S_1'\cdot S_2'\cdot S_3'\cdot S_4'\cdot S_5'\neq 1)$	$S_1'S_2'S_3'S_4'S_5'$
2	0 1 1 1 1 1	0 0 0 0 0
3	1 0 0 0 0 0	0 0 0 0 1
4	1 0 0 0 0 1	0 0 0 1 0

Hence, by logic (or table look-up) we finally obtain

$$N_{m31} = S_1\ S_2\ S_3\ S_4\ S_5 \quad .$$

From Table 3, the required logic formulas are formed as follows:

$$S_1= \overline{F}S_1' \ , \ S_2 = \overline{F}S_2' \ , \ S_3 = \overline{F}S_3'$$
$$S_4=F(S_4' \oplus S_5')+\overline{F}S_4' \ , \ S_5 = F \oplus S_5' \quad .$$

where $F =\alpha+ S_1'\cdot S_2'\cdot S_3'\cdot S_4'\cdot S_5'$

3.The Practical Procedure for Calculating A_{m31}

The first stage is to calculate "$A_1A_2A_3A_4+A_1A_2A_3+A_1A_2+A_1$" . To save integrated circuits, the calculation is divided into three sections, i.e.

$$\begin{array}{rcc:c:c} & A_1 & A_2 & A_3 & A_4 \\ & & A_1 & A_2 & A_3 \\ & & & A_1 & A_2 \\ + & & & & A_1 \\ \hline \end{array}$$

where $A_2+A_3=H_0H_1H_2H_3H_4H_5$ has been obtained from the process for calculating N_m.

The first section is $A_1A_2=a_0a_1a_2a_3a_4a_5$. The second section is to do

$$\begin{array}{rl} & H_0\ H_1\ H_2\ H_3\ H_4\ H_5 \\ + & a_0 \\ \hline & a_4^*\ a_5^*\ H_1\ H_2\ H_3\ H_4\ H_5 \end{array}$$

The third section is to do

$$\begin{array}{rr} & H_0\ H_1\ H_2\ H_3\ H_4\ H_5 \\ & a_0\ a_{11}a_{12}a_{13}a_{14}a_{15} \\ + & a_0 \\ \hline & a_9^*\ a_{10}^*a_{11}''a_{12}''a_{13}''a_{14}''a_{15}'' \end{array}$$

The second stage is to do the final calculation of A_{m31},

$$\begin{array}{r} a_0\ a_1\ a_2\ a_3\ a_4\ a_5\ H_1\ H_2\ H_3\ H_4\ H_5\ a_{11}''a_{12}''a_{13}''a_{14}''a_{15}'' \\ 0\ \ 0\ \ 0\ \ 0\ \ a_4^*\ a_5^*\ 0\ \ 0\ \ 0\ \ a_9^*\ a_{10}^*0\ \ 0\ \ 0\ \ \alpha_1\ \alpha_2 \\ +\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad F\ \ \\ \hline a_0'\ a_1'\ a_2'\ a_3'\ a_4'\ a_5'\ a_6'\ a_7'\ a_8'\ a_9'\ a_{10}'a_{11}'a_{12}'a_{13}'a_{14}'a_{15}' \end{array}$$

where "$\alpha_1\alpha_2$+F" is the quotient of $\dfrac{A_1\ +A_2\ +A_3\ +A_4\ +A_5}{31}$, which has been obtained from calculating N_{m31}.

Address Transformation for Mode 17

1.The Formulas for N_{m17} and A_{m17}

In this case, the total capacity of the main memory is 1088k words, so we have similarly

$$A = a_0\ a_1\ a_2\ \cdots\cdots\ a_{20} \quad .$$

Let

$$\begin{aligned} A_0 &= a_0 \\ A_1 &= a_1\ a_2\ a_3\ a_4 \\ A_2 &= a_5\ a_6\ a_7\ a_8 \\ A_3 &= a_9\ a_{10}a_{11}a_{12} \\ A_4 &= a_{13}\ a_{14}\ a_{15}\ a_{16} \\ A_5 &= a_{17}\ a_{18}\ a_{19}\ a_{20} \quad . \end{aligned}$$

The process of address transformation for Mode 17 is much more complicated than that for Mode 31. In order to use the same stages as those used for Mode31, we put $A_0=0$, and directly take $A_1A_2A_3A_4$ as A_{m17}.

Since $A_0=a_0=0$,(it is guaranteed by programming), we have

$$\tilde{A} = \frac{A_1\ A_2\ A_3\ A_4\ A_5}{17}$$

$$= \frac{A_1\ A_2\ A_3\ A_4\ A_5}{2^4\ +\ 1}$$

$$= A_1A_2A_3A_4A_5(2^{-4}-2^{-8}+2^{-12}+\ldots)$$

$$= A_1A_2A_3A_4-A_1A_2A_3+A_1A_2-A_1 + \frac{A_1-A_2+A_3-A_4+A_5}{17} \quad .$$

As before, N_{m17} is equal to the remainder of

$$\frac{A_1\ -A_2\ +A_3\ -A_4\ +A_5}{17} \quad ,$$

i.e. $$N_{m17}=\left\{\frac{A_1-A_2+A_3-A_4+A_5}{17}\right\}$$

$$=\left\{\frac{A_1+A_3+A_5+34-A_2-A_4}{17}\right\}$$

$$=\left\{\frac{A_1+A_3+A_5+(32-A_2-A_4)+2}{17}\right\}$$

$$=\left\{\frac{A_1+A_3+A_5+\overline{A_2+A}_4+3}{17}\right\} \quad ,$$

where $\overline{A_2+ A_4}$ is the reverse code of $(A_2 + A_4)$.

2.The Practical Procedure for Calculating N_{m17}

In the first stage we calculate

$$S'' = A_1+A_3+A_5+\overline{A_2+A_4}+3$$

$$=\alpha_1\alpha_2\alpha_3 S''_1\ S''_2\ S''_3\ S''_4 \quad .$$

In the second stage we produce two kinds of modifications for S". Firstly, substracting "$\alpha_1\alpha_2\alpha_3$"*17 from S", we get

$$S' = 0\ S''_1\ S''_2\ S''_3\ S''_4 + 11\,\overline{\alpha_1}\,\overline{\alpha_2}\,\overline{\alpha_3} + 1$$

$$=\alpha'\ S'_1\ S'_2\ S'_3\ S'_4 \quad .$$

Because $\max(A_1-A_2+A_3-A_4+A_5+34)=1001111$, S' has only five kinds of different states from which the corresponding result of N_{m17} is obtained as shown in Table 4.

Hence, by logic (or table look-up), we obtain

$$N_{m17} =\alpha\, S_1\ S_2\ S_3\ S_4 \quad .$$

Table 4

State	$S'=\alpha' S'_1 S'_2 S'_3 S'_4$	$N_{m17}=\alpha S_1 S_2 S_3 S_4$
1	$0\ S'_1 S'_2 S'_3 S'_4$	$0\ S'_1 S'_2 S'_3 S'_4$
2	1 1 1 1 1	1 0 0 0 0
3	1 1 1 1 0	0 1 1 1 1
4	1 1 1 0 1	0 1 1 1 0
5	1 1 1 0 0	0 1 1 0 1

From Table 4, the required logic formulas are formed as follows

$$\alpha \ =\alpha' S'_3 S'_4 \ , \ S_1=\alpha'\,\overline{S'_3\ S'_4} +\overline{\alpha'}S'_1$$

$$S_2 \ =\alpha'\,\overline{S'_3\ S'_4} +\overline{\alpha'}S'_2$$

$$S_3 \ = \alpha'(S'_3 \oplus S'_4) +\overline{\alpha'}S'_3$$

$$S_4 \ = \alpha' \oplus S'_4$$

Thus the simplification of address transformation for Mode 17 is the compensation for 64kw reduction of memory capacity. In other words, we exchange 64kw memory capacity for the feature of conflict-free access.

The Memory Control Unit of AP-601 Computer [2][6][7][8]

As shown in Fig.1, the memory control unit of the AP-601 computer is used to organize the configuration of Mode 17/ Mode 31, arrange the accesses of multi-requests to be executed, and process the compression and restoration of vector access, indirect address, pipelines' chaining, etc.

It provides double busses for data array access, and a single bus for I/O and scalar access. In order to produce address transformations continuously, there are two address forming pipelines consisting of four segments each. The former two segments are used to form the absolute address and the latter two are used to produce address transformations (Fig.2, Fig.3). To match the address forming pipelines, there are another two pipelines for data written in, and two for data read out. The control signal generator is also a pipelined unit for synchronizing the operation of the whole memory control system.

In short, all the main memory system including the memory control unit is fully pipelined and equipped with self-testing and self-checking circuits.

The logic diagrams shown in Fig.2 and Fig.3 are separately designed from the formulas which have been derived for calculating N_m and A_m, in the above sections.

Conclusion

As mentioned above, the ratio of the average speed of Mode 17 to that of Mode 16 is $K_{17}\approx 1.5$. Similarly, we have $K_{31}\approx 1.4$. Since the quantity of integrated circuits used for address transformation may be neglected in comparison with that of the whole memory system, we get a 50% increase in access speed with 1/16 increase in memory hardware for Mode 17. And we get a 40% increase in access speed with 1/32 decrease in memory hardware and 64kw decrease in memory capacity.

Since the new memory system is fully pipelined, it results in very high data array access speed and is operated with very high efficiency.

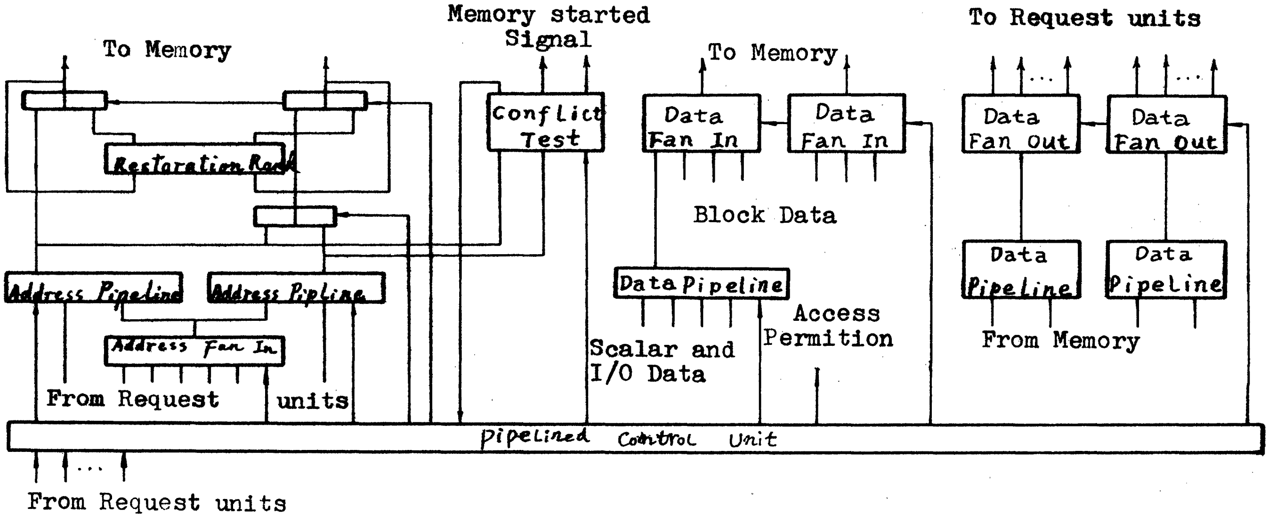


Fig.1, Block Diagram of Memory Control Unit

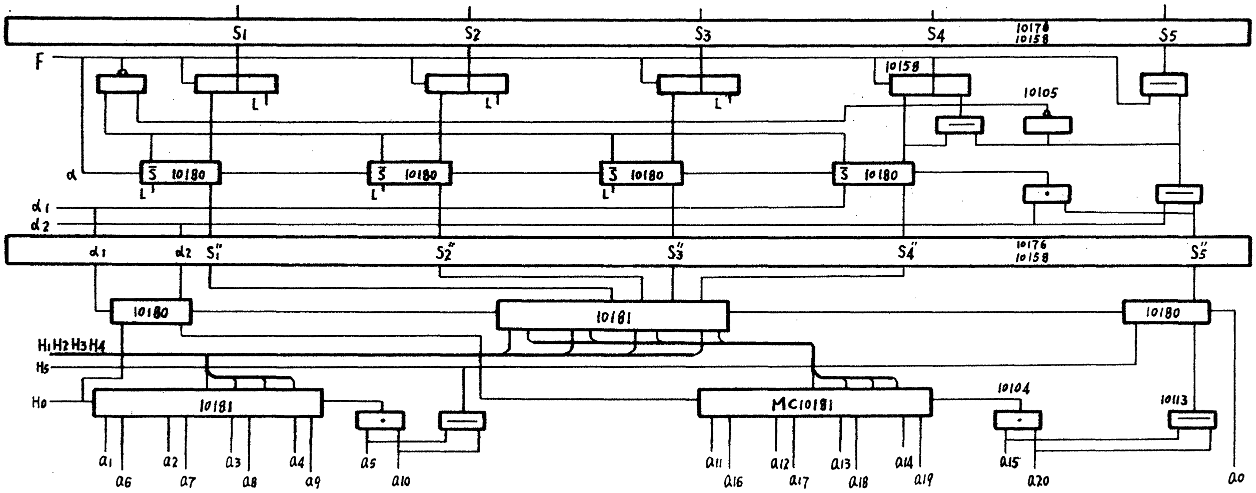


Fig.2, Logic Diagram for Calculating N_{m31}

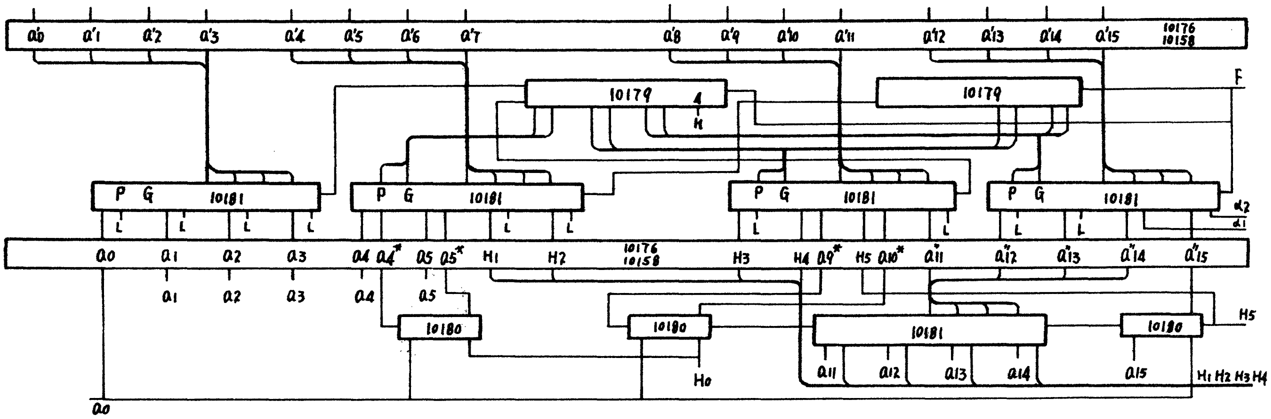

a'0 a'1 a'2 a'3 a'4 a'5 a'6 a'7 a'8 a'9 a'10 a'11 a'12 a'13 a'14 a'15
10176 10158
F
10179
10179
4
H
P G 10181
L
d2
d1
a0 a1 a2 a3 a4 a4* a5 a5* H1 H2 H3 H4 a9* H5 a10* a''11 a''12 a''13 a''14 a''15
a1 a2 a3 a4 a5
10180
10180
10181
10180
H0
a11 a12 a13 a14 a15
H5
H1 H2 H3 H4
a0

Fig.3, Logic Diagram for Calculating A_{m31}

References

[1] Burroughs Corporation,"Introduction to Burroughs Scientific processor," Document 1105327, Detroit Mich. July 1977.

[2] D.H.Lawrie,"Access and Alignment of Data in an Array Processor", IEEE Trans. on Computers, Vol.C-24,No.12 pp.1145-1155, Dec. 1975.

[3] D.Chang, D.J.Kuck, and D.H.Lawrie, "On the Effective Bandwidth of Parallel Memories," IEEE Trans. On Computers, Vol.C-26, No.5 pp.480-490, May 1977.

[4] Carl Jensen,"Taking Another Approach to Supercomputing," Datamation March. 1978.

[5] Cray Research Inc."Cray-1 Computer system Hardware Reference Manual," Minneapolis Minn. Nov.1977.

[6] P.M.Johnson," An Introduction to Vector Processing," Computer Design, Feb.1978 pp89-97.

[7] C.V.Ramamoorthy and H.F.Li," Pipeline architecture," ACM Computing Surveys, Mar.1977, pp.61-102.

[8] R.M.Russel," The Cray-1 Computer System," Communications of the ACM, Jan.1978 pp.63-72.

MODELING OF SHARED-RESOURCE SYSTEMS USING

THE CENTRAL-SERVER QUEUEING MODEL

N. C. Strole
IBM Corporation
Research Triangle Park, N. C.

P. N. Marinos
Dept. of Electrical Engineering
Duke University

SUMMARY

This paper introduces an efficient analytic method based on the central-server queueing model for establishing the initial design parameters for the simulation analysis of shared-resource computer architectures. The analytic model, originally designed for the optimization of multiprogramming systems, is shown to be applicable to the cost-throughput optimization of a shared-resource, multiprocessor system specifically designed for concurrent execution of tasks within a single job. Results from the analytic model are corroborated via simulation techniques with the Duke University Shared-Resource Simulator (DUSRS).

Background. One of the major goals in simulating shared-resource computer architectures is to determine an optimum resource configuration for handling a specified workload. An optimum configuration for a system may be that combination of resources that maximizes system throughput for a given workload, or possibly a system configuration that suffers the least degredation in throughput due to faults in system components (fault-tolerant system). However, many system designers are interested in maximizing the system throughput for a given system cost [4]. An analytic technique based on the central-server queueing model, first presented by Trivedi and Wagner [9], is shown in this paper to be applicable to the cost-throughput optimization of a shared-resource, multiprocessor system specifically designed for concurrent execution of tasks within a single job. This model is demonstrated to be an efficient method for establishing the initial design parameters required by the Duke University Shared-Resource Simulator (DUSRS) [8] for the simulation of shared-resource computer architectures.

An analytic model for maximizing system throughput in a multiprogramming environment by proper choice of device speeds, subject to a system cost constraint, was shown to be applicable to closed queueing networks and proven to yield a global optimum to the optimization problem [9]. The method of Lagrange multipliers was implemented in a Fortran program [5] to transform the problem into a system of nonlinear equations that were solved by employing an existing system subroutine package. Workload parameters, branching probabilities, and device costs were factored into the analysis over a range of system costs with varying degrees of multiprogramming, resulting in relative device utilizations, device speeds, and job throughput rates for each system budget and degree of multiprogramming. A thorough development of this analytic model can be found in [10].

Modeling Parallelism. The active resources in a multiprogrammed system, such as the bus and I/O channels, can be represented by a central-server queueing model having m+1 service centers (Figure 1). The ith service center consists of a single server and a queue for temporarily holding tasks waiting for service. The classical usage of the central-server queueing model assumes that the degree of multiprogramming, n, represents the number of concurrent jobs. Each job can be in only one of the servers or queues at any given time, and therefore, an individual job can be represented as a series of tasks, each requiring a particular system resource and a certain execution time for its completion. The tasks are serial in nature in that they pass from one server to the next during execution, thus not allowing task concurrency within a single job. All tasks must pass through the central server (CPU) before branching to one of the I/O servers with a fixed branching probability P_i , $i=0,1,,\ldots,m$.

To accurately model shared-resource systems, task concurrency within a single job must be represented. A precedence relation on the set of tasks, T, present in a job can be established by the partial ordering relation, R, such that tRt' whenever task t completes before t' can begin. Tasks t and t' are independent if they are non R-related. A directed path of tasks of length k may be defined as $(x_1 x_2)$ $(x_2 x_3)\ldots(x_{k-1}x_k)$ where x_i denotes a node (or task) at the ith level in the path. In such a directed path, for i and j such that $1\leq i<j\leq k$, x_j is a successor of x_i and x_i is a predecessor of x_j. A terminal task is one with no successor, while one with no predecessors is an initial task[3].

The maximum length, k_{max}, of any path within the job, represents the number of potential levels of parallelism within the job. Assuming a total of N tasks within a job, the degree of parallelism, D, for the job (i.e., the number of parallel tasks per level) is defined to be:

$$D = N/k_{max} \qquad (1)$$

Thus, D represents the mean number of concurrent active tasks within the job at any given time. For example, the hypothetical precedence graph shown in Figure 2 is composed of 20 tasks with a maximum path length of 5, resulting in a degree of parallelism, D=4.

The central-server model in Figure 1 can be adapted to model a single job having a fixed number of concurrent tasks, D, as shown in Figure 3. The CPU becomes the master instruction processor (MASTER); the m I/O devices become m distinct

0190-3918/81/0000/0325$00.75 © 1981 IEEE

classes of functional units, each represented by a single equivalent server (FUNIT); and D becomes the degree of multiprogramming. Whenever a task leaves the new job path, it is assumed that this represents the termination of the present job, which is immediately replaced by a new job having the same mean number of concurrent tasks, D.

The N tasks which comprise the job can be partitioned into g sets, Q_i, i=1,2,.....,g, where the tasks in each set Q_i require a type-i functional resource from the system pool of g resource classes. It is assumed that one additional task is present in the system that does not branch to one of the functional units upon completion at the MASTER to represent the termination of one job and the entry of a new job along the new program path. We define P_O as the new job path branching probability and it is given by:

$$P_o = 1/(N+1) \qquad (2)$$

The probability, Pi, that any given task will require a type-i device is given by:

$$P_i = |Q_i| \, /(N+1) \quad i=1,2,...g, \quad (3)$$

where $|Q_i|$ denotes the cardinality of set Q_i. Also, the sum of all P_i's must be such that:

$$\sum_{i=0}^{g} P_i = 1. \qquad (4)$$

The probability, P_i, that a task within the job will require service at the ith server is assumed to be identical to the branching probability, p_i, in the central-server model shown in Figure 1. Thus, the branching probabilities for the central-server model in Figure 3 are defined by equation (3).

It is now conjectured that one may equate the degree of parallelism, D, within a job to the degree of multiprogramming in the optimization problem, and thus use the analytic program to determine an optimum set of device speeds for a given budget. Branching probabilities are determined by the distribution of task types throughout the job as described above.

Several restrictions must be applied to the optimization problem to permit a closed-form solution; that is,

1. Task execution times are assumed to be exponentially distributed.
2. A task can occupy only one server at a time.
3. All tasks must pass through the MASTER upon leaving a FUNIT.
4. The number of concurrent tasks must remain constant.

The assumption that a task can occupy only one server or FUNIT at a time is not always true, since actual shared-resource systems permit tasks to simultaneously access multiple resources, such as memory, buses, and processing elements. Therefore, analytic techniques are limited in their ability to accurately model delays due to resource contention. The analytic model is thus seen as a possible tool for establishing some general system parameters for use in a more representative simulation model of a shared-resource system.

The Duke University Shared-Resource Simulator (DUSRS) has been developed as a research tool for the evaluation of shared-resource computer architectures and parallel program schemata [1]. The user can define a pool of system resource classes to be shared concurrently by one or more instruction stream processors, each with an optional multiprogramming capability. Workload data can be generated by a separate program facility that allows the user to implicitly define the hardware interconnection and control structure of the system, along with the process tasks to be executed [7]. The simulation output gives the resulting job execution times and functional resource utilizations.

The basic hypothesis that the degree of parallelism, D, within a single job can replace the degree of multiprogramming, n, in a central-server queueing model for the cost-throughput optimization of a shared-resource system, was validated by comparing the results from various runs of the analytic model with the results from comparable runs of the DUSRS simulation program [7]. Characteristic workloads, consisting of 10 jobs with degrees of parallelism, D, of 2, 4, and 8, were evaluated by both the analytic and the simulation models for a system consisting of one MASTER, one PROCESSOR, and one I/O CHANNEL. The 95% confidence interval for job execution time was computed (Table 1) as described in [6] and plotted with the analytic results (Table 2) as a function of system cost.

The graphical results for D=8 (Figure 4) show that the mean job execution times as determined by the analytic model fall within the 95% confidence interval established from the DUSRS simulation results. In addition, the device utilizations from the two methods are in very close agreement as shown in Table 3. Thus, the hypothesis that the degree of parallelism, D, within a job can replace the degree of multiprogramming, n, in a central-server queueing model for the cost-throughput optimization of a shared-resource system is shown to be valid, and the device speeds produced by the analytic model can be interpreted as the effective speed of the FUNIT classes, comprised of an unknown number of functional units. However, the simulation runs were implemented so as to accurately reflect the queueing model restrictions of allowing only one server (FUNIT) of each type. Thus, the true parallelism within the workloads that would have allowed multiple PROCESS and IOTASKS to occur simultaneously was not simulated.

The mean number of tasks in each queue in a FUNIT class can be estimated using a technique known as mean value analysis [2]. We can assume that this represents the average number of concurrent tasks within a job requiring access to each FUNIT class. A further simulation experiment was conducted in which the number of FUNITs in each class (except MASTER) was increased to this number to allow concurrent task execution, with no changes to the workload. The device speeds were reduced so that the effective speed of each device class was comparable to that of a faster single device. The 95% confidence interval established from this experiment (Table 4) also enclosed the analytic results as plotted in Figure 4.

Conclusion. In this paper, an analytic technique has been described for determining a set of cost-throughput optimized parameters for use in a more realistic and detailed simulation model of a shared-resource system for executing jobs containing potential parallelism among tasks. This technique was validated using independent simulation experiments for a class of systems that were assumed to be representable by a central-server closed queueing model.

REFERENCES

[1] N. Bell, A Shared Resource Schema for Parallel Computation, Ph.D Dissertation, Duke University, (1977).

[2] J. P. Buzen and P. J. Denning, "Measuring and Calculating Queue Length Distributions", Computer, (April 1980), pp. 33-44.

[3] E. G. Coffman and P. J. Denning, Operating Systems Theory, Prentice-Hall, (1973).

[4] D. Ferrari, Computer Systems Performance Evaluation, Academic Press, (1972).

[5] R. E. Kinicki, Queuing Models for Computer System Configuration Planning, Ph.D Dissertation, Duke University, (1978).

[6] H. Kobayashi, Modeling and Analysis, Addison-Wesley, (1978).

[7] N. C. Strole, Simulation Facility for Performance Evaluation of Shared-Resource Computer Architectures, Ph.D Dissertation, Duke University, (1980).

[8] N. C. Strole and P. N. Marinos, "A Shared-Resource Simulation Facility", Proceedings of 1979 Johns Hopkins Conference on Information Sciences and Systems, pp. 85-90.

[9] K. S. Trivedi and R. A. Wagner, "A Decision Model for Closed Queuing Networks", IEEE Trans. on Software Engr., (July 1979), pp. 328-332.

[10] K. S. Trivedi and R. E. Kinicki,"A Model for Computer Configuration Design", Computer, (April 1980), pp. 47-54.

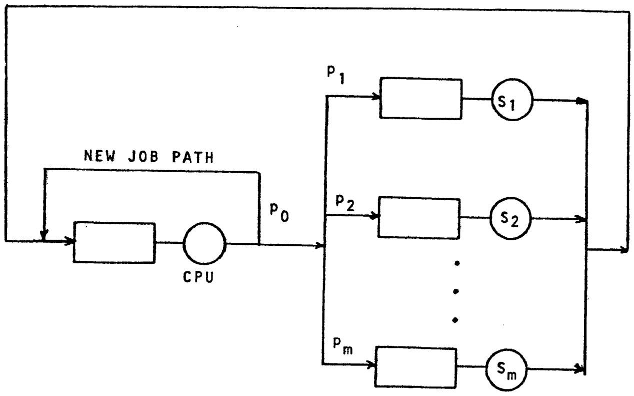


FIGURE 1 - CENTRAL-SERVER QUEUEING MODEL

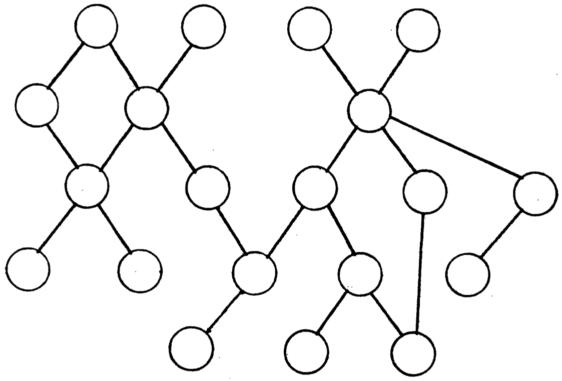


FIGURE 2 - PRECEDENCE GRAPH WITH DEGREE = 4

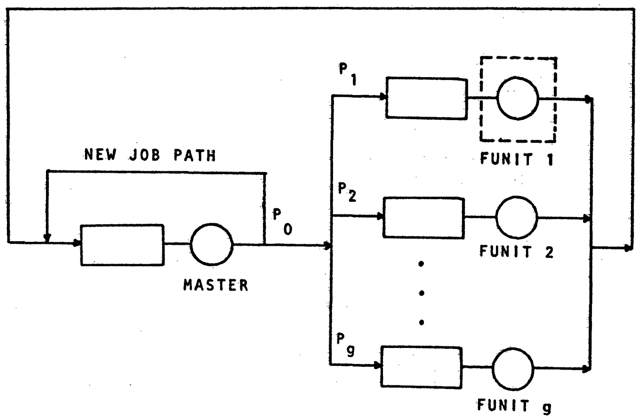


FIGURE 3
CENTRAL-SERVER MODEL FOR SHARED-RESOURCE SYSTEM

TABLE 1 - DUSRS 95% CONFIDENCE INTERVAL- DEGREE=8

JOB -RUNTIME (SEC)				
BUDGT→	$10K	$12K	$13K	$15K
MEAN S.D.	1.030 0.096	0.722 0.067	0.614 0.056	0.462 0.042
E LOW HIGH	±0.072 0.958 1.102	±0.051 0.671 0.773	±0.042 0.572 0.656	±0.032 0.430 0.494

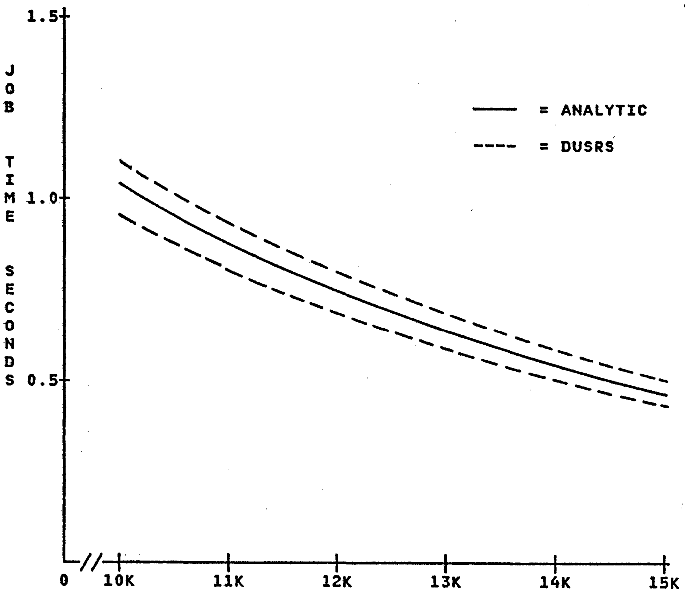


FIGURE 4 - 95% CONFIDENCE INTERVAL, DEGREE=8

TABLE 2 - ANALYTIC RESULTS

JOB RUNTIME (SEC)				
BUDGT→	$10K	$12K	$13K	$15K
MEAN	1.047	0.727	0.619	0.465

TABLE 3 - ANALYTIC VS DUSRS UTILIZATION RESULTS

DEG	DEVICE TYPE	ANALYTIC RELATIVE	DUSRS RELATIVE	ANALYTIC ACTUAL	DUSRS ACTUAL
2	MASTER PROCSR I/O	1.0 1.72 1.49	1.0 1.71 1.48	36.0% 61.9% 53.6%	36.0% 61.7% 53.3%
4	MASTER PROCSR I/O	1.0 1.45 1.31	1.0 1.47 1.30	52.3% 75.9% 68.5%	52.9% 77.5% 69.0%
8	MASTER PROCSR I/O	1.0 1.23 1.17	1.0 1.22 1.16	69.6% 85.6% 81.8%	70.7% 86.4% 82.4%

TABLE 4

DUSRS 95% CONFIDENCE INTERVAL WITH MULTIPLE DEVICES - DEGREE=8

JOB RUNTIME (SEC)				
BUDGT→	$10K	$12K	$13K	$15K
MEAN S.D.	1.063 0.114	0.737 0.080	0.631 0.064	0.472 0.050
E LOW HIGH	±0.070 0.993 1.133	±0.040 0.697 0.777	±0.039 0.592 0.670	±0.031 0.441 0.503

ABOVE RUNS MADE WITH 3 PROCESSORS AND 3 I/O CHANNELS

APPROXIMATE MODELS FOR MULTIPLE BUS MULTIPROCESSOR SYSTEMS

M. Ajmone Marsan
Istituto di Elettronica e Telecomunicazioni
Politecnico di Torino - Italy

M. Gerla
Computer Science Department
University of California, Los Angeles - USA

ABSTRACT Markovian models are developed for the performance analysis of multiprocessor systems intercommunicating via a set of busses. The performance index is the average number of active processors, called processing power. The computational complexity of the exact models increases very rapidly with system size, thus making the exact analysis impractical even for medium size systems. To overcome the complexity of computation, several approximate models are introduced. The approximate results are compared with the exact ones and found to be surprisingly accurate for a wide range of configurations.

1. INTRODUCTION

Early multiprocessor systems were developed using crossbar networks to connect processors and memories /1-5/. With the availability of inexpensive microprocessors, multiprocessor systems with a very large number of components are now becoming feasible and cost effective. For such systems a crossbar interconnection network may be intolerably expensive and in general it would provide a bandwidth much higher than needed. A more attractive alternative is represented by bus oriented interconnection networks. Single or multiple bus architectures can be used, according to the bandwidth required for the specific application. These interconnection networks are generally called "multiple-bus" or "highway deficient" /5/ networks. Some papers addressing the analysis of bus systems appeared very recently in the literature /5-7/.

This study considers multiple processor systems that exchange information through a common memory which consists of several modules. Processors and common memory modules are connected by a set of "global busses". Each global bus can connect any processor to any memory module. Every processor is also connected (and has exclusive access) to a private memory. We indicate a multiprocessor system with p processors, m common memories and b busses with the notation pxmxb. The block diagram of a 3x3x2 system is shown in fig. 1.

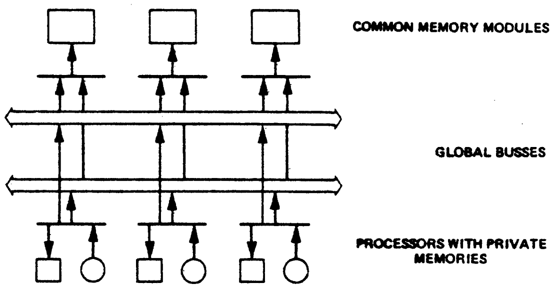


Fig. 1 - Block diagram of a 3x3x2 system.

The exchange of information is accomplished by fist writing the information in the appropriate common memory module and then reading it from the destination processor. Due to the sharing of both memory modules and busses, contention may arise, causing processors to queue for a resource which is currently in use. If the number of busses b is greater or equal to the smaller between the number of processors p and the number of memories m, i.e. $b \geqslant \min(m,p)$, then the contention is only caused by the sharing of memory modules.

Multiple processor systems for which the inequality holds are usually known as "crossbar" architectures. Multiple processor systems for which the inequality does not hold are usually called "highway deficient" systems or "multiple bus" architecture. For these systems we assume throughout this paper that $p \geqslant m \geqslant b$. The case $m > p$ can be analyzed using the same techniques described here; the models are generally simpler than those presented in this paper.

It is possible to construct a queueing network model for the analysis of both types of systems. The general case is shown in fig. 2. Processors join memory queues, and before proceeding to service (i.e. accessing memory) they must be granted a permit (bus). The permit is returned upon completion of service. The general model is thus a closed queueing network with p classes of customers and with passive resources /8,9/, which in this case represent the busses. In the case of crossbar architectures the presence of busses can be ignored, thus making the analysis substantially simpler than for multiple bus systems.

A processor can be in one of three different states:

This work was performed while M. Ajmone Marsan was visiting the UCLA Computer Science Department. The research was supported in part by ONR under contract N00014-79-C-0866 and in part by NATO.

0190-3918/81/0000/0329$00.75 © 1981 IEEE

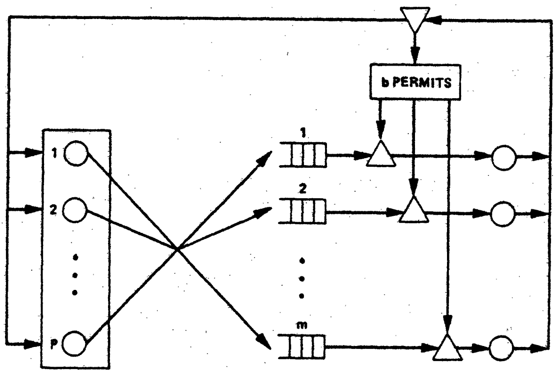


Fig. 2 - Closed queueing network model.

(1) The processor can execute in its private memory

(2) The processor can exchange data with other cooperating processors, by reading from, or writing into the common memory modules.

(3) The processor can be waiting to access a common memory module.

We say that a processor is ACTIVE when it is in state (1), and the goal of our analysis is to determine the average number of active processors, P, called processing power.

P is the performance index considered in the sequel. Other important performance measures are simply related to P.

The assumptions we make regarding the operation of the system are similar to those found in the literature on crossbar systems.

Each processing unit is active for some time while the CPU is executing a program that only requires accesses to its own private memory; the duration of these activity periods is an exponentially distributed random variable with the same parameter λ for all processors. At the end of an activity period processors generate access requests directed to a specific memory, chosen at random among the external common memory modules; each memory is requested with the same probability 1/m. If a bus is available and the requested memory is free, the processor accesses it for an exponentially distributed period with parameter μ, the same for all processors and memories. If either no bus is available or the requested memory is busy, the processor idles waiting for the necessary resources. At the end of an access the processor begins a new activity period; bus and memory are released and can be accessed by other processors. An arbitration mechanism for the assignment of the bus is assumed, that randomly chooses among the heads of the nonempty queues referencing free memory modules.

The model we consider is thus completely symmetric with respect to processors and memories. These symmetries are not necessary to obtain a Markovian model, but allow some reductions in the size of the resulting Markov Chain.

With the above assumptions we can construct a Markov chain to model the behavior of the system. Using the theory of "Lumpable" Markov chains /10/, we can reduce the number of states of the chain /11/.

The state definition for the exact lumped chain is:

$$(n_m, q_1, q_2, \dots q_m) \qquad (1)$$

where

n_m is the number of processors currently accessing a common memory

$q_1, \dots, q_b$ are the numbers of processors queueing for the memories currently accessed, arranged in decreasing order

$q_{b+1}, \dots, q_m$ are the numbers of processors queueing for a free memory, not accessible because no bus is available, arranged in decreasing order.

The general pxmxb case is not easy to handle, even after lumping is applied. We will therefore introduce in the next section some approximations which further reduce the size of the Markov chain and permit us to attack the most general case.

2. APPROXIMATE MODELS

The reason for the introduction of approximate Markovian models is that, for general multibus systems, the number of states increases very rapidly with system size. The explosive growth is due to the detailed information that the states must record about the queues inside the system. In particular for each state of the Markov chain the number of customers queued for all common memory modules must be recorded. That is, we not only need to know the number of the queued customers, but also must be concerned with all the possible ways of distributing these customers among the system queues. If we reduce the amount of information about the status of the queues we have no longer a first order Markov chain behavior in the evolution of the system through the state space. The approximate Markov models that we introduce in this section analyze the system behavior by assuming that the transitions between the states with reduced queueing information still satisfy the Markov property. The results that we will obtain in this way are approximate and must then be compared to the exact ones to test their accuracy.

In order to define a simplified model, one needs to specify:

a) the state definition, that is the amount of information used to describe the state of the Markov chain. As was mentioned before we will use reduced information about the queues in the system.

b) the method to calculate the transition rates for the simplified Markov model. As the behavior is approximated by the simplified Markov chain the transition rates must be evaluated according to some empirical rule, and several different rules can be envisioned.

Three different state definitions (named A, B and C) and two heuristic methods for the evaluation of the transition rates (named 1 and 2) were considered. The approximate models are named using the letter referring to the state description and the number referring to the evaluation of the transition rates.

> Model A1 - The state of the system is represented by the total number of processors waiting either for a busy memory or for a busy bus, n_q, and by the number of processors currently accessing a common memory module, n_m. We thus have a pair

$$(n_m, n_q) \tag{2}$$

The transition rates are evaluated by assuming that each queued processor requests, with uniform probability, any of the common memory modules currently not accessible (this approximation implies that a queued processor can randomly reselect a new memory when a memory or bus becomes unblocked).

Next we introduce a modification of model A1, by specifying a different method for the calculation of the transition rates:

> Model A2 - The state of the system is defined as in model A1). The transition rates are evaluated using an "averaging" technique.

In order to evaluate the new transition rates between two macrostates, we count the number of states that we merge into a macrostate, add all rates from the merged states to each neighboring macrostate, and define as transition rates the ratio between the sum of transition rates and the number of states merged.

We now consider another definition of system state (yet retaining the rate computation rule of model A2):

> Model B2 - The state of the system is represented by the following triplet: (1) the number of processors accessing a common memory module; (2) the total number of processors waiting either for a busy memory or for a busy bus; and (3) a flag which is set to zero when no processor is queued for a bus, and is set to one when one or more processors are queued for a bus in order to access a free common memory module.
> The transition rates are evaluated using the averaging technique described in the approximation A2.

Clearly, model B2 is a refinement of A2, since the state is improved by adding a binary information concerning the system queues.

All the preceding approximate models lack of one feature which is very desirable in all analytic models: namely, a closed form solution. We introduce here the simplest possible model which provides us with a closed form solution.

> Model C2 - The system state is simply the number of active processors: no account is kept of the state of internal queues. The transition rates are evaluated using the averaging technique.

We have reduced the system description to a birth and death Markov chain, whose solution is easily obtained.

3. RESULTS

We compare exact and approximate analytic results by considering a 6x4x2 system. Results are presented in Table 1. The first column gives the value of $\varrho = \lambda / \mu$, the second column shows the exact value of processing power as a function of ϱ , evaluated from the exact lumped chain. The other columns show the percentage error that affects the processing power value computed with each of the four approximations introduced. For this case the exact chain has 37 states, whereas the approximate chains have 12, 12, 16 and 7 states, respectively.

Approximations A1, A2 and B2 seem to yield upper bounds on the processing power, whereas C2 gives a lower bound. The upper bound can be intuitively explained for approximation A1, since the random redistributing of processors to memories tends to relieve congestion and thus to improve performance. The bounds seem to be rather tight, since percentage errors well below 10% were typically observed.

A 16-processor, 8-memory, 3-bus system was simulated, in order to test the accuracy of the approximate models for large system size. Results are shown in table 2. The approximate Markov chains of models A1 and C2, having 46 and 17 states respectively, were solved. The results show that the approximate models behave very well for a system of this size; indeed, the approximate results are so close to the simulation results that in most cases they fall within the 99.9% confidence interval. Moreover, since the system of linear equation associated with the approximate Markov chain can be easily solved with numerical methods, the approximate models require much less computer time than a simulation program.

REFERENCES

/1/ D.P. Bhandarkar "Analysys of memory interference in multiprocessors". IEEE Transactions on Computers, September 1975, pp. 897-908.

/2/ F. Baskett and A.J. Smith "Interference in multiprocessor computer systems with interleaved memory". Communications of the ACM, June 1976, pp. 327-334.

/3/ C.H. Hoogendoorn "A general model for memory interference in multiprocessors". IEEE Transactions on Computers, October 1977, pp. 998-1005.

/4/ A.S. Sethi and N. Deo "Interference in multiprocessor systems with localized memory access probabilities". IEEE Transactions on Computers, February 1979, pp. 157-173.

/5/ P.J. Willis "Derivation and comparison of multiprocessor contention measures". IEE Journal on Computers and Digital Techniques, August 1978, pp. 93-98.

/6/ S. Hoener and W. Roeder "Efficiency of a multiprocessor system with time-shared busses". EUROMICRO Newsletter, 1977, pp. 35-42

/7/ F. Fung and H. Torng "On the analysis of memory conflicts and bus contentions in a multiple-microprocessor system". IEEE Transactions on Computers, January 1979, pp. 28-37.

/8/ K.M. Chandy and C.H. Sauer "Approximate methods for analyzing queueing network models of computer systems". ACM Computing Surveys, September 1978, pp. 281-317.

/9/ T.W. Keller "Computer system models with passive resources". PhD Thesis, University of Texas at Austin, 1976.

/10/ J.G. Kemeni and J.L. Snell "Finite Markov chains". Van Nostrand, Princeton, 1960

/11/ M. Ajmone Marsan and M. Gerla "Markov models for multiple bus multiprocessor systems" UCLA Technical Report No. CSD 810304, Feb. 1981.

ϱ	exact	A1	A2	B2	C2
.01	5.94	.0	.0	.0	.0
.1	5.38	.07	.06	.01	-.39
.3	4.17	.89	.59	.15	-2.24
.5	3.19	1.82	.99	.30	-3.45
1.	1.86	2.52	.89	.28	-3.73
3.	.65	1.83	.27	.07	-2.75
5.	.39	1.56	.18	.08	-2.49
10.	.20	1.36	.13	.10	-2.29

Table 1 - Exact results and percentage errors for the 6x4x2 system.

ϱ	simulation	A1	C2
.01	15.98	15.98	15.98
.1	14.24	14.27	13.89
.333	8.59	8.73	8.20
.5	6.01	5.99	5.79
1.	2.99	2.99	2.97
3.	1.01	1.00	0.99
5.	.60	.60	.60
10.	.30	.30	.30

Table 2 - Simulation and approximate results for the 16x8x3 system.

THE ANALYSIS OF A DECENTRALIZED CONTROL ALGORITHM FOR JOB SCHEDULING UTILIZING BAYESIAN DECISION THEORY

John A. Stankovic
Department of Electrical and Computer Engineering
University of Massachusetts
Amherst, Massachusetts 01003

Abstract -- The principles of Bayesian decision theory can be applied as a systematic approach to complex decision making under conditions of imperfect knowledge. Decentralized control of individual functions of a distributed processing system seems to be an especially relevant area for application of Bayesian decision theory. This paper formulates the distributed processing job scheduling problem in Bayesian decision theory terms, solves the problem for a particular algorithm, and discusses the usefulness, limitations and open questions regarding this approach.

Introduction

Many of the potential advantages of a "cooperative" distributed processing system depend on the ability to develop effective decentralized algorithms for the executive control functions of the system, e.g., communication and scheduling. Proposed decentralized algorithms must be analyzed for performance (e.g., average delay of messages for the communication function or throughput for the scheduling function), for logical correctness, reliability, overhead costs, stability, fairness, extensibility, cost and difficulty of initialization, understandability, and how well the algorithms meet the specifications [1,2]. Even though many of these criteria are subjective, both the criteria themselves and their interactions must be addressed as best as possible by system designers. A further complication to the analysis is due to the conditions of imperfect knowledge that exist in a distributed system [1,3]. What is required is a methodology that addresses these criteria, their interactions, and the conditions of imperfect knowledge.

To date, mathematical programming, queueing theory and control theory have been applied to the analysis of decentralized control problems. Solutions using these disciplines are limited in some or all of the following ways: they are static optimization problems, require large amounts of computation, depend on accurate knowledge of the current state of the system, do not include details of the algorithms, and have limited or no potential for incorporating the other important issues of an algorithm's effectiveness listed above.

Our approach to the development and analysis of decentralized control algorithms is to use Bayesian decision theory [4,5,6,7,8,9]. This theory directly addresses decision making under uncertainty, and has the ability to incorporate many complex factors and their interactions via the utility function, e.g., reliability and performance. Furthermore, because of the systematic approach of Bayesian decision theory, designers of distributed systems can apply the principles of Bayesian decision theory as a methodology for algorithm design and evaluation as this paper illustrates.

This paper presents an example that illustrates the use of Bayesian decision theory in developing and analyzing a decentralized job scheduling algorithm. Several variations of the basic example are also presented. The examples are simple enough to understand yet involved enough to illustrate the practical use of decision theory to evaluate job scheduling decisions.

Decision Theory Under Uncertainty

There are seven essential steps in the formulation and solution of a decision problem under uncertainty. A particular formulation and solution is called Bayesian decision theory. In the following sections each of these steps is explained for a decentralized job scheduling algorithm. We then explain how to make use of the evaluation by incorporating the analytical results into the scheduling algorithm. The algorithm itself is described in a piecemeal fashion throughout the seven steps of the problem formulation.

Step One—Actions

The job scheduling function in a distributed processing system has two tasks: schedule jobs to run on hosts, and move jobs between hosts to balance the system load. The job scheduling function is implemented by n controllers (decision makers), one controller running on each host. For illustrative purposes assume n = 5. Each controller schedules on a FCFS basis when its host becomes available. There is no decision strategy needed for choosing the next job to run. Hence, this action of the scheduler need not enter into the decision problem. On the other hand, in order to balance the load each controller i may perform one of the following actions $A = \{a_j\}$ $j = 0,\ldots 5$:

a_0: move no jobs

a_1: move one job from queue i to host 1

a_2: move one job from queue i to host 2

a_3: move one job from queue i to host 3

a_4: move one job from queue i to host 4

a_5: move one job from queue i to host 5

Moving a job to oneself is considered as moving 0 jobs, hence a_0 is not really required. It is used in this example simply to emphasize the action of moving no jobs independently of the controller that is being considered. When activated, a controller continues to iterate until a) it decides to move 0 jobs, or b) it has decided to move

0190-3918/81/0000/0333$00.75 © 1981 IEEE

enough jobs to satisfy itself that it has done its share in balancing the load at this instance of time. Part (b) must take stability into account, i.e., all controllers should not dump all of their excess jobs into a lightly loaded host because that will most likely cause an unstable system. The actual movement of the jobs will be done asynchronously with the scheduler. The time required for the movement is dependent on the length of the jobs, traffic of the network and the distance to be moved. Periodically, state information is passed around the system in a manner similar to how ARPANET routing information is passed [10,11]. The scheduling algorithm itself is assumed to complete before the next state information update message arrives. This assumption is reasonable since the algorithm essentially just performs a table lookup. Note, that for more complicated situations each controller might have a different set of possible actions, or move clusters of jobs at once. Such actions are simply added to the set of possible actions.

Step Two—States of Nature

As the second step of a decision problem, the actual states of nature that could occur are specified. For scheduling, the states of nature are defined in terms of how busy a host is. In this example, this quantity is measured in terms of the number of jobs in the queue. More complicated quantification of the busy estimate can be based on many factors including, for example, the estimated requirements of the jobs in the queue [12]. However, the state information used to derive the busy factor is independent of the use of that busy factor in the decision theory context. The states of nature are defined as $\theta = \{\theta_0, \theta_1, \ldots \theta_{15}\}$ where:

θ_0: no hosts are least busy

θ_1: host 1 is least busy by 1-2 jobs

θ_2: host 2 is least busy by 1-2 jobs

θ_3: host 3 is least busy by 1-2 jobs

θ_4: host 4 is least busy by 1-2 jobs

θ_5: host 5 is least busy by 1-2 jobs

θ_6: host 1 is least busy by 3-5 jobs inclusive

θ_7: host 2 is least busy by 3-5 jobs inclusive

θ_8: host 3 is least busy by 3-5 jobs inclusive

θ_9: host 4 is least busy by 3-5 jobs inclusive

θ_{10}: host 5 is least busy by 3-5 jobs inclusive

θ_{11}: host 1 is leasy busy by >5 jobs

θ_{12}: host 2 is least busy by >5 jobs

θ_{13}: host 3 is least busy by >5 jobs

θ_{14}: host 4 is least busy by >5 jobs

θ_{15}: host 5 is least busy by >5 jobs

Obviously, the set θ can be specified in many ways. The parameters 1-2 jobs, 3-5 jobs and >5 jobs chosen in the above specification of θ would typically be tuneable system parameters and in practice the three classes would probably vary more than shown above. Furthermore, in a real system this definition of θ is probably not reasonable because it does not distinguish between the following cases. Suppose host 1 through host 4 have 0 jobs and host 5 has 8 jobs, versus host 1 through host 4 with 1000 jobs and host 5 with 1008 jobs. In the latter case job movement is probably not warranted. Another possibility is to define θ relative to the local host's busyness. We continue with the above definition of θ for ease of explanation.

Note, that due to the absence of uniqueness in both time and space [3] in a distributed system, θ_i cannot be known precisely. Hence, what is required is the probability of a given state occurring, $P(\theta_i)$, $i = 0,\ldots,15$ and this is best estimated by measurements or simulation. The very nature of a distributed system requires that probabilities be utilized in the analysis rather than assuming accurate state information. Fortunately, decision theory under uncertainty satisfies this criterion for analysis.

Step Three—Utility Function

The function $u(\theta_i,a_j)$ that assigns gains (or losses) to each action a_j $j=0,\ldots,5$ for each state of nature θ_i, $i=0,\ldots,15$ is called a utility function. A difficult and subjective aspect of decision theory under uncertainty is specifying the utility function. On the other hand, very complex interrelationships and situations can be subsumed within the specification of the utility function allowing, for example, the evaluation of both reliability and performance simultaneously. The usefulness of applying decision theory to decentralized control analysis revolves around the utility function. That is, if the derived utility functions are accurate in predicting practical decisions, then this theory will prove of tremendous help to the development and analysis of decentralized control algorithms. A hypothesis of this paper is that since there are methodological techniques[a] for deriving utility functions (even though the values are subjective), it is possible to derive an effective utility function for the scheduling function. At a minimum, the derivation of the utility function should provide a good methodology for addressing complex interactions and uncertainty existing in distributed systems.

The utility function $u(\theta_i,a_j)$ can be expressed in table format. Table 1 is the utility function derived for our job scheduling example. We now briefly describe how the values in the table were chosen.

It was decided to use a scale from 0 to 100 to quantify utility. If the state of nature is θ_0 (all hosts are equally busy) and the action is a_0 (no jobs are moved) then no utility is gained or lost from performing action a_0. This serves as a

[a] These techniques are not the subject of this paper but they are based on several simple axioms (see [9].)

reference point and is assigned the value 50. Hence, values between 0-49 indicate losses of utility, while values 51-100 indicate gains of utility. For states of nature θ_1-θ_4 (host i is least busy by 1-2 jobs) inclusive, the utility of moving a job is chosen to be 60; $u(\theta_1,a_1) = u(\theta_2,a_2) = u(\theta_3,a_3) = u(\theta_4,a_4) = 60$. This is less than the utility for states θ_6 - θ_9 (host (i-5) is least busy by 3-5 jobs) which is chosen to be 75 to indicate a non-linear increase in utility; $u(\theta_6,a_1) = u(\theta_7,a_2) = u(\theta_8,a_3) = u(\theta_9,a_4) = 75$. The utility for states θ_{11}-θ_{15} (host (i-10) is least busy by >5 jobs) is chosen to be maximum and assigned the value 100; $u(\theta_{11},a_1) = u(\theta_{12},a_2) = u(\theta_{13},a_3) = u(\theta_{14},a_4) = 100$. This again accounts for a non-linear increase of utility over the previous states.

Note entries $u(\theta_5,a_5) = 55$; $u(\theta_{10},a_5) = 60$; $u(\theta_{15},a_5) = 65$ in Table 1. These utilities are lower than the utilities for other hosts in these same situations. This might be true, for example, because host 5 has limited memory or is a slow processor. In fact, many of the complex factors that should be incorporated into the analysis of decentralized algorithms could be subsumed in the utility function. As an example, the $u(\theta_{14},a_4)$ in Table 1 was chosen to be 100. This was based on the idea that if host 4 were least busy by more than 5 jobs then sending a job there is exactly what should be done. However, $u(\theta_{14},a_4)$ could be tempered to account for factors such as the overhead of moving jobs, or the reliability of that particular host, or the probability that other hosts will also detect the same condition and send jobs to host i. The last factor implies that to maintain a stable system if a host becomes lightly loaded it should not be flooded with jobs.

The scheduling algorithm in this example, like the original ARPANET routing does not protect against ping-ponging (cycles). Hence, the utility may be further lowered to account for the probability of a cycle developing.

All other entries in Table 1 represent losses of utility. These entries indicate that moving jobs to the wrong host is counterproductive. For example, $u(\theta_1,a_2)$, the utility of moving a job to host 2 if the real state is θ_1 corresponds to a loss of utility and is chosen to be 30. Other losses are chosen to be consistent with this loss.

In summary, even though the development of the utility function is subjective, attempting to create such a function is a viable methodology that forces designers to consider the factors involved and their interrelationships. Furthermore, the sensitivity of the quantification of the utility function for a given problem can be determined and then used in the design process.

Step Four—Observations

Each controller maintains a table of state information. The values in this table estimate how busy each host of the network is. This table is periodically updated in a manner analogous to the ARPANET routing tables [10]. In this way, each controller maintains its own view of the state of nature of the network. The controller's view at a particular instant of time is called an observation. More formally, the conditional probability that host i observes z_1 when the true state of nature is θ_1 is written as $P(z_1|\theta_1)$. In our example, the set of possible observations is $Z = z_0, z_1, \ldots z_{15}$ where each z_k corresponds to θ_i for k = i except that the z_k's are observations. The values $P(z_k|\theta_i)$ assumed for this example are listed in table 2. In practice these probabilities can be determined by simulation or by measurement, and improved over time. For a "quasi-static" distributed system it is possible to obtain these probabilities dynamically. Again, such a probabilistic view of a system is, in fact, what a distributed algorithm must work with. Hence, the model being developed closely resembles the real system in this regard.

Step Five—Strategies

Next, the controller needs to formulate pure strategies which are decision rules that specify the action a_j, j = 0,...,5 that the controller takes in response to a particular observation z_k, k = 0,...,15. The set of possible pure strategies equals the number of actions raised to the power x where x is equal to the number of observations. In this example, the number of pure strategies equals 6^{16}. Fortunately, by using Bayesian decision theory an efficient computation procedure can be used to determine the best strategy without calculating all possible pure strategies. However, the set of probabilities $P(\theta_i)$ i=1,2,..., 15 is required. For this example, the assumed $P(\theta_i)$ is given in Table 3. Again, initially $P(\theta_i)$ would have to be estimated, but could be refined as measurements of the system were taken.

Step Six—Value of Strategies

In the decision theory problem formulation, step six is to compute the action probabilities for each pure strategy. However, Bayesian decision theory does not require the calculation of all the pure strategies nor the action probabilities. In lieu of these calculations, the Bayesian computational procedure takes three inputs. Inputs include the probabilities $P(Z|\theta_i)$ i = 0,..., 15 (Table 2), $P(\theta_i)$ i = 0,...,15 (Table 3) and the utility function $u(\theta_i,a_j)$ i= 0,...15 and j = 0,...,5 (Table 1). Then by a series of simple multiplications a maximizing action for each observed state of nature is produced. (See [9] for a description of the computational procedure.

It is also easy to calculate the expected value of each action and the weighted expected utility of the Bayesian strategy. This last numerical quantity can be used as a rough comparison of different scheduling algorithms. Precise comparisons are not meaningful unless the utility functions of each algorithm are consistent with each other.

Step Seven—Choice Criterion

Simply, the Bayesian strategy maximizes the expected average utility. That is, in our example the Bayesian strategy is the set of actions that maximize $\sum_{i=0}^{15} P(\theta_i)P(Z|\theta_i)\ u(\theta_i,A)$. Other choice criteria are possible but are not discussed in this paper. This then completes the formulation and means for a solution of a decision problem.

Utilization of Results

The designer first formulates the problem and algorithm in Bayesian decision theory terms. This is a subjective but methodological approach. A sensitivity analysis is then performed on the utility function to either enhance the confidence in the quantification of utility or identify those entries in the utility table that are highly sensitive. Careful treatment of these entries is then required. The next step for the designer is completely objective and consists of a simple computational procedure. The result is a set of maximizing actions for each decision maker (Table 4). In general, the maximizing actions for each decision maker may be different due to different utility functions and/or conditional probabilities.

The initial maximizing actions for each decision maker are then stored locally as part of the scheduling algorithm. When a job scheduling decision is to be made, a simple table look up is performed based on the current observation about the state of the network. This table look up identifies the proper decision for this scheduler. An important part of this process is the fact that this approach requires a low execution time overhead for the scheduler. Then special monitor nodes of the network act to dynamically adjust the probability distributions and maximizing actions by gathering statistics, recalculating maximizing actions, and downline loading these maximizing actions to the scheduling entities. This should be a reasonable heuristic if the system is quasi-static. Simulation studies are planned.

Bayesian Decision Theory Calculations

A PASCAL program was written for the Bayesian decision theory computational procedure and maximizing actions for the scheduling problem described above were calculated. The result is shown in Table 4 and is labelled Run 1. The input probabilities $P(\theta_i)$ for this run are reproduced in Table 4 for convenience. The results include the action a_j for each observation Z that maximizes expected average utility. Additional solutions were calculated for different probability functions $P(\theta_i)$. These results are labeled Run 2, 3 and 4 and are presented in Tables 5, 6, 7 respectively. Using the original probability function $P(\theta_i)$ of Run 1, Runs 5 and 6 were made by varying the Utility Function. In Run 5 only one row of the utility table (table 1) was changed. In Run 6, two rows were altered. These results are reported in Tables 8 and 9 respectively.

The primary intent of these calculations is to show how non-obvious maximizing actions result even for this simple example. By obvious results is meant that if a host i is observed as least busy then the proper action is to send jobs to it. If however, the probabilities and utilities interact in a manner as to make some other action optimal then this is a non-obvious result. If more complicating factors are added into the problem, all of the maximizing actions may be non-obvious. The maximizing actions in Tables 4-9 inclusive that are non-obvious are marked with an asterisk. Several representative non-obvious results from Tables 4-9 are now discussed.

For example, in Run 2 regardless of the observation z_0 - z_6 inclusive jobs are sent to host 1 to maximize expected utility. This bias occurs primarily because of the high probability that the true state of nature is θ_1, and the low utility gain of moving jobs when there exists only a difference of 1-2 jobs.

The most interesting observation about Run 3 is that for observation z_{10} (host 5 least busy by 3-5 jobs) the maximizing action is a_0 (move no jobs). This result is a combination of the high probability (.3) of θ_0 (all hosts equally busy) being the true state of nature and the low utility for host 5 because it is the slowest processor in our example. Yet, when the observation is z_{15} (host 5 least busy by >5 jobs) the maximizing action is a_5 (move jobs to host 5). This implies that the utility of the movement of jobs dominates the low probability of this state existing.

In Run 5 the $u(\theta_1,a_2)$ is increased from 30 to 50 over Run 1. This may occur for many different reasons, e.g., host 2 might be a very fast processor or be extremely reliable. This increase is enough to cause maximizing actions for observations z_0 and z_5 to be a_2. Yet, the $u(\theta_1,a_3)$ increasing from 30-40 and $u(\theta_1,a_5)$ decreasing from 30-20 are not enough of a utility function change to result in any non-obvious maximizing actions.

Similar justifications (post analysis) can be made for all the starred entries in Tables 4-9. However, before one does the Bayesian decision theory computations it is not easy to come up with the same results.

Conclusions

There is a definite need for an effective technique to develop, analyze and compare decentralized algorithms for "cooperative" distributed systems. Such systems are quite complex with many interacting forces, operate in a "noisy" environment, are inherently probabilistic, and often operate under strict time requirements. Most analysis techniques do not treat problems of this nature. In this paper it was shown how to apply Bayesian decision theory as a methodology to deal with this problem.

However, there are three main issues that must be resolved before this technique proves completely useable in practice. The first is concerned with the ability to develop an effective utility function. Our hypothesis was that even though the utility function was subjective it has the potential for subsuming some of the complicated interacting forces in an accurate way. It is a systematic method for dealing with complex interactions and uncertainty. A sensitivity analysis can be performed to enhance one's confidence in the utility quantification. We claimed that at a minimum this approach was more methodological and worthwhile than current techniques.

The second important issue concerns the dynamics of a distributed system. It is simple to claim that new sets of maximizing actions can be calculated when the statistics of the network change. Yet, will the number of significant changes in the statistics be small enough to be feasible? How often will a switch to a new set of maximizing actions be needed? Will the new period be long enough to make the switch worthwhile? Answers to these questions seem possible for what might be called "quasi-static" distributed processing systems.

The third issue is whether the two apriori probability functions $P(\theta_i)$ and $P(\theta_i|Z)$ can be known in practice. Initially, a best guess is made, or a standard statistical model is assumed. Then these probabilities can be revised in terms of the measured activities as experience with the system grows. It is also possible to maximize expected utility given only $P(\theta_i)$ and $u(\theta_i,a_i)$. This is sometimes called the no data problem because the decision is made without making a current observation. Therefore, if the decision maker does not take samples (observations) of the current state of the network into account then there is no need for the $P(\theta_i|Z)$ information. This technique might be appropriate in some systems for some algorithms.

References

[1] John A. Stankovic, "A Comprehensive Framework for Evaluating Decentralized Control," Proceedings 1980 International Conference on Parallel Processing, August 1980.

[2] G. E. LeLann, "An Analysis of Different Approaches to Distributed Computing," Proceedings the First International Conference on Distributed Computing Systems," Huntsville, Alabama, October 1-5, 1979.

[3] G. LeLann, "Distributed Systems—Towards a Formal Approach," Proceedings IFIP Congress, Toronto, North Holland Pub., August 1980, pp. 155-160.

[4] Herman Chernoff and L. E. Moses, Elementary Decision Theory, Wiley & Sons, NY, 1959.

[5] A. N. Halter and G. W. Dean, Decisions Under Uncertainty, South-Western Pub. Co., Chicago, IL, 1971.

[6] B. W. Lindgren, Elements of Decision Theory, The MacMillan Co., New York, 1971.

[7] Guillermo Owen, Game Theory, W. B. Saunders Co., Philadelphia, PA 1968.

[8] Howard Raiffa and Robert Schlaifer, Applied Statistical Decision Theory, Div. of Research, Graduate School of Business Adm., Harvard University, Cambridge, MA 1961.

[9] R. L. Winkler, Introduction to Bayesian Inference and Decision, Holt, Rinehard & Winston, Inc., New York, 1972.

[10] Leonard Kleinrock, Queueing Systems: Volume 2: Computer Applications, John Wiley & Sons, New York, 1976.

[11] John M. McQuillan and David C. Walden, "The ARPA Network Design Decisions," Computer Networks, The International Journal of Distributed Informatique, Vol. 1, No. 5, August 1977, pp. 243-289.

[12] L. Casey and N. Shelness, "A Domain Structure for Distributed Computer Systems," Proceedings of Sixth ACM Symposium on Operating System Principles, November 1977, pp. 101-108.

[13] Wesley W. Chu, L. J. Holloway, Min-Tsung Lau and Kemal Efe, "Task Allocation in Distributed Data Processing," IEEE Computer, Vol. 13, No. 11, November 1980, pp. 57-69.

[14] H. S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," IEEE Trans. on Software Engineering, Vol. SE-3, No. 1, January 1977, pp. 85-93.

[15] H. S. Stone and S. H. Bokhari, "Control of Distributed Processes," IEEE Computer, July 1978, pp. 97-106.

Table 1: Utility Function = $u(\theta_i, a_j)$

States of Nature θ	a_0	a_1	a_2	a_3	a_4	a_5
			A = Actions			
θ_0	50	30	30	30	30	15
θ_1	30	60	30	30	30	30
θ_2	30	30	60	30	30	30
θ_3	30	30	30	60	30	30
θ_4	30	30	30	30	60	30
θ_5	35	35	35	35	35	55
θ_6	15	75	15	15	15	15
θ_7	15	15	75	15	15	15
θ_8	15	15	15	75	15	15
θ_9	15	15	15	15	75	15
θ_{10}	20	20	20	20	20	60
θ_{11}	0	100	0	0	0	0
θ_{12}	0	0	100	0	0	0
θ_{13}	0	0	0	100	0	0
θ_{14}	0	0	0	0	100	0
θ_{15}	5	5	5	5	5	65

Table 2: $P(Z|\theta_i)$

States of Nature	z_0	z_1	z_2	z_3	z_4	z_5	z_6	z_7	z_8	z_9	z_{10}	z_{11}	z_{12}	z_{13}	z_{14}	z_{15}
θ_0	.0835	.0833	.0833	.0833	.0833	.0833	.07	.07	.07	.07	.07	.03	.03	.03	.03	.03
θ_1	.043	.14	.08	.08	.08	.08	.1	.0625	.0625	.0625	.0625	.075	.018	.018	.018	.018
θ_2	.043	.08	.04	.08	.08	.08	.0625	.1	.0625	.0625	.0625	.018	.075	.018	.018	.018
θ_3	.043	.08	.08	.14	.08	.08	.0625	.0625	.1	.0625	.0625	.018	.018	.075	.018	.018
θ_4	.043	.08	.08	.08	.14	.08	.0625	.0625	.0625	.1	.0625	.018	.018	.018	.075	.018
θ_5	.043	.08	.08	.08	.08	.14	.0625	.0625	.0625	.0625	.1	.018	.018	.018	.018	.018
θ_6	.01	.07	.07	.07	.07	.07	.24	.03	.03	.03	.03	.15	.0125	.0125	.0125	.0125
θ_7	.01	.15	.15	.07	.07	.07	.03	.24	.03	.03	.03	.0125	.15	.0125	.0125	.0125
θ_8	.01	.07	.07	.15	.07	.07	.03	.03	.24	.03	.03	.0125	.0125	.015	.0125	.0125
θ_9	.01	.07	.07	.07	.015	.07	.03	.03	.03	.24	.03	.0125	.0125	.0125	.15	.0125
θ_{10}	.01	.07	.07	.07	.07	.15	.03	.03	.03	.03	.24	.0125	.0125	.0125	.0125	.15
θ_{11}	.005	.06	.06	.06	.06	.06	.2	.03	.03	.03	.03	.24	.0156	.0156	.0156	.0156
θ_{12}	.005	.13	.13	.06	.06	.03	.2	.03	.03	.03	.0156	.24	.0156	.0156	.0156	.0156
θ_{13}	.005	.06	.06	.13	.06	.06	.03	.03	.2	.03	.03	.0156	.0156	.24	.0156	.0156
θ_{14}	.005	.06	.06	.06	.13	.06	.03	.03	.03	.2	.03	.0156	.0156	.0156	.24	.0156
θ_{15}	.005	.06	.06	.06	.06	.13	.03	.03	.03	.03	.2	.0156	.0156	.0156	.0156	.24

Table 3: P(θ_i)

θ_0	.03	θ_8	.07
θ_1	.082	θ_9	.08
θ_2	.082	θ_{10}	.06
θ_3	.082	θ_{11}	.044
θ_4	.096	θ_{12}	.044
θ_5	.072	θ_{13}	.044
θ_6	.07	θ_{14}	.052
θ_7	.07	θ_{15}	.052

Table 4: Run 1

		Results	
θ	P(θ_i)	Z	A (maximizing action)
θ_0	.03	z_0	a_4 *
θ_1	.082	z_1	a_1
θ_2	.082	z_2	a_2
θ_3	.082	z_3	a_3
θ_4	.096	z_4	a_4
θ_5	.072	z_5	a_4*
θ_6	.07	z_6	a_1
θ_7	.07	z_7	a_2
θ_8	.07	z_8	a_3
θ_9	.08	z_9	a_4
θ_{10}	.06	z_{10}	a_5
θ_{11}	.044	z_{11}	a_1
θ_{12}	.044	z_{12}	a_2
θ_{13}	.044	z_{13}	a_3
θ_{14}	.052	z_{14}	a_4
θ_{15}	.052	z_{15}	a_5

Table 5: Run 2

		Results	
θ	P(θ_i)	Z	A (maximizing action)
θ_0	.001	z_0	a_1 *
θ_1	.199	z_1	a_1
θ_2	.05	z_2	a_1 *
θ_3	.05	z_3	a_1 *
θ_4	.05	z_4	a_1 *
θ_5	.05	z_5	a_1 *
θ_6	.1	z_6	a_1
θ_7	.05	z_7	a_2
θ_8	.05	z_8	a_3
θ_9	.05	z_9	a_4
θ_{10}	.05	z_{10}	a_5
θ_{11}	.1	z_{11}	a_1
θ_{12}	.05	z_{12}	a_2
θ_{13}	.05	z_{13}	a_3
θ_{14}	.05	z_{14}	a_4
θ_{15}	.05	z_{15}	a_5

Table 6: Run 3

		Results	
θ	P(θ_i)	Z	A (maximizing action)
θ_0	.3	z_0	a_0
θ_1	.1	z_1	a_1
θ_2	.1	z_2	a_2
θ_3	.1	z_3	a_0 *
θ_4	.1	z_4	a_4
θ_5	.1	z_5	a_0 *
θ_6	.02	z_6	a_1
θ_7	.02	z_7	a_2
θ_8	.02	z_8	a_3
θ_9	.02	z_9	a_4
θ_{10}	.02	z_{10}	a_0 *
θ_{11}	.02	z_{11}	a_1
θ_{12}	.02	z_{12}	a_2
θ_{13}	.02	z_{13}	a_3
θ_{14}	.02	z_{14}	a_4
θ_{15}	.02	z_{15}	a_5

Table 7: Run 4

θ	$P(\theta_i)$	Results Z	A (maximizing action)
θ_0	.01	z_0	a_1 *
θ_1	.02	z_1	a_1
θ_2	.02	z_2	a_2
θ_3	.02	z_3	a_1 *
θ_4	.02	z_4	a_4
θ_5	.02	z_5	a_1 *
θ_6	.2	z_6	a_1
θ_7	.1	z_7	a_2
θ_8	.01	z_8	a_3
θ_9	.05	z_9	a_4
θ_{10}	.15	z_{10}	a_5 *
θ_{11}	.1	z_{11}	a_1
θ_{12}	.0525	z_{12}	a_2
θ_{13}	.055	z_{13}	a_3
θ_{14}	.075	z_{14}	a_4
θ_{15}	.0525	z_{15}	a_5

Table 8: Run 5

θ	$u(\theta_i,a_j)$					
	a_0	a_1	a_2	a_3	a_4	a_5
θ_1	30	60	50	40	30	20

Note: θ_1 row is the only change made to the utility function of Table 1.

Results		Results	
Z	A (maximizing action)	Z	A (maximizing action)
z_0	a_2 *	z_8	a_3
z_1	a_1	z_9	a_4
z_2	a_2	z_{10}	a_5
z_3	a_3	z_{11}	a_1
z_4	a_4	z_{12}	a_2
z_5	a_2 *	z_{13}	a_3
z_6	a_1	z_{14}	a_4
z_7	a_2	z_{15}	a_5

Table 9: Run 6

θ	$u(\theta_i,a_j)$					
	a_0	a_1	a_2	a_3	a_4	a_5
θ_{12}	40	10	100	0	10	10
θ_{13}	50	0	0	70	0	0

Note: θ_{12} and θ_{13} are the only changes made to the utility function of Table 1.

Results		Results	
Z	A (maximizing action)	Z	A (maximizing action)
z_0	a_4 *	z_8	a_3
z_1	a_1	z_9	a_4
z_2	a_2	z_{10}	a_5
z_3	a_3	z_{11}	a_1
z_4	a_4	z_{12}	a_2
z_5	a_4 *	z_{13}	a_3
z_6	a_1	z_{14}	a_4
z_7	a_2	z_{15}	a_5

COORDINATING LARGE NUMBERS OF PROCESSORS

Allan Gottlieb*, B.D. Lubachevsky, and Larry Rudolph
Courant Institute of Mathematical Sciences, NYU
New York NY 10012

(extended abstract)

ABSTRACT

In this paper we implement several basic operating system primitives by using a "replace-add" operation, which can supersede the standard "test and set", and which appears to be a universal primitive for efficiently coordinating large numbers of independently acting sequential processors. We also present a hardware implementation of replace-add that permits multiple replace-adds to be processed nearly as efficiently as loads and stores. Moreover, the crucial special case of concurrent replace-adds updating the same variable is handled particularly well: If every PE simultaneously addresses a replace-add at the same variable, all these requests are satisfied in the time required to process just one request.

1.0 INTRODUCTION

Very large scale parallel processing, made possible by the refinement of VLSI technology, is becoming a reality. Although current MIMD (multiple instruction streams - multiple data streams) configurations rarely include more than a few dozen processing elements (PEs), much larger configurations are being designed (Burroughs [3], CHoPP (see Sullivan et al. [23]), etc.) and configurations involving tens of thousands of PEs will soon be feasible.

Since in such configurations the relative cost of serial bottlenecks rises linearly with the number of PEs present, users of these future ultra-large scale parallel machines will be anxious to avoid the use of critical (and hence necessarily serial) code sections, even if these sections are short enough to be entirely acceptable in current practice.

In this report we implement several basic operating system primitives by using a "replace-add" operation, which can supersede the standard "test and set", and which appears to be a universal primitive for efficiently coordinating large numbers of independently acting sequential processors. We also present a hardware implementation of replace-add that permits multiple replace-adds to be processed nearly as efficiently as loads and stores. Moreover, the crucial special case of concurrent replace-adds updating the same variable is handled particularly well: If every PE simultaneously addresses a replace-add at the same variable, all these requests are satisfied in the time required to process just one request.

Critical sections, used to enforce mutual exclusion when multiprocessing a single PE, were introduced by Dijkstra [5] and later refined by Knuth [16] and Eisenberg and McGuire [10]. Later, Dijkstra [7] and Lamport [17] studied similar issues for parallel processing. Although this report also considers similar issues, we assume a somewhat different computational model. Various multiprocessor synchronization primitives, including those used below, have been compared by Lipton [19], Burns et al. [2], Henderson and Zalcstein [14], and Dolev [8].

This report is organized as follows. First, our "paracomputer" model of computation is explained and the replace-add operation is defined (section 2). We then use the replace-add operation to implement semaphores (section 3) and to solve the readers/writers problem without recourse to critical section code (section 5). A distributed queue management technique that also avoids the use of critical sections is derived and then enhanced to form the core of a distributed operating system scheduler (section 6). Finally, the replace-add hardware design is outlined (section 9).

2.0 COMPUTATIONAL MODEL

The replace-add operation, on which many of our considerations are based, was introduced in the 1967 studies of the Athene hypothetical parallel computer system (Draughon et al. [9]). Before describing this operation, a generalized test and set that appears to be an attractive primitive for coordinating concurrent processes, we first discuss our model of parallel computation.

2.1 The Machine - An ideal parallel processor, dubbed a "paracomputer" by Schwartz [21], consists of identical PEs sharing a common memory. The individual PEs may also have attached local memory, which we refer to as their "private" memories; the memory shared by and

This work was supported in part by the US DOE under Contract No. DE-AC02-76ER03077 and in part by the NSF under Grant No. NSF-MCS76-00116.

* On leave from York College, CUNY

0190-3918/81/0000/0341$00.75 © 1981 IEEE

common to all processors is called "public", and variables stored there are called "public variables". The PEs can simultaneously read any public cell in one cycle. Moreover, simultaneous writes (including the replace-add operation described below) are likewise effected in a single cycle and a memory cell to which such writes are directed will contain some one of the quantities written into it. This requirement on simultaneous memory updates illustrates the (paracomputer) serialization principle: The effect of simultaneous actions by the PEs is as if the actions occurred in some (unspecified) serial order. Note that simultaneous memory updates are not serialized; in fact they are accomplished in one cycle. The serialization principle speaks only of the effect of their action and not of their implementation. (Paracomputers must be regarded as idealized computational models since physical fan-in limitations prevent their realization.)

Our (realizable) approximation to a paracomputer is an MIMD parallel processor in which each PE can directly access its private memory and can access the public memory via a (multicycle) interconnection network. Since in this more realistic architecture a public memory access may require many PE cycles, we must carefully define the notion of simultaneous: Two actions r1 and r2 are simultaneous if r1 starts before r2 finishes and r2 starts before r1 finishes.

2.2 Replace-Add - The format of the replace-add operation, which forms the basis of much of our subsequent discussion, is RepAdd(V,e), where V is an integer variable and e is an integer expression. This indivisible operation yields the sum S=V+e as its value and replaces the contents of storage location V by this sum. Moreover, RepAdd must satisfy the serialization principle: Assume that V is a public variable (as it ordinarily will be) and many (perhaps very many) replace-add operations simultaneously address V. Then the effect is as if these operations occurred in some (unspecified) serial order, i.e. V receives the appropriate total increment and each operation yields the intermediate value of V corresponding to its position in this order*. The following example illustrates the semantics of replace-add: If V is a public variable, if PEi executes

```
            ANSi <-- RepAdd(V,ei)   ,
```

if PEj simultaneously executes

```
            ANSj <-- RepAdd(V,ej)   ,
```

and if V is not simultaneously updated by another PEk, then either

```
            ANSi <-- V+ei
            ANSj <-- V+ei+ej
```

or

```
            ANSi <-- V+ei+ej
            ANSj <-- V+ej
```

and, in either case, the value of V becomes V+ei+ej. The first possibility corresponds to the serialized order in which first PEi executes its replace-add and then PEj executes its replace-add; the second possibility corresponds to the opposite serialization. Suppose, to be still more specific, that V initially contained the value 10, and that ei=2 and ej=6. Then, after the simultaneous executions, V will contain 18 and either ANSi=12 and ANSj=18 or ANSi=18 and ANSj=16.

In section 9 we present a hardware design in which the replace-add operation requires essentially the same execution time as a load or store and in which simultaneous replace-adds updating the same variable are processed particularly effeciently.

3.0 SEMAPHORES

Having reviewed the basic replace-add operation, we proceed to describe its role in implementing a variety of higher-level programming operations. We first present a replace-add based implementation of Dijkstra's [5] P(S) and V(S) operations (thus illustrating that replace-add obviates one important need for test-and-set), and then generalize this implementation to PVchunk operations PC(S,e) (resp. VC(S,e)) where S is incremented (resp. decremented) by e (see [25]). Subsequent sections show that our implementation of PC and VC permits more parallelism than traditional implementations.

Recall that the P and V operations are used to protect critical code sections by enforcing the following "PV-property": If many processors concurrently execute*

```
Procedure PVTest
    Comment: Initially S=1.
        Cycle { P(S)
            critical section
            V(S) }
End Procedure
```

if the critical section does not modify S, and if no PE ceases execution, then at any time T at most one processor is executing its critical section and there exists a time t ≥ T when exactly one processor is executing a critical section.

3.1 Implementing PV - In this section we present a PV implementation that satisfies the

* These intermediate values result from executing prefixes of the serialized list of operations.

* We use "{" and "}" for the tokens "Begin" and "End" respectively. However, our indentation convention obviates the need for these tokens.

PV-property given above (see [12] for a proof of this claim). The P(S) operation first waits until the public variable S equals 1 and then executes RepAdd(S,-1). If the result is zero, the critical section may be entered. If the result is negative, some other processor has control of the section and so P(S) "covers its tracks" and then tries again. The V(S) implementation consists simply of a replace-add incrementing S by 1. The following code is an appropriate implementation of these important primitives. (As will be explained below, various subtleties are involved.)

```
Procedure P(S)
    Repeat
        If S-1 ≥ 0 Then
            If RepAdd(S,-1) ≥ 0 Then OK <-- true
            Else { RepAdd(S,1)
                   OK <-- False }
    Until OK
End Procedure

Procedure V(S)
    RepAdd(S,1)
End Procedure
```

To emphasize a subtle point inherent in our implementation of P, consider the following very similar, but actually incorrect, implementation.

```
Comment: Incorrect implementation of P
Procedure NaiveP(S)
    Repeat
        If RepAdd(S,-1) ≥ 0 Then OK <-- true
        Else { RepAdd(S,1)
               OK <-- false }
    Until OK
End Procedure
```

If one compares this simplified form with the correct original shown earlier, it may appear that we have merely removed a "redundant" test. However, the simplified code can in fact fail due to unacceptable race conditions. Suppose, for example, three PEs, A, B, and C, execute P(S) at the same time with S having its initial value of 1. If the serial order effected is equivalent to A executes first followed by B and C, then S is set to -2 and A enters the critical section. Suppose that A subsequently leaves the critical section, thus incrementing S to -1. The section should now be free to be entered by either B or C. The above code will allow this to occur as soon as S is incremented to +1 from its current value of -1. However, this may never happen, since the following endless scenario is possible: B increments S to 0 and then decrements S back to -1 before C executes its next instruction; thus B fails to enter the critical section. Then, while B is between instructions, C increments and immediately decrements S. B and C continue in this fashion indefinitely causing S to vary between 0 and -1, never reaching +1. Since every decrement occurs when S=0, the critical section is never entered and thus the PV-property is not satisfied.

Note that this race condition, unlikely when just three processors are involved, becomes steadily more probable as we increase the number of processors trapped in the semaphore.

3.2 Implementing PVchunk - In order to solve the readers/writers and other synchronization problems, it is convenient to define PVchunk operations where the increment e applied to the public variable S is not restricted to ±1. We write these operations as PC and VC and implement them using the same test-modify-retest paradigm seen above for P and V. The following code assumes that S has been initialized to some positive integer.

```
Procedure PC(S,e)
    Repeat
        If S-e ≥ 0 Then
            If RepAdd(S,-e) ≥ 0 Then OK <-- true
            Else { RepAdd(S,e)
                   OK <-- false }
    Until OK
End Procedure

Procedure VC(S,e)
    RepAdd(S,e)
End Procedure
```

3.3 Remark - It is worth noting that Dijkstra [6] considered the replace-add operation and examined the NaiveP procedure considered above, noting essentially the same race condition that we have discussed. Dijkstra concluded that the replace-add was a less appropriate coordination primitive than a simpler "swap" instruction. However, this conclusion becomes progressively less acceptable as the number of PEs grows larger since the swap instruction leads to serial bottlenecks.

4.0 THE TEST-MODIFY-RETEST PARADIGM

The previous section showed the need to test a semaphore before a decrement-and-test operation is applied. Since such test-decrement-retest (and corresponding test-increment-retest) sequences occur often, we define two procedures, each embodying one of these two basic sequences, which are used throughout the remainder of this report.

```
Boolean Procedure TDR(S,Delta)
    If S-Delta ≥ 0 Then
        If RepAdd(S,-Delta) ≥ 0 Then
            TDR <-- True
        Else { RepAdd(S,Delta)
               TDR <-- false }
End Procedure
```

```
Boolean Procedure TIR(S,Delta,Bound)
    If S+Delta ≤ Bound Then
        If RepAdd(S,Delta) ≤ Bound Then
            TIR <-- true
        Else {  RepAdd(S,-Delta)
              TIR <-- false  }
End Procedure
```

Using TDR the PC procedure of section 3.2 can be expressed as simply:

```
Procedure PC(S,e)
    Repeat Until TDR(S,e)
End Procedure
```

5.0 READERS AND WRITERS

In preparation for the more complex problems to be considered below, we now use the PC and VC operations to solve the well known readers-writers problem, in which a group of "reader" processes and "writer" processes are to share the use of a resource. Many readers may use the resource simultaneously, but all other processes become blocked as soon as a single writer is active.

The basic idea behind the following simple solution is to maintain a counter equal to n(1-w)-r, where n is (no less than) the maximum possible number of active readers in the system, and r and w equal the number of active readers and writers respectively.

```
Procedure Reader
        PC(S,1)
        read-body
        VC(S,1)
End Procedure

Procedure Writer
        PC(S,n)
        write-body
        VC(S,n)
End Procedure
```

Note that, in the absence of writers, no serial code is executed by the above implementation. In contrast, standard "test and set" based implementations use (very small) critical sections to protect the adjustment of their counters. Although the simple solution given above allows a continuing stream of readers to lockout all writers and vice-versa, solutions avoiding these potential lockouts are given in [12] and [20].

6.0 MANAGEMENT OF HIGHLY PARALLEL QUEUES

Although at first glance the important problem of queue management may appear to require use of at least a few inherently serial operations, we show in this section that a queue can be shared among processors without using any code that might create serial bottlenecks. The procedures to be shown next maintain the basic first-in first-out property of a queue, whose proper formulation in the assumed environment of large numbers of simultaneous insertions and deletions is as follows: If insertion of a data item p is completed before insertion of another data item q is started, then it must not be possible for a deletion yielding q to complete before a deletion yielding p has started.

Since queues are the central data structure for many algorithms, a concurrent queue access method can be an important tool for constructing parallel programs. When analyzing one of their parallel shortest path algorithms, Deo et al. [4] dramatize the need for this tool.

> "However, regardless of the number of processors used, we expect that algorithm PPDM has a constant upper bound on its speedup, because every processor demands private use of the Q."

6.1 The Algorithm - In the algorithm below we represent a queue of length Size by a public circular array Q[0:Size-1] with public variables I and D pointing to the locations of the items last inserted and deleted (these correspond to the rear and front of the queue respectively). Thus MOD(I+1,Size) and MOD(D+1,Size) yield the locations for the next insertion and deletion, respectively. Initially I=D=0 (corresponding to an empty queue).

We maintain two additional counters, #Ql and #Qu, which give lower and upper bounds respectively on the number of items in the queue and which never differ by more than the number of active insertions and deletions. Initially #Ql=#Qu=0, indicating no activity and an empty queue. The parameters QueueOverflow and QueueUnderflow, appearing in the code shown below, are flags denoting the exceptional conditions that occur when a processor attempts to insert into a full queue or delete from an empty queue. The actions appropriate for the QueueOverflow and QueueUnderflow conditions are application dependent: One possibility is simply to retry an offending insert or delete; another possibility is to proceed to some other task.

Code for a critical-section-free implementation of Insert and Delete is given below. The insert operation proceeds as follows: First a TIR is used to guarantee the existence of space for the insertion, and to increment the upper bound #Qu. If the TIR fails, a

QueueOverflow occurs. If it succeeds, the expression Mod(RepAdd(I,1),Size) gives the appropriate location for the insertion while simultaneously updating the insert pointer, and the insert procedure waits its turn to overwrite this cell (this point is discussed below). Finally, the lower bound #Ql is incremented. The delete operation is performed in a symmetrical fashion; the deletion of data can be viewed as the insertion of vacant space.

```
Procedure Insert(Data,Q,QueueOverflow)
    If TIR(#Qu,1,Size) Then {
        MyI <-- Mod(RepAdd(I,1),Size)
        Wait turn at MyI
        Q[MyI] <-- Data
        RepAdd(#Ql,1)
        QueueOverflow <-- False  }
    Else QueueOverflow <-- True
End Procedure

Procedure Delete(Data,Q,QueueUnderflow)
    If TDR(#Ql,1) Then {
        MyD <-- Mod(RepAdd(D,1),Size)
        Wait turn at MyD
        Data <-- Q[MyD]
        RepAdd(#Qu,-1)
        QueueUnderflow <-- False  }
    Else QueueUnderflow <-- True
End Procedure
```

6.2 Cell Contention - Since we assume that PEs can execute at widely differing rates (due, for example, to memory contention, see section 12), it is possible that many active insert and delete operations can have been assigned the same queue cell location L. When the queue is nearly full or nearly empty, conflicts involving one insert and one delete are reasonably likely (but a simple "cell-vacant" flag would be sufficient to resolve them). However, the circular array structure allows the (unlikely) possiblity that many active insert and delete operations all attempt to address the same cell simultaneously. We prevent this anomaly by associating semaphores with each cell (see [12] for details).

6.3 Avoiding Integer Overflows - Care is required to avoid potential overflows of the I and D counters caused by the combination of small word size, large numbers of processors, and high queue insertion rate. Since we need only maintain the values of I and D modulo the queue size, we may bound the size of I by inserting the statement:

If I ≥ MaxInt-#PE Then RepAdd(I,-Size) ,

where MaxInt is the largest representable integer, immediately before the statement that increments I. Since many, even all, processors may execute this statement simultaneously, we require that MaxInt-#PE-#PE*Size ≥ MinInt .

7.0 OTHER PARALLEL DATA STRUCTURES

Having discussed queues, we now briefly indicate how the replace-add operation can be used to provide highly concurrent access to other important data structures. A more detailed presentation is found in [12].

7.1 Task Queues - In order to define a queue-like data structure appropriate for the scheduler component of a highly parallel operating system, the queue mechanism described above should be enhanced to permit insertions of items tagged with priorities and multiplicities. In such a queue, each item i has an associated multiplicity mi indicating the number of times i is to be deleted before it is actually removed from the queue, i.e. the pair (i,mi) represents mi consecutive entries of item i in a much longer (hypothetical) queue. The parallel operating system we envision needs to support the following two primitives:

1. RequestPE(N,P,CodeBlock) - whereby a request is made for N processes to execute a block of code at priority P.

2. ReleasePE - whereby the PE invoking this primitive announces that it has completed its assigned task and is available for reassignment.

The scheduler responds to the first primitive by inserting CodeBlock onto the task queue with priority P and multiplicity N. To implement the second primitive the scheduler deletes an entry from the task queue and transfers control to the corresponding CodeBlock.

7.2 Stacks - A stack of length Size is implemented as a public array S[0:Size-1] with a public variable Top indicating the current top of the stack. A push operation addresses stack location S[RepAdd(Top,1)] thereby incrementing Top and a pop operation addresses stack location S[RepAdd(Top,-1)+1] thereby decrementing Top. Since simultaneous pushes and pops can all address the same stack location, a (small) parallel access queue into which pushes insert items and from which pops delete items is associated with each stack location.

7.3 Avail Lists - Parallel access to the free space (avail) list used by the linked allocation scheme described in Knuth [16] is acheived by maintaining a queue of pointers to free blocks. Acquiring (resp. returning) blocks is accomplished by deleting from (resp. inserting into) this queue of pointers.

8.0 DETECTING COMPLETION OF PARALLEL ACTIVITY

Since the cessation of activity involving a shared resource often indicates completion of a given task, it is important to be able to detect this event. In this report we consider a typical example, namely detecting the situation in which a shared queue is and will remain empty, i.e. when all the PEs are trying to delete from an empty queue. This is the natural termination condition for applications in which multiple PEs, each acting as both a producer and as a consumer, use a global queue to buffer data items which they pass among themselves.

If the problem of detecting completion is temporarily ignored, the following code typifies such applications:

```
Cycle {
    If producer cycle Then {
        produce data
        Repeat Insert(Data,Q,Overflow)
        Until Not Overflow }
    Else { Comment: consumer cycle.
        Repeat Delete(Data,Q,Underflow)
        Until Not Underflow
        consume data }}
```

However, the queue Underflow condition generated by Delete is not sufficient to signify task completion since inserts may still occur after the Underflow condition has occurred. Thus, to detect a state in which all PEs are trying to delete from an empty queue (this state is denoted T), we must modify the code shown above, which we do as follows. When a queue-underflow occurs, instead of retrying the delete, we increment a counter W which is then compared with #PE. If they are equal, state T has occurred. If not, the PE loops until either the queue becomes nonempty, in which case W is decremented and the deletion is retried; or until W equals #PE and state T has occured. The detailed code follows:

```
Comment: Initially W = 0.
Cycle {
    If producer cycle Then {
        produce data
        Repeat Insert(Q,Data,Overflow)
        Until Not Overflow }
    Else { Comment: Consumer cycle.
        Repeat Delete(Data,Q,Underflow)
            If Underflow Then {
                RepAdd(W,1)
                Repeat Until W = #PE Or #Q1 > 0
                If W=#PE Then Comment: state T.
                Else RepAdd(W,-1) }
        Until Not Underflow
        consume data }}
```

9.0 HARDWARE IMPLEMENTATION

In this section we show to implement an omega-network enhanced to enable the network to process multiple replace-add operations in a highly parallel manner. See [1] and [18] for a description of Omega-networks and figure 1 for an illustration.

We suppose that P = 2**D PEs are to communicate with a like number of memory modules (MMs) and define a memory cycle to be the time required for a single PE, in the absence of any other communication traffic, to transmit a request to an MM and then receive a response. This cycle time equals the MM access time plus twice the network transmission time.

9.1 Implementing Loads And Stores

- The well known manner in which an omega-network can be used to implement memory loads and stores relies on the existence of a (unique) path connecting each PE-MM pair. Requests from PEs to MMs are transmitted along these paths and responses are transmitted along the reverse paths. Unfortunately, however, two concurrent requests conflict whenever the corresponding paths are not edge disjoint.

One way to resolve these conflicts is to "kill" one of the two requests and have it resubmitted by the PE. Despite the primitive nature of this scheme, proposed by Burroughs Corporation for their FMP [3], it exhibits good average case behaviour (see [13] and [26]). Alternatively, we may resolve conflicts by enqueuing one of the two conflicting requests in the first switch at which they conflict (see [11]). In the sequel we do not use this queuing technique: For expository purposes, we prefer the simpler Burroughs approach.

It is worth noting that some conflicts are "favorable": When concurrent loads and stores are directed at the same memory location and meet at a switch, a scheme described below shows how they can be combined (and thereby satisfied) without introducing any delay. Moreover, by determining the most favorable serial order for these simultaneous requests, an enhanced switch can combine them efficiently. The actions appropriate for each favorable conflict are as follows (some of these optimizations appear in the CHoPP design, see [24] and [15]):

1. Load-Load: Transmit one of the two (identical) loads and return to each the value obtained from memory.

2. Load-Store: Transmit the store and return its value to satisfy the load.

3. Store-Store: Transmit either store and ignore the other.

Favorable conflicts reduce communication traffic and thereby increase the percentage of satisfied requests. Since combined requests can themselves be combined, any number of concurrent memory references to the same location can all be satisfied in one memory cycle (assuming the absence of conflicts with requests destined for other memory locations).

9.2 Implementing Replace-Add - The replace-add operation can be realized by augmenting the MMs with adders and connecting them, via an omega-network, to the PEs: When a RepAdd(X,e) operation is transmitted through the network to the MM containing X, the value of X and the transmitted e are brought to the MM adder, and the sum is both stored in X and returned through the network to the requesting PE.

Since we expect that concurrent replace-add operations will frequently reference the same memory location, efficient performance in the case of favorable conflicts is very important. Fortunately, by including memory and an adder in each switch, the network can achieve for replace-adds the excellent performance described above for loads and stores. (Note that, although we will continue to use the term "switch" for the devices located at the nodes of the enhanced omega-network, these devices are fuctionally closer to microprocessors than to simple switches and thus introduce non-trivial delays.)

When two replace-adds referencing the same public variable, say RepAdd(X,e) and RepAdd(X,f), conflict at a switch, we effect the serialization order "RepAdd(X,e) immediately followed by RepAdd(X,f)". This is done as follows: The switch forms the sum e+f, transmits the combined request RepAdd(X,e+f), and stores the value f in its local memory (see figure 2). When the value Y is returned to the switch (in response to RepAdd(X,e+f)), the switch returns Y to satisfy the original request RepAdd(X,f) and returns Y-f to satisfy the original request RepAdd(X,e). If there was no other conflict, Y = X+e+f; thus the values returned are X+e and X+e+f and the memory location X receives this Y value X+e+f. If other RepAdd(X,g) are simultaneously processed, the combined requests are themselves combined and the associativity of addition guarantees that the procedure gives a result consistent with the serialization principle.

In summary, the switches process favorable replace-add conflicts as follows:

1. RepAdd-RepAdd. As described above, a combined request is transmitted and the result used to satisfy both replace-adds.

2. RepAdd-Load. Treat Load(X) as RepAdd(X,0).

3. RepAdd(X,e)-Store(X,f). Transmit a store of e+f and satisfy the replace-add by returning e+f.

The above scheme reduces communications traffic and exhibits good average case performance. A detailed analysis and hardware design will appear in [20].

10.0 SUMMARY

Since the relative cost of serial bottlenecks rises linearly with the number of PEs present, elimination of such bottlenecks will become steadily more important in future parallel processors. By exhibiting replace-add based bottleneck-free implementations for several important operating system primitives, and by presenting an efficient hardware realization of the replace-add operation, we hope to have shown that this operation is an appropriate synchronization tool for ultra-large scale parallel machines. We note that our replace-add implementation avoids the hardware bottleneck usually associated with concurrent access to a single memory location.

Recall that in the absence of writers, no serial code is executed by our readers-writers implementation and that completely parallel behavior is also exhibited by our queue access method (unless the queue in question is full or empty). In contrast, standard "test and set" based solutions to the readers-writers and queue management problems use (very small) critical sections to protect the adjustment of their counters. We note for example that paracomputer simulations indicated a serial bottleneck in our parallel codes for radiation transport until we replaced standard queue access methods with the ones given above.

Since we expect that on chip delay times will typically be less than chip to chip transmission times, the network overhead imposed by supporting the replace-add operation should not degrade network transmission time significantly. Therfore, we believe that future parallel processors, utilizing something close to the hardware design presented above, can realize the replace-add in very little more than the execution time required for a public memory reference. We note that the "ultracomputer" group at N.Y.U. is developing a preliminary design for a prototype machine and operating system incorporating the ideas presented above.

Acknowledgement

The authors thank Jack Schwartz and Clyde Kruskal for technical contributions, and thank Kevin McAuliffe, Marc Snir and Jim Wilson for carefully reading preliminary versions of this paper.

REFERENCES

[1] V. E. Benes, Mathematical Theory of Connecting Networks and Telephone Traffic, Academic Press, NY, 1965.

[2] James E. Burns, Michael J. Fischer, Paul Jackson, Nancy A. Lynch, Gary L. Peterson, "Shared Data Requirements for Implementations of Mutual Exclusion Using a Test-and-Set Primitive", Proc. 1978 Intern. Conf. on Parallel Processing, pp. 79-87.

[3] Burroughs Corp., Numerical Aerodynamic Simulation Facility Feasibilty Study, NAS2-9897, March 1979.

[4] Narsingh Deo, C.Y. Pang, and R.E. Lord, "Two Parallel Algorithms for Shortest Path Problems", Proc. 1980 Intern. Conf. on Parallel Processing, pp. 244-253.

[5] E. W. Dijkstra, "Solution of a Problem in Concurrent Programming Control", CACM 8, 1965, p. 569.

[6] E. W. Dijkstra, "Hierarchial Orderings of Sequential Processes" in Operating Systems Techniques, C. A. R. Hoare and R. H. Perrot Editors, Academic Press, NY, 1972.

[7] E. W. Dijkstra, "Self-Stabilizing Systems in Spite of Distributed Control", CACM 17, 1974, pp. 643-644.

[8] Danny Dolev, "A Comparative Study of Synchronization by Parallel Control Systems", Ph.D. Thesis, Weizmann Institute of Science, Rehovot, Israel, 1979.

[9] E. Draughon, R. Grishman, J. Schwartz, and A. Stein, "Programming Considerations for Parallel Computers", Courant Institute, N.Y.U., IMM 362, Nov. 1967

[10] M. A. Eisenberg and M. R. McGuire, "Further Comments on Dijkstra's Concurrent Programming Control Problem", CACM 15, 1972, p. 999.

[11] Allan Gottlieb and Clyde Kruskal, "A Data Motion Algorithm", Ultracomputer Note #7, Courant Institute, N.Y.U, 1980.

[12] Allan Gottlieb, Boris Lubachevsky, and Larry Rudolph, "Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors", Ultracomputer Note #16, Courant Institute, N.Y.U., 1980.

[13] Allan Gottlieb and J.T. Schwartz, "Networks and Algorithms for Very Large Scale Parallel Computation", to appear in Computer.

[14] Peter B. Henderson and Yechezkel Zalcstein, "Characterization of the Synchronization Languages for PV Systems", Proc. 1978 Intern. Conf. on Parallel Processing.

[15] David Klappholz, "Stocastically Conflict-free Database Memory Systems", Proc. Intern. Conf. on Parallel Processing.

[16] Donald E. Knuth, "Additional Comments on a Problem in Concurrent Programming Control", CACM 9, 1966, p. 321.

[17] Leslie Lamport, "A New Solution of Dijkstra's Concurrent Programming Problem", CACM 17, 1974, pp. 453-455.

[18] Duncan Lawrie, "Access and Alignment of Data in an Array Processor", IEEE Trans. C-24, 1975, pp. 1145-1155.

[19] R. J. Lipton, "Limitations of Synchronization Primitives with Conditional Branching and Global Variables", Proc. of the 6th Annual ACM Symp. on Theory of Comp., 1974, pp. 230-241.

[20] Larry Rudolph, Ph.D. Thesis, New York University, in preparation.

[21] J. T. Schwartz, "Ultracomputers", ACM TOPLAS, 1980[a], pp. 484-521.

[22] Howard J. Siegel, "Single Instruction - Multiple Data Stream Machine Interconnection Design", Proc. 1976 Intern. Conf. on Parallel Processing, pp. 272-280.

[23] Herbert Sullivan, Theodore BashKow, and David Klappholz, "A Large Scale Homogeneous, Fully Distributed Parallel Machine", Proc. of the 4th Annual Sump. on Comp. Arch., 1977, pp. 105-125.

[24] Herbert Sullivan and Lenard Cohn, U.S. pattent application pending, 1979.

[25] H. Vantilborgh and A. vanLamsweerde, "On an Extension of Dijkstra's Semaphore Primitives", Inf. Proc. Let. 1, 1972, pp. 181-186.

[26] L. G. Valiant, "Experiments with a Parallel Communications Scheme", presented at the 18th Allerton Conf. on Communication, Control, and Computing, 1980.

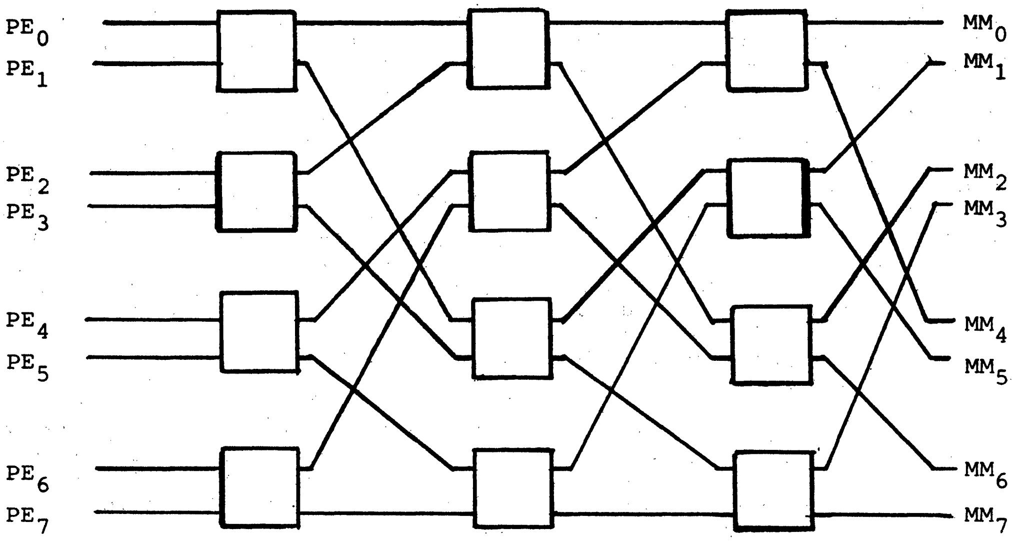


Figure 1. An 8-input omega network.

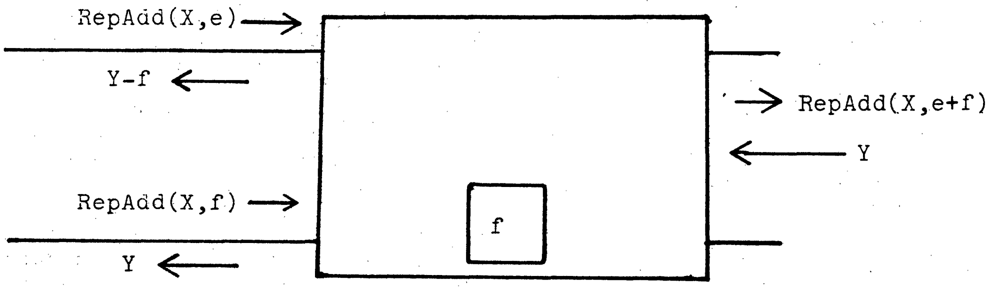


Figure 2.

Treatment of simultaneous replace-add operations addressing the same memory location.

Parallel Scheduling Algorithms*

Eliezer Dekel and Sartaj Sahni
University of Minnesota

1. Introduction

With the continuing dramatic decline in the cost of hardware, it is becoming feasible to economically build computers with thousands of processors. In fact, Batcher [1] describes a computer (MPP) with 16,384 processors that is currently being built for NASA. In coming years, one can expect to see computers with a hundred thousand or even a million processing elements. This expectation has motivated the study of parallel algorithms.

Since the complexity of a parallel algorithm depends very much on the architecture of the parallel computer on which it is run, it is necessary to keep the architecture in mind when designing the algorithm. Several parallel architectures have been proposed and studied. In this paper, we shall deal directly only with the single instruction stream, multiple data stream (SIMD) model. Our techniques and algorithms readily adapt to the other models (eg: multiple instruction stream multiple data stream (MIMD) and data flow models). References to several papers dealing with algorithms for SIMD machines can be found in [2].

When measuring the effectiveness of a parallel algorithm, one needs to consider both its complexity as well as its cost in terms of the number of PEs used. The effectiveness of processor utilization (EPU) is the complexity of the best sequential algorithm for P divided by (number of PEs used by A * complexity of A).

2. Minimum Finish Time

When preemptions are permitted, a minimum finish time schedule for m machines is efficiently obtained using Mc Naughton's rule. Let $p_1, p_2, \ldots, p_n$ be the processing times of the n jobs. The finish time, f, of an optimal preemptive schedule is given by:

$$f = \max\{ \max_{1 \le i \le n} \{p_i\}, \frac{1}{m} \sum_{i=1}^{n} p_i \}$$

Using f, the optimal schedule may be constructed in O(n) time .

Using the parallel algorithms of [3], $\max\{p_i\}$ and $\sum_{i=1}^{n} p_i$ may be computed in O(logn) time with n/logn PEs. To obtain the actual schedule, we also need $A_i = \sum_{j=1}^{i} p_j$, $1 \le i \le n$. All the A_is can be computed in O(logn) time using n/logn PEs [3]. Let $A_Ø=Ø$. Each job i can now determine its own processing assignment by using the following rule:

```
x ← ⌈A_{i-1}/f⌉ * f - A_{i-1}
case
   :x=Ø : schedule job i on machine
        ⌈A_i/f⌉ from Ø to p_i
  :x>p_i: schedule job i on machine
        ⌈A_i/f⌉ from f-x to f-x+p_i
  :else: schedule job i on machine
        ⌈A_i/f⌉ from Ø to p_i-x
 end case
```

If we have n PEs, all the machine assignments can be computed in O(1) time. However, using only n/logn PEs, these assignments may be obtained in O(logn) time (i.e., each PE computes at most ⌈logn⌉ assignments). So, the overall scheduling algorithm has a complexity of O(logn) and uses n/logn PEs. So, its EPU is $\Omega(n/(\log n * n/\log n)) = \Omega(1)$.

3. Number of Tardy Jobs

Let $J=\{(p_i,d_i) | 1 \le i \le n\}$ define a set of n jobs. p_i is the processing time of job i and d_i is its due time. Let S be any one machine schedule for J. Job i is _tardy_ in the schedule S iff it completes after its due time d_i.

Hodgson and Moore [4] have developed an O(nlogn) sequential algorithm that obtains a schedule that minimizes the number of tardy jobs.

The problem of finding a schedule that minimizes the number of tardy jobs is equivalent to that of selecting a maximum cardinality subset U of J such that every job in U can be completed by its due time. Jobs not in U can be scheduled after those in U and will be tardy. A set of jobs U such that every job in U can be scheduled to complete by its due time is called a _feasible set_. It is well known that a set of jobs U is feasible iff scheduling jobs in U in nondecreasing order of due times results in no tardy jobs.

When $p_i=1$, $1 \le i \le n$, a maximum cardinality feasible set $\overline{U}$ can be obtained by considering the jobs in nondecreasing order of due times. The job j currently being considered can be added to U iff $|U|<d_j$. Procedure FEAS(J,b) is a slight generalization. It finds a maximum subset of J that can be scheduled in the interval [Ø,b]. DONE(i) is set to -1 if job i is not selected and is set to a number greater than Ø otherwise. If DONE(i) > Ø, then job i is to be scheduled from DONE(i) - 1 to DONE(i). The procedure itself returns a value that equals the number of jobs selected. The correctness of FEAS is easily established using an exchange argument. Its complexity is O(nlogn) as it takes this much time to order the jobs by due time.

* This research was supported in part by the Office of Naval Research under contract NØØØ14-8Ø-C-Ø65Ø.

Let J be a set of n unit processing time jobs. Let $D(i)$, $1 \le i \le k$ be the distinct due times of the jobs in J. Assume that $D(i) < D(i+1)$, $1 \le i \le k$. Let $n(i)$ be the number of jobs in J with due time $D(i)$, $1 \le i \le k$. Clearly, $\sum n(i)=n$. Let $D(0)=0$ and $n(0)=0$. Define $F(i)$ to be the value of j when procedure FEAS (Figure 4.1) has just finished considering all jobs in J with due time at most D_i. It is evident that:

```
Procedure FEAS(J,n,b)
    //select a maximum number of jobs
    for processing in [0,b]  n=|J|//
    set J; integer n,b; global
    DONE(1:n)
    sort J into nondecreasing order of
    due times
    DONE(1:n) ← -1  //initialize//
    j ← 0
    for i ← 1 to n do
        case
        :j>b: return(j)
        :j<d_i:  j ← j+1; DONE(i) ← j
        end case
    end for
    return(j)
end FEAS
```

Figure 4.1

$$F(0) = D(0) = 0$$

$$F(i)=\min\{F(i-1)+n(i),D(i),b\},\ 1 \le i \le k \qquad (4.1)$$

Expanding the recurrence (4.1), we obtain:

$$(4.2)\ F(m) = \min\{ \min_{1 \le i \le m} \{D(i)+ \sum_{q=i+1}^{m} n(q)\},b\}$$

The maximum number of jobs in J that can be scheduled in $[0,b]$, $b>0$, so that none is tardy is $F(k)$. $F(k)$ may be efficiently computed, in parallel as follows. Let the due times of the n jobs in J be $d(1), d(2),\dots,d(n)$. Let $d(0)=0$. We may assume that $d(i)>0$, $1 \le i \le n$. The computation steps are:

Step 1: sort $d(1:n)$ into nondecreasing order.

Step 2: determine the points $r(0), \dots, r(k-1)$ in $d(0:n)$ where the due times change I.e. $r(i) < r(i+1), 1 \le i<k$ and $d(r(i)) \neq d(r(i)+1)$. Let $r(k)=n$. Clearly, $r(0)=0$, and $n(i)=r(i)-r(i-1)$ and $D(i) = d(r(i))$, $1 \le i \le k$; $D(0)=a$.

Step 3: since $D(i) + \sum_{i+1}^{k} n(q) = D(i)+n-r(i)$, we compute $F(k)=\min\{n+ \min_{0 \le i \le k} \{D(i)-r(i)\},b\}$ []

With n^2 PEs, step 1 can be carried out in O(logn) time [5]. Using n-1 PEs, the boundary points can be found in O(1) time. PE(i) simply checks to see if $d(i)<d(i+1)$, $1 \le i \le n-1$. If so, then i is a boundary point. 0 and n are also boundary points. The boundary points have now to be moved into memory positions $r(0),r(1),\dots,r(k)$. This can be done in O(logn) time using n PEs and the data concentration algorithm of [7]. Another data concentration step moves $d(r(0))$, $d(r(1))$, ..., $d(r(k))$ into $D(0)$, $D(1)$, ..., $D(k)$. Using k+1 PEs, $D(i)-r(i)$, $0 \le i \le k$ can be computed in O(1) time. $\min\{D(i)-r(i)\}$ can be obtained in O(logk) time using the binary tree computation model of [3]. As explained in [3], only O(k/logk) PEs are needed for this; but using k/2 PEs is faster). F(k) can now be computed using an additional O(1) time. The overall complexity is therefore O(logn) and n^2 PEs are used. The EPU of the above algorithm is $\Omega((n\log n/(\log n * n^2)) = \Omega(1/n)$.

4. Conclusions

The extent to which parallel computers will find application will depend largely on our ability to find efficient algorithms for them. The reader is referred to [6] for further examples of efficient parallel algorithms.

References

1. Batcher, K. E., "MPP - a massively parallel processor," proc. 1979 Int. Conf. on Parallel Processing, IEEE, p 249, 1979.
2. Dekel, E., Nassimi, D., and Sahni S., "Parallel matrix and graph algorithms," Department of Computer Science, University of Minnesota, TR 79-10, 1979, to appear in SIAM Computing.
3. Dekel, E. and Sahni, S.,"Binary Trees and papallel scheduling algorithms," Department of Computer Science, University of Minnesota, TR 80-19, 1980.
4. Moore, J. M., "An n job, one machine sequencing algorithm for minimizing the number of late jobs," Management Sci. 15, PP. 102-109, 1968.
5. Muller, D. E., and Preparata, F. P., "Bounds to complexities of networks for sorting and for switching," JACM, Vol. 22, No. 2, April 1975, pp. 195-201.
6. Dekel, E. and Sahni, S., "Parallel scheduling algorithms", University of Minnesota, Technical Report TR 81-1, 1981.
7. Nassimi, D. and Sahni, S., "Data broadcasting in SIMD computers," IEEE Computers, c-30, no. 2, Feb 81, 101-107.

OPTIMAL LOAD BALANCING STRATEGIES FOR A MULTIPLE PROCESSOR SYSTEM

Lionel M. Ni
Department of Computer Science
Michigan State University
East Lansing, MI. 48824

Kai Hwang
School of Electrical Engineering
Purdue University
West Lafayette, IN. 47907

Abstract -- To balance the workload among multiple processors is of fundamental importance in enhancing the performance of a multiple processor system (MPS). Optimal probabilistic load balancing policies are studied in this paper. Multiple processor systems are classified into four categories according to homogeneous versus heterogeneous processors and single-job class versus multiple-job classes. Closed-form solutions are derived for scheduling an MPS with single job class. An optimal load balancing algorithm is developed for an MPS with multiple job classes. The probabilistic scheduling policy is easy to be implemented in an MPS and can be extended to optimize message routing in a computer communications network.

I. Introduction

A loosely coupled Multiple Processor System (MPS) consists of multiple number of independent processors receiving jobs from a common job scheduler [2,3]. Such MPSs are considered a kind of distributed computer systems. The motivation to develop MPS is to allow resources sharing and to achieve higher system throughput and reliability. The objective of this study is to develop optimal load balancing techniques for achieving the above goals. The system performance of such an MPS is generally indicated by the average job turnaround time.

In an MPS, the job scheduler is responsible to dispatch jobs among several processors. An arriving job is routed to one of the processors according to the scheduling policy and the job characteristics. Load balancing can be done either deterministically or probabilistically. The deterministic routing assigns the next processor depending on the current state of the system. The probabilistic routing dispatches jobs in a proportionate approach, which is independent of the system state.

Only probabilistic routing strategies are considered in this paper. The job scheduling probabilities are solved for each job class to each of the processors with given workload and job assignment pattern. As a result, the minimal average job turnaround time can be achieved. An optimal deterministic routing policy (if exist) should provide a better system performance than that provided by an optimal probabilistic routing policy [6]. To prove the optimality of a specific deterministic routing policy is a nontrivial task. Usually, a deterministic routing policy must be compared with other routing policies to display its superiority. Most performance evaluation under a deterministic routing policy is conducted on MPS with only two or three processors [2,3,5]. A probabilistic job routing policy is easier to implement in MPSs with arbitrary number of processors. The scheduling overhead is low, because current processor information is not needed. The probabilistic approach can be also used to evaluate existing deterministic routing policies.

Recently, Chow and Kohler [3] presented a queueing model to analyze a single-job-class and heterogeneous MPS. They proposed a proportional branching policy, which assigns the job scheduling probability in proportional to the processing speed of the processor. The proportional branching policy can prevent the queue from saturation, but cannot minimize the average job turnaround time. For the deterministic case, they presented an approximated numerical method to analyze a two-processor heterogeneous MPS. Towley studied the deterministic routing in a closed queueing network [9]. A single-server processor-sharing system with many job classes has been studied by [4]. Baskett, et al. studied the behavior of queueing networks with different classes of customers [1].

Some related researches were conducted by [6,7] in packet switched computer communications networks. Computer network generally assumes fixed routing (probabilistic routing), since it is easy to describe by means of a routing table. Adaptive routing (deterministic routing), on the other hand, is complex to describe, and requires simulation to evaluate channel flows and delays. Furthermore, it was shown by [7] that at steady state, flow patterns and delays induced by good adaptive routing policies are very close to those obtained with optimal fixed routing policies. Foschini [6] studied deterministic routing policies in a packet switched network with multiple packet classes, where the outgoing trunks have different capabilities. He employed a diffusion analysis to study the effect of routing strategies under a nearly overloaded situation.

Optimal solutions to the load balancing problem are developed in this paper for a multiple processor system with single job class. The proportional branching policy suggested by Chow and Kohler [3] is formally proved to be nonoptimal. This study extends the MPS environment from single job class to multiple job classes. An optimal algorithm is developed to calculate the optimal job scheduling probabilities for each processor with multiple job classes. A comparison of various load balancing policies for MPSs is also given.

II. System Classification and Scheduling Models

A homogeneous MPS contains identical processors. Whereas, a heterogeneous MPS contains different processors. Depending on the processor

0190-3918/81/0000/0352$00.75 © 1981 IEEE

capability and assignability, jobs are classified into multiple classes. Different classes of jobs are to be assigned to different subsets of processors. In terms of processor capabilities and job classes, an MPS can be classified into one of the following four categories.

SCHO: Single job Class HOmogeneous system.
SCHE: Single job Class HEterogeneous system.
MCHO: Multiple job Classes HOmogeneous system.
MCHE: Multiple job Classes HEterogeneous system.

Queues of jobs are formed at each processor based on the stochastic nature of job arrival and given job classification. The single most important performance measure of an MPS is the average job turnaround time. This includes the time from the submission of a job through the dispatcher to its completion by one of the processors. SCHE systems have been studied by Chow and Kohler [2]. A queueing model for an SCHE system is shown in Fig.1a. Jobs from the same class are dispatched to the j-th processor with probability S_j. The model can be generalized to consider multiple classes of job arrivals to the dispatcher as depicted in Fig.1b. This queueing network is used to model the scheduling environment of an MCHE system. With minor modification, it can be applied to other three classes of multiple processor systems as well.

Each processor in the MPS is modeled by an M/M/1 queue. Let n be the total number of processors and m be the total number of distinct job classes. For n processors, we have n independent M/M/1 queues. The i-th job class has a Poisson arrival rate with mean λ_i. The j-th processor has an exponentially distributed service rate with mean μ_j. Upon the arrival of a new job, the job dispatcher is responsible for assigning the job to one of the processors. The probabilistic scheduling policy is independent of the state of the system. The state of the system is represented by the number of jobs in each of the queues at any instance. The first-come first-served (FCFS) queueing discipline is assumed, and jockeying is not allowed in this study.

Let $M=\{1,2,\ldots,m\}$ and $N=\{1,2,\ldots,n\}$ be two sets representing indices of job classes and processors respectively. Jobs in different classes arrive independently. The total job arrival rate , λ, is the sum of all different classes of job arrival rates, λ_i. The job assignment matrix $\underline{A}$ $=(a_{ij})$ is an m by n matrix, where a_{ij} indicates that the i-th class job can be executed on the j-th processor; $a_{ij}=0$ otherwise. The job scheduling matrix $\underline{S}=(s_{ij})$ is an m by n matrix, where s_{ij} is the probability of the i-th class job being assigned to the j-th processor. Obviously, $s_{ij}=0$ if $a_{ij}=0$. After the job scheduling matrix is determined, the actual job arrival rate, λ'_j, to the j-th processor can be expressed by

$$\lambda'_j = \sum_{i\varepsilon M} s_{ij}\lambda_i$$

Since each arrival source is a Poisson process, the linear combination of them is also a Poisson process with mean arrival rate λ'_j. Hence, we have

$$\lambda = \sum_{i\varepsilon M} \lambda_i = \sum_{j\varepsilon N} \lambda'_j \qquad (1)$$

Once the scheduling matrix $\underline{S}$ is determined, the model in Fig.1b can be decomposed into n independent M/M/1 queues, where the j-th queue has mean arrival rate λ'_j and service rate μ_j respectively. All queues behave independently but constrained by the linear relation in Eq.(1).

An M/M/1 queue is solvable under the unsaturated condition, $\lambda'_j < \mu_j$. At equilibrium state, the average job turnaround time among jobs serviced by the j-th processor is calculated by $T_j = 1/(\mu_j - \lambda'_j)$ for all j. We want to find a particular assignment of λ'_j satisfying Eq.(1) to minimize the average of all T_j's. Specifically, we define the average job turnaround time

$$T = \sum_{j\varepsilon N} T_j(\lambda'_j / \lambda) \qquad (2)$$

The problem of finding an optimal job scheduling matrix resulting in a minimal average job turnaround time can be formulated as a nonlinear programming problem as follows:

$$\text{Minimize } T = \sum_{j\varepsilon N} T_j(\lambda'_j / \lambda)$$

provided that

$$\lambda_i < \sum_{j\varepsilon N} a_{ij}\mu_j \qquad \text{for } i\varepsilon M \qquad (3)$$

$$\lambda < \sum_{j\varepsilon N} \mu_j \qquad (4)$$

subject to

$$s_{ij} \geq 0 \qquad \text{for } i\varepsilon M,\ j\varepsilon N \qquad (5)$$

$$\sum_{j\varepsilon N} s_{ij} = 1 \qquad \text{for } i\varepsilon M \qquad (6)$$

$$\lambda'_j = \sum_{i\varepsilon M} \lambda_i s_{ij} < \mu_j \qquad \text{for } j\varepsilon N \qquad (7)$$

Condition in Eq.(3) prevents any one class of jobs from saturating the system. Condition in Eq.(4) ensures that the total job arrival rate is less than the total service rate. Constraint in Eq.(7) prevents any processor from saturation during the scheduling process.

III. Optimal Load Balancing with Single Job Class

In a single job class environment, the job assignment matrix $\underline{A}$ is a 1 by n row matrix with all components equal to 1. Also m=1, $\lambda=\lambda_1$. We shall use S_j to represent the probability s_{1j}. The optimization problem stated in Sec.II can be simplified to

$$\text{Minimize } T = \sum_{j\varepsilon N} S_j/(\mu_j-\lambda S_j) \qquad (8)$$

provided that

$$\lambda < \sum_{j\varepsilon N}\mu_j \qquad (9)$$

subject to

$$S_j \geq 0 \qquad \text{for } j\varepsilon N \qquad (10)$$

$$\sum_{j\varepsilon N} S_j = 1 \qquad (11)$$

$$\lambda S_j < \mu_j \qquad \text{for } j\varepsilon N \qquad (12)$$

To minimize the objective function T in Eq. (8), we employed the Method of Lagrange multiplier. Due to page limitations, proofs of all the following theorems are skipped. Interested readers may refer to [8] for details of all the proofs.

The objective function in Eq.(8) can be proved convex with respect to S_j for all j. In an SCHE system, processors may have different processing speeds. Obviously, a processor with higher service rate should have higher probability to be assigned with jobs. Without loss of generality, the service rates $\{\mu_j\}$ of the n processors are denoted in descending order

$$\mu_1 \geq \mu_2 \geq \cdot\cdot\cdot \geq \mu_n > 0 \qquad (13)$$

Theorem 1:

In an SCHE system, the job scheduling matrix S, which minimizes T and satisfies the constraints in Eqs.(10)-(12), has the following probabilities.

$$\begin{aligned} S_j &= [\mu_j-\sqrt{\mu_j}(\theta_k-\lambda)/\beta_k]/\lambda && \text{for } 1\leq j\leq k \\ &= 0 && \text{for } k<j\leq n \end{aligned} \qquad (14)$$

where

$$\theta_k = \sum_{i=1}^{k} \mu_i \quad \text{and} \quad \beta_k = \sum_{i=1}^{k} \sqrt{\mu_i}$$

and k is determined by the job arrival rate λ as follows:

$$\theta_k-\sqrt{\mu_k}\beta_k < \lambda \leq \theta_{k+1}-\sqrt{\mu_{k+1}}\beta_{k+1} \qquad \text{for } 1\leq k<n \qquad (15)$$

or

$$\theta_n-\sqrt{\mu_n}\beta_n < \lambda < \theta_n \qquad \text{for } k=n \qquad (16)$$

with this optimum assignment, we obtain

$$T_j=\beta_k/(\sqrt{\mu_j}(\theta_k-\lambda)) \qquad \text{for } 1<j<k$$

and the minimized average job turnaround time

$$T=\frac{\beta_k^2 - k(\theta_k-\lambda)}{\lambda(\theta_k - \lambda)} \qquad (17)$$

Note that $\lambda_j' = \lambda S_j$. This means that the actual job arrival rate to the j-th processor is equal to the service rate of the j-th processor subtracting a term which is proportional to the square root of the service rate of the j-th processor. In an SCHO system ($\mu_j=\mu$, for all j), the job scheduling matrix S has equal probability $S_j = 1/n$ for all j. This means that jobs are assigned randomly among processors with equal probability as expected.

In a light traffic environment, only the first k processors are assigned with jobs as stated in Theorem 1. The average job turnaround time under this circumstance is faster than the service time of any of the remaining n-k slower processors. This fact is proved in [8] by showing that $T<1/\mu_j$ for all j>k.

Chow and Kohler proposed a proportional branching policy for an SCHE system [3]. The scheduling probabilities are proportional to the service rate of processors, but independent of the job arrival rate, i.e., $S_j = \mu_j / \theta_n$ for all j. We have discovered in [8] that this proportional branching policy is not necessarily optimal.

Most scheduling studies on loosely coupled MPSs were conducted in a single-class job environment. In what follows, we compare our scheduling policy with two known policies in a single-class job environment.

(1) the proportional branching policy proposed by Chow and Kohler.

By Eq.(8), the average job turnaround time for the proportional branching policy, T_1, can be expressed as $T_1=n/(\theta_n-\lambda)$.

(2) the optimal probabilistic scheduling policy proposed by Ni and Hwang.

The average job turnaround time for the optimal probabilistic scheduling policy, T_2, was stated in Eq.(17).

(3) the deterministic scheduling policy proposed by Foschini.

A deterministic policy routes an arriving job to the processor that offers the least expected turnaround time. An arriving job is sent to the queue which has the minimum ratio of the queue length to service rate. If minimum ratio is not unique, the job dispatcher selects from the ties the one with maximum service rate. A generalized version of this policy was studied by Foschini [6]. This policy is considered the best scheduling policy for an SCHE system [3].

(4) The ideal scheduling with a single fast processor.

This corresponds to the case when a system has single processor whose service rate is the sum of service rates of all n individual processors in an MPS. This is an ideal case because the single processor has the same capability of the whole MPS but the ill effect due to load unbalancing disappeared. This ideal case is included for comparison purpose only. The single processor is a standard M/M/1 queue with service rate θ_n. The average job turnaround time of such a system equals $T_4 = 1/(\theta_n-\lambda)$.

There is no doubt that the deterministic scheduling policy will result in the least turnaround time. However, closed-form solution of the average job turnaround time can not be obtained for deterministic scheduling. One approach to obtain a meaningful solution requires to perform extensive simulation experiments which are rather time-consuming. Chow and Kohler developed an efficient technique for analyzing deterministic scheduling in a two-processor SCHE system. Their solutions are accurate only for a light traffic environment. When the job arrival rate approaches the total service rate, the accuracy begins to deteriorate.

Consider a two-processor SCHE system. When $\mu_1 = 4$ and $\mu_2 = 1$, we observed from Fig.2 that

$$T_1 \geq T_2 \geq T_3 \geq T_4 \qquad (18)$$

for any choice of λ. Note that when λ is close to the total service rate, T_3 can not be obtained due to the light traffic assumption made by Chow and Kohler. Under light load conditions, T_2 and T_3 approach the performance of the single fast processor, because most of the jobs are assigned to the fast processor. When the arrival rate increases, the deterministic scheduling policy displays its superiority over the probabilistic scheduling policy. The rapidly declined performance of the proportional branching policy is due to its failure considering the effect of the arrival rate.

IV. Environment of Multiple Job Classes

The number of unknown scheduling probabilities for a multiple-job-classes environment equals the number of nonzero elements in the assignment matrix $\underline{A}$. In terms of the unknown scheduling probabilities, s_{ij}, the objective function can be expressed as

$$T = \frac{1}{\lambda}\left[\sum_{j=1}^{n}\left(\sum_{i=1}^{m}\lambda_i s_{ij}\right)/\left(\mu_j - \sum_{i=1}^{m}\lambda_i s_{ij}\right)\right] \qquad (19)$$

There are two obstacles which prevent a direct solution of Eq.(19). First, the objective function T can not be proved to be convex. Secondly, even if T is convex, the method of Lagrange multiplier cannot be used to simplify the problem, because at least m Lagrange multipliers are required.

In our model, the average service rate of each processor is assumed time-invariant. In other words, different classes of jobs assigned to the same processor have the same average service time. From the viewpoint of a processor, job classes do not make any difference in achieving the average service rate once a job has been assigned to it. Let us temporarily ignore the job preference restriction, that is, each job can be assigned to any of the processors. The total job arrival rate can be calculated by Eq.(1). The optimal job arrival rate assigned to each processor can be derived directly from Theorem 1. Specifically, if δ_j denotes the optimal job assignment rate to the j-th processor, then $\delta_j = \lambda S_j$ for all j. This does suggest how an optimal assignment in an MPS with multiple job classes can be achieved. If we distribute different job classes to multiple processors such that the actual job arrival rate, $\lambda_j^{'}$, equals the optimal job assignment rate, δ_j, for each of the processors, the optimal job scheduling matrix can then be calculated. More specifically,

$$\delta_j = \sum_{i\varepsilon M}\lambda_i s_{ij} = \lambda_j^{'} \qquad \text{for } j\varepsilon N \qquad (20)$$

Equation (20) is basically a set of n linear equations over more than n variables. This implies that there may exit none, or one, or infinitely many solutions to Eq.(20) subject to the constraints given in Eqs.(5) to (7). The case of no solution must be avoided; whereas, the other two cases are acceptable in the search of an optimal scheduling matrix. The following example shows a singular case in which solution does not exist.

Consider an MCHO system with two processors and two different job classes as illustrated in Fig.3 with $\lambda_1 = 4, \lambda_2 = 2$, and $\mu = 10$. Also only $a_{12}=0$ for the job assignment matrix $\underline{A}$.

From Theorem 1, we obtain $S_1 = 0.5$ and $S_2 = 0.5$. Therefore, both δ_1 and δ_2 are equal to 3. Substituting these values into Eq.(20), we obtain

$$\begin{cases} 4s_{11} + 2s_{21} = 3 \\ 2s_{22} = 3 \end{cases}$$

Obviously, $s_{22} = 1.5 > 1$ violates the constraint. Therefore, the solution does not exist. In this example, the first job class must be assigned to the first processor.· The optimization problem becomes how to find the optimal scheduling probabilities, s_{21} and s_{22}, provided that λ_1 was assigned to the first processor $(s_{11}=1)$. In general, the problem of finding the optimal job assignment rate, δ_j, with some preassignment of jobs can be formulated as follows.

Let c_j be the <u>preassigned job arrival rate</u> to the j-th processor. The preassigned rate may come from any of the job classes, but equally treated by the processor. Let η_i be the arrival rate of the i-th job class, in which the assignment has not been determined. We shall refer η_i as the <u>unassigned job arrival rate</u> of the i-th job class. The preassigned and the unassigned job arrival rates are related by

$$\eta = \sum_{i\varepsilon M}\eta_i = \lambda - \sum_{j\varepsilon N}c_j = \lambda - \gamma_n \quad \text{where } \gamma_k = \sum_{j=1}^{k}c_j \qquad (21)$$

Let S_j be the probability of jobs assigned to the j-th processor over the total unassigned jobs with arrival rate η. The problem of finding the optimal assignment rate to each processor, with some preassigned arrival rates, $\{c_j\}$, and a given total unassinged arrival rate, η, is formulated as follows:

$$\text{Minimize } T = \sum_{j=1}^{n}\frac{\eta S_j + c_j}{\lambda(\mu_j - \eta S_j - c_j)} \qquad (22)$$

provided that

$$\lambda_i < \sum_{j\varepsilon N} a_{ij}\mu_j \qquad \text{for } i\varepsilon M \qquad (23)$$

$$\lambda = \eta + \gamma_n < \sum_{j\varepsilon N} \mu_j = \theta_n \qquad (24)$$

$$c_j < \mu_j \qquad \text{for } j\varepsilon N \qquad (25)$$

subject to

$$S_j \geq 0 \qquad \text{for } j\varepsilon N \qquad (26)$$

$$\sum_{j\varepsilon N} S_j = 1 \qquad (27)$$

$$\eta S_j + c_j < \mu_j \qquad \text{for } j\varepsilon N \qquad (28)$$

It can be easily proved that the objective function in Eq.(22) is convex. Closed-form solution for the above constrained minimization problem is stated in Theorems 2 for heterogeneous multiple processor systems. Without loss of generality, we order the subscript j such that

$$(\mu_1-c_1)/\sqrt{\mu_1} \geq (\mu_2-c_2)/\sqrt{\mu_2} \geq \ldots \geq (\mu_n-c_n)/\sqrt{\mu_n} \qquad (29)$$

Theorem 2:

The optimal job assignment $\{\delta_j\}$ to a heterogeneous MPS with unassigned total arrival rate η and some preassigned rates c_j for all j, which minimizes T in Eq.(22) subject to the constraints in Eqs.(26)-(28), can be evaluated by

$$\delta_j = \begin{cases} (\mu_j-c_j)-\sqrt{\mu_j}(\theta_k-\gamma_k-\eta)/\beta_k & \text{for } 1\leq j\leq k \\ 0 & \text{for } k<j\leq n \end{cases} \qquad (30)$$

where k is determined by the unassigned job arrival rate η as follows:

$$(\theta_k-\gamma_k)-\beta_k(\mu_k-c_k)/\sqrt{\mu_k} < \eta \leq (\theta_{k+1}-\gamma_{k+1})- \beta_{k+1}(\mu_{k+1}-c_{k+1})/\sqrt{\mu_{k+1}} \qquad \text{for } 1\leq k<n$$

or (31)

$$(\theta_n-\gamma_n)-\beta_n(\mu_n-c_n)/\sqrt{\mu_n} < \eta < \theta_n-\gamma_n \qquad \text{for } k=n$$

The physical meaning of the optimal assignment in Eq.(30) can be interpreted as follows. If the traffic is very heavy, i.e. $\lambda=\eta+\gamma_n$ approaches θ_n, the optimum assignment is very close to the available capacity of the processor, i.e., $\mu_j - c_j$ for the j-th processor. When the traffic becomes light, the optimal job assignment is formed by subtracting a value proportional to $\sqrt{\mu_j}$ from the available capacity of that processor. If c_j=0 for all j, Theorem 2 becomes equilivalent to Theorem 1. If $\mu_j=\mu$ for all j in Theorem 2, Theorem 2 can be applied to a homogeneous MPS.

V. An Optimal Load Balancing Algorithm

A recursive optimal load balancing algorithm is developed below to generate the scheduling matrix $\underline{S}$ for an MPS in a multiple-job-class environment. Although there may be many solutions to $\underline{S}$, the average job turnaround time is unique and minimized for all possible solutions of $\underline{S}$. Our purpose is to find a systematic procedure to generate at least one of the possible solutions of $\underline{S}$. The notation c_{ij} is used to denote the rate of the i-th class job assigned to the j-th processor. All a_{ij}, c_{ij}, η_i, and μ_j (for $1\leq i\leq m$, $1\leq j\leq n$) are global variables. M is a set of active job classes. $i\varepsilon M$ indicates that the assignment of the i-th job class has not been determined, i.e., $\eta_i \neq 0$. N is a set of active processors. $j\varepsilon N$ indicates that $a_{kj}\neq 0$ for at least one $k\varepsilon M$. Both M and N sets are local variables.

The Load Balancing Algorithm:

Input: Global variables: a_{ij}, c_{ij}, η_i, and μ_j.
Local variables: M and N sets.

Output: Changes on those global variables.

Procedure:

1. For each $i\varepsilon M$ with $\Sigma a_{ij}=1$, find a particular k such that $a_{ik}=1$. Then set $c_{ik}\leftarrow c_{ik}+\eta_i$, $\eta_i\leftarrow 0$, and $a_{ik}\leftarrow 0$ for that i. Update M and N.
2. For those $j\varepsilon N$, calculate the corresponding job assignment rates δ_j for $j\varepsilon N$ by applying Theorem 2, where $\eta=\Sigma\eta_i$ and $c_j=\Sigma c_{ij}$.
3. Form a set N', where $j\varepsilon N'$ if $j\varepsilon N$ and $\Sigma\eta_i a_{ij}<\delta_j$. Form a set M', where $i\varepsilon M'$ if $i\varepsilon M$ and there is at least a $j\varepsilon N'$ such that $a_{ij}=1$. If $N'\neq\emptyset$, invoke this algorithm with inputs M' and N'.
4. Form M"=M-M' and N"=N-N'. For each $j\varepsilon N''$ with $\Sigma a_{ij}=1$, find a particular k such that $a_{kj}=1$ and $\delta_j\neq 0$. Set $c_{kj}\leftarrow c_{kj}+\delta_j$ and $\eta_k\leftarrow\eta_k-\delta_j$.
5. If N'=0 in (3) or at least one particular k was found in (4), invoke this algorithm again with local inputs M" and N".
6. Update M and N. Solve the following set of linear equations.

   $$\begin{cases} \sum_{i\varepsilon M} \eta_i X_{ij} = \delta_j & \text{for } j\,\varepsilon N \\ \sum_{j\varepsilon N} X_{ij} = 1 & \text{for } i\,\varepsilon M \end{cases}$$

   where X_{ij}'s are unknown variables satisfying

   $X_{ij}=0$ if $a_{ij}=0$; $X_{ij}\geq 0$ if $a_{ij}=1$ for $i\varepsilon M$, $j\varepsilon N$

   This set of linear equations always has infinitely many solutions. Picking any one solution is sufficient.
7. For $j\varepsilon M$ and $j\varepsilon N$, set $c_{ij}\leftarrow c_{ij}+X_{ij}\eta_i$ and $a_{ij}\leftarrow 0$.

In the main program:

1. Given m job classes with average arrival rates λ_i, n processors with average service rate μ_j, an m by n job assignment matrix $\underline{A}$, and conditions in Eqs.(5)-(7).
2. Initialize local variables $M=\{1,2,\ldots,m\}$ and $N=\{1,2,\ldots,n\}$.
3. Initialize global variables; $c_{ij}\leftarrow 0$ and $\eta_i\leftarrow\lambda_i$ for all $i\varepsilon M$ and $j\varepsilon N$.
4. Invoke the load balancing algorithm.

5. For $1 \leq i \leq m$ and $1 \leq j \leq n$, calculate $s_{ij} = c_{ij}/\lambda_i$.

VI. Conclusions

Optimal probabilistic load balancing policies are developed for a multiple processor system with either single job class or multiple job classes. Those policies provide a test bed to determine the superiority of any deterministic scheduling policy over probabilistic ones. With the high implementation overhead of deterministic policy, we conclude that the proposed probabilistic scheduling policy is more feasible and can be systematically implemented in commercial multiple processor systems.

Acknowledgements

This work was supported in part by the NSF research grant MCS-78-18906A02 and in part by DOT research contract R920044.

References

[1] Baskett, F., et al. "Open, closed, and mixed networks of queues with different classes of customers", J. of ACM, (April, 1975), pp. 248-260.

[2] Chow, Y.C. and Kohler, W.H., "Dynamic load balancing in a homogeneous two-processor distributed system", Computer Performance, (Eds. K.M. Chandy and M. Reiser), pp.39-52.

[3] Chow, Y.C. and Kohler, W.H., "Models for dynamic load balancing in a heterogeneous multiple processor system", IEEE Trans. on Computer, (May, 1979), pp.354-361,

[4] Fayolle, G., et al. "Sharing a processor among many job classes", J. of ACM, (July 1980), pp.519-532.

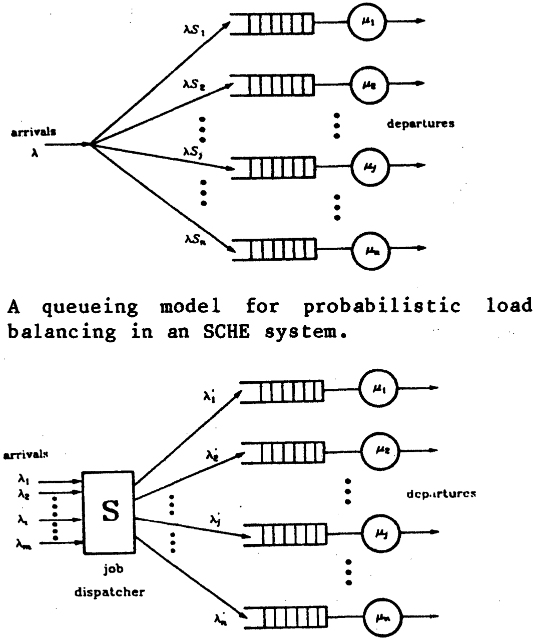
[5] Flatto, L., Two parallel queues with equal servicing rates, IBM Research Rep. RC 5916, (March 1976).

[6] Foschini, G.J., "On heavy traffic diffusion analysis and dynamic routing in packet switched networks", Computer Performance, (Eds. K.M. Chandy and M. Reiser), pp.499-513.

[7] Gerla, M. and Kleinrock, L., "On the topological design of distributed computer networks", IEEE Trans. on Comm. (January 1977), pp.48-60.

[8] Ni, L.M. and Hwang, K., Probabilistic load balancing in a multiple processor system with many job classes, School of Electrical Engineering, Purdue University, TR-EE 81-1, (January 1981), 41 pp.

[9] Towsley, D., "Queueing network models with state-dependent routing", J. of ACM, (April 1980), pp.323-337.



(a) A queueing model for probabilistic load balancing in an SCHE system.

(b) A queueing model for probabilistic load balancing in an MCHE system.

Fig. 1. Probabilistic load balancing models for a multiple processor system.

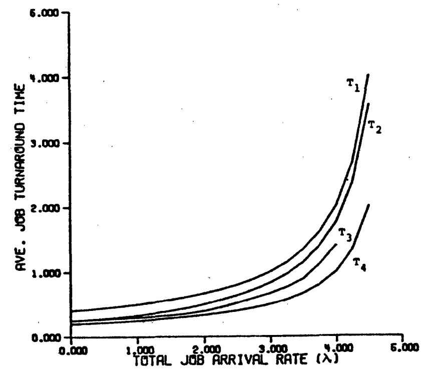


Fig. 2. Comparison of three load balancing policies for a two-processor heterogeneous system with single job class.

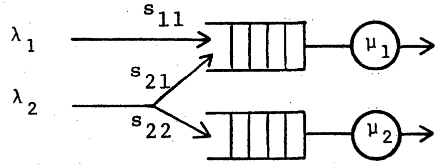


Fig. 3. The queueing model of an MCHO system with two processors and two job classes.

TASK ASSIGNMENT IN DISTRIBUTED MULTIPROCESSOR SYSTEMS *

Virginia Lo and Jane W. S. Liu
Department of Computer Science
1304 W. Springfield Avenue
University of Illinois
Urbana, Illinois 61801

Summary

Our research addresses the problem of task assignment in distributed multiprocessor systems. By distributed multiprocessor systems we mean any configuration of processors in which the cost of communication between processors is non-negligible. A set of tasks to be assigned to the processors are referred to collectively as a distributed process. In order to achieve their common goal, each of the tasks performs two activities: execution on one of the processors, utilizing the local memory and resources of that processor, and communication with one or more other tasks in the distributed process (transmission of data and/or synchronization information). An assignment of tasks to processors designates one processor for each task to reside on for the lifetime of that task and is thus a static assignment.

More precisely, we define a distributed process as a set of k tasks $T=\{t_1, t_2, \ldots, t_k\}$. In a multiprocessor system containing n processors $P=\{p_1, p_2, \ldots, p_n\}$, let x_{ij} denote the execution cost of task t_i when it is assigned to and hence executed on processor p_j. Let c_{ij} denote the communication cost between two tasks t_i and t_j if they are assigned to different processors. Throughout our discussion, we will assume that the communication cost between two tasks executed on the same processor is negligible and that communication costs are independent of processor (as in a fully-interconnected network of processors). An assignment of tasks to processors can be formally described by a function from the set of tasks to the set of processors, $f:T \rightarrow P$, and an optimal assignment is one which minimizes some prespecified performance criterion.

We consider two different performance criteria for optimal assignments of tasks to processors: minimization of the total sum of execution and communication costs and minimization of the execution and communication costs incurred by the processor with maximum cost. The latter is also referred to as minimization of latest finishing time because of the equivalence of this problem to certain deterministic scheduling problems. As an illustration of these performance criteria, consider a system of 4 tasks and 3 processors. For the assignment $f(t_1) = p_1$, $f(t_2) = p_2$, $f(t_3) = p_2$, and $f(t_4) = p_3$ the total sum of execution and communication costs is $x_{11}+x_{22}+x_{32}+x_{43}+c_{12}+c_{13}+c_{14}+c_{24}+c_{34}$ and the latest finishing time is max {f1,f2,f3} where

$$f1 = x_{11}+c_{12}+c_{13}+c_{14},$$

$$f2 = x_{22}+x_{32}+c_{12}+c_{13}+c_{24}+c_{34},$$

$$f3 = x_{43}+c_{14}+c_{24}+c_{34}.$$

Task assignment to minimize the total sum of execution and communication costs has been analyzed using a network flow model and network flow algorithms by a number of researchers [1]-[2], [6]-[8]. A system of n processors and k tasks can be modeled as a network by letting each processor be a distinguished node and each task be an ordinary node. An edge is drawn between each pair of task nodes t_i and t_j and is given the weight c_{ij}. There is an edge from each task node t_i to each processor node p_q with the weight

$$w_{iq} = \frac{1}{n-1}\sum_{p\neq q} x_{ip} - \frac{n-2}{n-1} x_{iq}. \qquad (1)$$

An <u>n-way cut</u> is a set of edges which partitions the nodes of the network into n disjoint subsets wth exactly one processor node in each subset and thus corresponds naturally to an assignment of tasks to processors. The <u>capacity of an n-way cut</u> is the sum of the weights on the edges in the cut and is exactly equal to the sum of execution and communication costs incurred by the assignment because of the judicious choice of weights according to Equation (1).

For 2 processor systems, known efficient Max Flow/Min Cut algorithms can be used to find an optimal assignment [7]. However, the problem of finding a minimum n-way cut for n > 2 is NP-complete and is thus unlikely to have any efficient (polynomial-time) solution. Therefore, we have devised the following group of efficient heuristic algorithms which together yield optimal or near optimal assignments for tasks in a general n-processor system. Simulation results indicate their performance to be very good. The group of algorithms is described collectively as Algorithm A and individually as Part I (Iterative), Part II (Lump), and Part III (Greedy).

*This work was partially supported by the U.S. Office of Naval Research under Contract No. N00014-79-C-0775 and by the U.S. Dept. of Energy under Contract No. DE-AC02-76ER02383.A003.

0190-3918/81/0000/0358$00.75 © 1981 IEEE

Part I is derived using the network model of the n-processor system described above and is designed based on the following known result [7]: Consider a network G_j obtained from the n-processor network by replacing the set of processor nodes $P - \{p_j\}$ with the node $\bar{p}_j$ and by replacing edges from each task node to the set of processor nodes $P - \{p_j\}$ with one new edge with weight equal to the sum of the weights on the replaced edges. The minimum cut in the network G_j with p_j and $\bar{p}_j$ as distinguished nodes induces a partition of nodes in G_j into two disjoint subsets, A_j containing p_j and $\bar{A}_j$ containing $\bar{p}_j$. In an optimal task assignment, tasks in A_j are assigned to processor p_j.

In each iteration of Part I, the Max Flow/Min Cut Algorithm is applied for each processor node p_j and $\bar{p}_j$ as distinguished nodes (as described above) to determine the subset of tasks assigned to p_j. The resultant assignment may be partial in that there may be tasks which remain unassigned. Let T^m denote the set of tasks which remain unassigned after m iterations. We construct a network G^{m+1} from the network G^m used in the mth iteration by deleting from G^m all task nodes not in T^m and by redefining the execution cost for t_i in T^m on processor p_j as

$x^m_{ij} = x_{ij}$ plus the sum of communication costs between t_i and all tasks already assigned to processors other than p_j.

The weight on the edge from t_i to p_j is recalculated according to Equation (1) with these new values of execution cost for all tasks in T^m. The process of applying the Max Flow/Min Cut Algorithm in the network G^{m+1} with p_j and $\bar{p}_j$ as distinguished nodes for each processor p_j is repeated. The iteration process halts when either all tasks are assigned (in which case the assignment is optimal) or when no tasks are assigned in the last iteration. In the latter case, Part II of Algorithm is invoked on the subset of tasks T^m not assigned by Part I.

II computes a lower bound L on the cost of an optimal n-way cut when more than one processor is utilized for a reduced network containing the unassigned task nodes and the processor nodes:

$$L = \sum_{t_i \varepsilon T^m} \min_{p} (x_{ip}) + \min_{i \neq r} c(p_r, p_i)$$

where $c(p_r,p_i)$ is the cost of the minimum cut for some arbitrarily chosen processor p_r and processor p_i.

Based on this lower bound, the algorithm then checks to see if it would be cheaper to assign all remaining tasks to one processor. If so, the tasks in T^m are all assigned to the one processor yielding minimum total execution cost for those tasks. In this case, the resultant assignment in combination with the assignment from Part I is optimal. Otherwise, Part III is invoked to complete the assignment.

Part III, Algorithm Simple Greedy, locates clusters of tasks between which communication costs are "large". Tasks in a cluster are then assigned to the same processor, and the resultant assignment may be suboptimal. In particular, Simple Greedy computes C, the average communication cost over all pairs of tasks. Simple Greedy then deletes all edges for which $c_{ij} < C$. Each edge $e = (t_i,t_j)$ for which $c_{ij} > C$ is then examined. Let G_i be the task cluster containing t_i and G_j be the task cluster containing t_j. The algorithm tests to see if there exists a processor for which the total execution cost for all tasks in $G_i \cup G_j$ is non-infinite. If so, the two clusters are merged into one large cluster, edges between tasks in the new cluster are deleted, and the process continues. When no more edges remain, each cluster is assigned to the processor with minimum total execution cost for the tasks in that cluster.

In order to evaluate the performance of Algorithm A, simulation runs were made on data consisting of randomly generated task-processor configurations under the assumption that tasks tend to form clusters and that communication costs between tasks within a cluster are on the average larger than communication costs between tasks in different clusters. Task systems with 6 to 20 tasks and 3 to 5 processors were simulated. The number and size of task clusters, the intra-cluster and inter-cluster communication costs, and the execution costs were all generated from uniform random distributions. In addition, simulations were performed in which the entire assignment was performed by Simple Greedy alone. The table below shows the distribution of the ratio of latest finishing time for a heuristic algorithm (Algorithm A or Simple Greedy) to the latest finishing time of an optimal algorithm.

	Optimal =1	< 11/10	< 5/4	> 5/4
Algorithm A	69%	25%	6%	0%
Simple Greedy	71%	14%	11%	4%

We have also investigated the task assignment problem using an approach based on a classical model from deterministic scheduling theory [3]-[4]. We restrict our attention to the assignment of k independent tasks on n identical processors taking into account the overhead of communication between tasks assigned to different processors. Let P, T, and (c_{ij}) be the set of processors, set of tasks, and matrix of communication costs, respectively, as before, and let let (x_i) be a vector of execution costs where x_i is the cost of executing task t_i on each of the processors. The latest finishing time (LFT) of all tasks in T for an assignment f is defined as

$$\omega_f = \max_{1 \leq j \leq n} \Big[\sum_{f(t_i)=p_j} x_i + \sum_{\substack{f(t_i)=p_j \\ f(t_r)=p_j}} c_{ir} \Big].$$

The latest finishing time is thus the sum of execution and communication costs incurred by that processor for which execution costs plus communication costs is maximal over all processors. An optimal assignment f_{OPT} is one for which latest finishing time is minimal. We note that an assignment which minimizes latest finishing time of all tasks in the set T also maximizes utilization of the processors in the system.

We restrict our attention to systems in which communication costs are a simple monotonic non-decreasing function of execution costs. This assumption can be justified when interprocess communication occurs primarily due to data exchange (e.g., when the tasks form a producer-consumer pair). It can also be justified in program behavior models in which each task corresponds to one of k disjoint program localities and communication costs incur only during transitions between localities.

For the case $c_{ij} = \alpha(x_i \cdot x_j)$ we have the following results:

(1) Let $X = \sum_{t_i \epsilon T} x_i$ and n is the number of processors. For $\alpha > \frac{n}{X}$, in an optimal schedule all tasks are assigned to one processor.

(2) The problem of task assignment to minimize LFT for task systems with both execution and communication costs taken into account is equivalent to the same problem with communication costs ignored.

(3) The Longest Processing Time heuristic (LPT) assigns tasks according to the following rule: whenever a processor becomes available, the task with the greatest execution cost among those tasks not yet assigned is assigned to the free processor. We have the following tight bound:

$$\frac{\omega_{LPT}}{\omega_{OPT}} <= \frac{4}{3} - \frac{1}{3n}$$

(4) If for all tasks t_i in T,

$$x_i \epsilon \{u, 2u, 4u, \ldots, 2^r u, \ldots\}$$

for some constant u, then the assignment produced by LPT is optimal.

For the case $c_{ij} = \alpha(x_i + x_j)$ we have the following results:

(1) Let $X = \sum_{t_i \epsilon T} x_i$ and k is the number of tasks and n is the number of processors. For $\alpha > (n-1)/(k+(n-2))$, in an optimal schedule all tasks are assigned to one processor.

(2) For the LPT heuristic we have the following loose bound: $\frac{\omega_{LPT}}{\omega_{OPT}} < 2$.

The proofs of these results can be found in [5].

Our results indicate that useful suboptimal algorithms for the task assignment problem exist in both the case where the goal is to minimize total execution and communication costs and the case where the goal is to minimize latest finishing time. The former goal takes a global view of the system and aims to minimize total resource usage. The latter goal treats concurrency as the main factor in working for optimality with respect to resource usage. Both approaches represent important concerns for task assignment algorithms.

References

[1] S. H. Bokhari, "Dual Processor Scheduling with Dynamic Reassignment," IEEE Trans. on Software Engineering (Vol. SE-5, No. 4, July 1979), pp. 341-349.

[2] W. W. Chu, L. J. Holloway, M. T. Lan, and Kemal Efe, "Task Allocation in Distributed Data Processing," IEEE Computer (Nov. 1980), pp. 57-69.

[3] E. G. Coffman, Jr., Ed., Computer and Job Shop Scheduling Theory, John Wiley and Sons, New York, (1976), 298 pp.

[4] O. J. El-Dessouki, "Program Partitioning and Load Balancing in Network Computers," Illinois Institute of Technology, Ph.D. Thesis, (Dec. 1978).

[5] V. M. Lo, "Task Assignment in Distributed Multiprocessor Systems," Department of Computer Science, University of Illinois, in preparation.

[6] H. S. Stone and S. H. Bokhari, "Control of Distributed Processes," IEEE Computer (July 1978), pp. 97-106.

[7] H. S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," IEEE Trans. on Software Engineering (Vol. SE-3, No. 1, Jan. 1977), pp. 85-93.

[8] S. B. Wuand M. T. Liu, "Assignment of Tasks and Resources for Distributed Processing," IEEE COMPCON Proceedings on Distributed Processing (Fall 1980), pp. 655-662.

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES