1980 CONFERENCE ON PARALLEL PROCESSING 80CH1569-3

PROCEEDINGS OF THE 1980 INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING

Cosponsored by the

Ohio State University
Columbus, Ohio
and the

IEEE Computer Society

IEEE Catalog No. 80CH1569–3

PROCEEDINGS OF THE 1980 INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING

Papers presented on
August 26-29, 1980

Co-Sponsored by

Department of Electrical and Computer Engineering
OHIO STATE UNIVERSITY
Columbus, Ohio

and the

IEEE Computer Society

In Cooperation with the

Association for Computing Machinery

IEEE Catalog No. 80CH1569-3

Additional copies are available from:
IEEE Computer Society
10662 Los Vaqueros Circle
Los Alamitos, CA 90720

or

IEEE Service Center
445 Hoes Lane
Piscataway, NJ 08854

IEEE Catalog No. 80CH1569-3
Library of Congress No. 79-640377

Copyright and Reprint Permissions: Abstracting is permitted with credit to the source. Libraries are permitted to photocopy beyond the limits of U.S. copyright law for private use of patrons those articles in this volume that carry a code at the bottom of the first page, provided the per-copy fee indicated in the code is paid through the Copyright Clearance Center, P.O. Box 765, Schenectady, NY 12301. Instructors are permitted to photocopy isolated articles for noncommercial classroom use without fee. For other copying, reprint or republication permission, write to Director, Publishing Services, IEEE, 345 E. 47 St., New York, NY 10017. All rights reserved. Copyright © 1980 by The Institute of Electrical and Electronics Engineers, Inc.

PREFACE

For this Ninth International Conference on Parallel Processing we received a total of 117 papers, 31 of which were from 6 countries in Europe, Canada, Israel, Japan, and the People's Republic of China. Sixty-five papers were accepted for presentation at the meeting, 21 of which are to be presented in a one- and one-half hour poster session. In a poster session visual displays of all the papers are mounted on bulletin boards, and the author of each paper is present during the entire session for explanation and in-depth discussion with interested persons. This session allowed us to accept more interesting papers than would have been otherwise possible.

The conference featured a film festival covering the history of and advances in computer architecture, and a panel session addressing the outstanding issues of designing high performance computer systems.

We would like to thank Tse-yun Feng, the conference chairman, for arranging the location of this meeting, and printing and distributing the preliminary announcements. We are indebted to Mrs. Vivian Alsip for her valuable help in keeping all the correspondence to the authors and reviewers superbly organized. We also extend our thanks to Ms. Gerrie Katz of the IEEE Computer Society for her patience and help in producing this proceedings. Finally, we thank Tse-yun Feng and K.H. Kim for handling the papers by Banerjee, Gajski, and Kuck, and Lawrie and Vora.

PROGRAM COMMITTEE
David J. Kuck
Duncan H. Lawrie
Ahmed H. Sameh

TABLE OF CONTENTS

AUTHOR INDEX

LIST OF REFEREES
1980 INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING

W.B. Ackerman
D.P. Agrawal
D. E. Atkins
J-L. Baer
U. Banerjee
Y. Bard
G.H. Barnes
D.H. Bartley
T.R. Bashkow
K.E. Batcher
G.G. Belford
K. Bowyer
M. Bozyigit
W. Brainerd
J.D. Brock
R.M. Brown
R.H. Campbell
A. Casavant
V.G. Cerf
P-Y. Chen
S-c. Chen
Y-C. Chow
K. Culik
E.S. Davidson
E.W. Davis
N. Dershowitz
D.M. Dias
M. Edelberg
O. El-Dessouki
J. Ellis
J. Emery
P. Emrath
M.D. Ercegovac
M. Faiman
A. Fantechi
E.A. Feustel
P.M. Flanders
M.A. Franklin
D.D. Gajski
D.B. Gannon
O.N. Garcia
W.M. Gentleman
W. Gillett
M.J. Gonzalez, Jr.
J.R. Goodman
J. Grcar
I. Greenberg
I. Greif
D.H. Grit
R.K. Gupta
P.E. Hagerty
W. Handler
R. Haskin
F.P. Hiner, III
L. Hollaar
E. Horowitz
K.B. Irani
A.K. Jones
H.F. Jordan
R.Y. Kain
R.N. Kapur
V. Kathail
R.M. Keller
K.H. Kim
W. Kim
T. Kimura
J.C. Knight
D.J. Kopetzky
B. Krieg-Bruckner
W.J. Kubitz
D.J. Kuck
R. Kuhn
L. Lamport
J.L. Larson
D.H. Lawrie
B. Leasure
K.Y. Lee
R. B-L. Lee
V. Lesser
G. Lindstrom
G.J. Lipovski
B. Liskov
R.E. Lord
S.F. Lundstrom
T. Macke
M. Maekawa
M. Marathe
R.C.O. Martins
K. Maruyama
R.J. McMillen
J. Metzner
R.S. Michalski
C.T. Mickelson
D. Mickunas
R. Montoye
J.D. Mooney
P.T. Mueller, Jr.
W.D. Murray
D. Nassimi
C. Neuhauser
J.D. Noe
S.E. Orcutt
S. Owicki
D. Padua
R. Paige
D.S. Parker
J. Patel
J.C. Peterson
D. Plaisted
S. Preece
C.V. Ramamoorthy
K. Ramamritham
H.K. Reghbati
J.E. Robertson
T.L. Rodeheffer
Y. Saad
S.K. Sahni
A.H. Sameh
M.S. Schlansker
R.A. Schmidt
K. Schwans
H.J. Siegel
B.J. Smith
S.D. Smith
S.W. Song
M. Sowa
V.P. Srini
J.A. Stankovic
R. Stokes
R. Towle
S.H. Unger
K. Vairavan
J. Vanaken
A.M. van Tilborg
R.G. Voight
C.R. Vora
D. Watanabe
D. Weiss
J.E. Wirsching
D.S. Wise
J. Wisniewski
M. Wolfe
C-l. Wu
W.C. Yen
P-C. Yew

SESSION 1: SOFTWARE AND LANGUAGES

A PARALLEL OPERATING SYSTEM FOR AN MIMD COMPUTER

Rodney A. Schmidt
Denelcor, Inc.
Denver, Colorado 80205

Summary

The HEP computer system developed by Denelcor, Inc. under contract to the U.S. Army Ballistics Research Laboratory is an MIMD machine of the shared resource type as defined by Flynn [1]. The architecture of this machine has been covered earlier in a paper by Smith [2]. Briefly, processes in HEP reside within tasks, which define both a protection domain and an activitation state (dormant/active). Tasks reside within processors, all of which access a shared data memory. Multiple tasks may cooperate by sharing a common region in data memory. Cells in data memory have the property of being "full" or "empty" and the execution of instructions in processes may be snychronized by busy waiting (in hardware) on the full/empty state of data memory cells. Other than the state of data memory, processes and tasks in different processors have no means of synchronization or communication.

High-level language (e.g. FORTRAN) programs in this machine are explicitly parallel. Subprograms are made to run in parallel with the main program by an explicit CREATE statement analogous to CALL in ordinary FORTRAN. Code within a subprogram is SISD. The objective of the HEP operating system is to preserve the parallelism of the user program by executing in parallel during the performance of I/O and related supervisory functions. The operating system must:

1.) Allow all user processes to execute during I/O related supervisory computation;
2.) Allow multiple concurrent supervisory I/O computations;
3.) Allow reentrant use of code in the supervisor and the user program;
4.) Provide maximum user performance by consuming minimum resource in both time and space.

In SISD computers, reentrancy is usually obtained with some form of dynamic memory allocation. Concurrency of the operating system and the user is not possible due to the SISD nature of the machine.

In HEP, most dynamic memory allocation would generate considerable serialization of code around the resource lock required to safeguard the memory allocation data structure. In addition, HEP cannot allow any memory used by the system to be writeable by the user since the user is running truly in parallel with the system and could destroy any location at any time.

In the HEP operating system, the available general purpose registers (about 2,000 of them) are divided a priori into groups of uniform length. When a process is created, the creating process must obtain a register environment from a table of available groups. This operation is relatively infrequent and inexpensive. All register environments are identical, and no state is retained in them.

Main memory (data memory) environments are obtained at the subprogram level by each subprogram as it is invoked. Space is obtained from a pool of data memory environments peculiar to that subprogram. The user must specify at link time how many such environments should be allocated for each subprogram. Control of an environment is obtained via a table of free environments, but the table is local to the subprogram. Thus, serialization for access to an environment is only between multiple, nearly simultaneous, invocations of the same subprogram, and is much less damaging to performance.

Data memory environments are a resource not visible to the user, and as such can contribute to deadlock problems. Given the user's ability to increase the amount of data memory resource allocated to a subprogram, the deadlock problem can be circumvented without much difficulty.

Concurrent I/O presents its own set of problems. In FORTRAN, a single I/O is implemented with multiple calls to I/O formatting services. State must be retained by the formatter during this process. This state is bound to the I/O unit, not the subprogram. Further, the amount of space required is not known until run time. Thus, some type of run time memory management is required, and the resource thus allocated is invisible to the user. The space must be allocated in an area accessible to all processors in a multi-processor job, so that all tasks may share the same I/O units.

The strategy employed in HEP is to allocate I/O buffers for a logical unit upon the first I/O to the unit. The space is then consumed for the duration of the program, even if the I/O unit is closed. If the I/O unit is re-opened for another file, the record length of the new file must be less than or equal to that of the old file. In this implementation, space can be allocated from a top-of-memory pointer which moves in only one direction. Serialization of processes occurs only on simultaneous first I/O operations, and only for the few microseconds required to move the pointer. This contrasts with the substantial serialization introduced by the normal scheme of a linked list of available space with garbage collection.

Consideration is being given to allowing a user to supply his own logical record buffer, with only the fixed portion of the I/O state held at the top of memory. This would allow the user greater dynamism in the logical record size, at the expense of managing his own resources.

HEP supervisors require two types of dynamic memory: registers to use while copying logical records to/from physical records, and data memory to hold file parameters for open files. Of these, the register allocation is the simplest. Since the users register requirement can be determined from the number of processes requested (a control card parameter), all remaining registers in the register memory partition can be used for supervisor I/O operations. These registers are allocated from a bit table to active I/O operations.

Data memory allocation is more difficult. It is not known until run time how many files will be used, or how much logical record buffer space will be required by the user. Fortunately, the amount of supervisor space required per open file is constant. The operating system merely allocates supervisor space for enough files to accomodate the larger system programs (compiler, etc.) and leaves the remaining space for the user. The default limit on open files may be overridden with a control card for users with special requirements.

The present HEP system provides a high-performance low overhead environment for parallel computational activities. Our next activity will be to extend this capability with high-performance parallel I/O operation with speed comparable to our processing speeds. The parallel file system will include such features as record interlock within files and concurrent read/write capability from multiple jobs to the same file.

References

[1] Flynn, M.J., "Some Computer Organizations and Their Effectiveness", IEEE-C21, (September, 1972).

[2] Smith, B.J., "A Pipelined, Shared Resource MIMD Computer", Proc. of the 1978 International Conference on Parallel Processing, (1978), pp. 6-8.

THE PROGRAMMING LANGUAGE PARALLEL PASCAL

by

Anthony P. Reeves, John D. Bruner, and Mark S. Poret

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

Summary

An extended version of the Pascal programming language for Parallel processors is described. This language reduces the semantic gap between the very popular sequential Pascal language and a large group of highly structured parallel processors. Only a small number of carefully chosen features have been added to the conventional Pascal language. A specification of the language is given in [1].

Most parallel processors are currently programmed in either assembly language or a machine-dependent special version of Fortran. In some cases, an attempt has been made to implement a sequential high level language on a parallel processor. This may work well on a tightly-coupled processor with a small number of processing elements (PE's). The advantage is that existing programs may be used without change and that programmers do not have to learn anything new. Unfortunately, sequential languages are often unsuitable for the expression of array manipulations and efficiency is lost. By contrast, since Parallel Pascal has been designed for SIMD processors, it is a high level language offering efficiency, portability, and error detection and diagnosis facilities.

Parallel Pascal primitive operators are based on the instructions available on Parallel Matrix Processors (PMP's), a class of highly structured parallel processors involving a large number of PE's with a limited PE interconnection scheme. Two examples of PMP's are the MPP [2] and BASE [3]. It should be efficiently implementable on a very wide range of architectures, including vector and pipeline processors. However, the cost for this portability is that many powerful features of particular parallel processors may not be made easily available as operators. As a result, it may be necessary to perform some simple reformulation of algorithms to achieve optimum efficiency when transporting programs.

Parallel Pascal is not simply implementable on an MIMD processor. However, a program written in Parallel Pascal can be divided more easily into subtasks than an equivalent conventional Pascal program.

The prime objective of developing a parallel processor is to achieve high speed execution; therefore an efficient programming of any problem is essential. The extensive error checking available with conventional Pascal impairs the efficiency of the program execution unless the parallel processor contains error checking hardware (which is usually not the case). An implementation of Parallel Pascal should provide for the generation of code without runtime error checking once programs have been debugged. In addition, an implementation should provide for the inclusion of assembly language segments for critical sections, most likely as externally called procedures.

A translator program has been written for a subset of Parallel Pascal which will translate a Parallel Pascal program into a conventional Pascal program; the translator itself is written in Pascal. The design of Parallel Pascal is being refined as experience is gained with writing practical Parallel Pascal programs and running them via the translator. A Parallel Pascal compiler for the MPP is being developed.

The two principal goals of the Pascal programming language were to make available a language for teaching systematic, structured programming and to develop reliable, efficient implementations on presently available computers [4]. The resulting language is based on Algol 60 and has a richer set of program control structures and data structures (types).

The goal of implementability was achieved by considering how to simply compile the language when it was designed. The structure of the language was chosen so that a simple parsing algorithm could be used [5]. Unfortunately, the goal of simplicity has led to a few deficiencies which should be remedied in future language revisions. One serious deficiency for Parallel Pascal is the lack of dynamic arrays -- array dimensions may only be specified by constants. This provides simplicity and strict typing, but makes it very difficult to write a library of functions for general array operations.

A special version of Pascal with operating system features, called Concurrent Pascal [6] has already been developed by Per Brinch Hansen. In a sense, Concurrent Pascal reduces the semantic gap between a user Pascal program and the total computer environment including the supervisor mode and operating system. In Parallel Pascal an attempt is made to reduce the semantic gap between the Pascal language and parallel processor architectures.

In Parallel Pascal a set of standard functions

The work reported here was funded by NASA-Goddard Space Flight Center under grant NAG 5-3.

for general array manipulations will be introduced. All standard functions will be defined for any size arrays; this is consistent with the Pascal concept of standard functions operating on more than one data type. User defined procedures and functions will be limited to a single array size.

Parallel Pascal is characterized by the following extensions to Pascal:

(a) Arrays to be manipulated by the parallel processor may be explicitly declared as such by the word parallel, e.g.

```
a,b,c:  parallel array [1..8,1..8] of type
```

(b) Expressions may involve entire arrays; also, functions may return entire arrays, e.g.

```
a := b + sin(c) + 3
```

means

$$a[i,j] := b[i,j] + \sin(c[i,j]) + 3 \quad \forall\ i,j$$

(c) All control statements may have arrays for control variables, e.g.

```
if A>B then C := 3
```

means

$$\text{if } A[i,j]>B[i,j] \text{ then } C[i,j] := 3 \quad \forall\ i,j$$

(d) A new set of standard functions are available for entire array manipulation. These functions are defined for all array sizes and types.

```
shift(array, S1, S2, ..., Sn)
rotate(array, S1, S2, ..., Sn)
```

The shift function moves the data in the amounts specified by the integers S1 ... Sn (one S for each dimension of the array). Null values are inserted at the edges of the array. The rotate function is similar to shift except that the data shifted in at one edge of the array is the data shifted out of the opposite edge of the array.

```
expand(array, dimension, size)
```

The expand function replicates the array along a new dimension size times.

```
transpose(array, D1, D2)
```

This transposes an array about the two given dimensions D1 and D2. If only one dimension is specified then the data is "flipped" about that dimension.

There are also several functions which apply a reduction operator over all of the specified dimensions.

```
general format:  fn(array,D1,D2,...,Dn)
```

asum	arithmetic summation
aprod	product
aand	logical and
aor	logical or
amax	maximum value
amin	minimum value

For example, the sum of all elements in a matrix M is specified by asum(M,1,2) and a vector containing the maximum values of each row of M is specified by amax(M,2).

(e) For convenient input and output of parallel array data the procedures read and write have been extended so that a whole array may be read on written. The capability of reading or writing a subarray of a large array file may be added later.

(f) The index for a Parallel Pascal array may be scalar, elided, a logical vector or a set. A scalar index selects one item in a dimension and reduces the rank of the result by one. An elided index specifies all items in that dimension. A subset of items in a dimension may be specified by either a set or a logical vector. The logical vector must be the same length as the dimension it indexes.

Parallel Pascal also has a bit indexing mechanism for the low level programming of bit-serial parallel processors. This mechanism is outside the normal usage of the language; however, its availability may make it possible to avoid using assembly code for low level bit serial operations. This feature is, in general, not portable between different implementations as the bit representation of numbers is machine dependent.

References

1. Reeves, A. P., Bruner, J., and Poret, M., "The Programming Language Parallel Pascal", Internal Purdue Electrical Engineering report, 1980.

2. Batcher, K., "MPP -- A Massively Parallel Processor," Proceedings of the 1979 International Conference on Parallel Processing, 1979, p. 249.

3. Reeves, A. P., "A Systematically Designed Binary Array Processor," IEEE Transactions on Computers, April 1980.

4. Jensen, K. and Wirth, N., "Pascal User's Manual and Report," Springer Verlag, New York, 1974, p. 133.

5. Wirth, N., "The Design of a Pascal Compiler," Software-Practice and Experience, Vol. 1, 1971, p. 320.

6. Brinch Hansen, P., "The Programming Language Concurrent Pascal", IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, June 1975, pp. 199-207.

Decomposing a Program for Multiple Processor Systems

Arvind

Massachusetts Institute of Technology
Laboratory for Computer Science
545 Technology Square
Cambridge, Mass. 02139

Abstract

The success of high performance multiple processor systems depends upon our ability to decompose a program into small segments suitable for execution on one processor. It is argued in this paper that purely applicative languages are better suited for parallel processing because they offer considerable advantage over Fortran-like languages in program transformation and decomposition. A scheme for decomposing applicative programs is described through examples.

1 Introduction:

The operation of a multiple processor system designed to increase the execution speed of a single program can be viewed at two levels. At the *macroscopic* level the system carries out the computations of the user's high-level program. At the *microscopic* level each processor executes its own set of instructions and exchanges data with other processors as needed. Implementing a program on such a system requires transforming the high level description into a set of programs for the individual processors.

Work at the University of California, Irvine has shown how high-level dataflow programs can be mapped onto a set of asynchronously cooperating processors as the computation unfolds dynamically [4,10]. For applications such as partial differential equation simulation, however, the cost and overhead of fully general, dynamic mapping may be unwarranted. These applications are characterized by extremely high computational requirements and simple and regular program and data structures [3]. Hence a static mapping of activities onto processors may prove more efficient and cost effective without creating an undue loss of flexibility. Furthermore, a static mapping scheme for these problems could distribute activities and data structure elements over the processors in such a way that information flow is highly localized. This would allow a simpler, lower cost interconnection network than is required to achieve high performance with dynamic mapping. The effectiveness of static mapping of activities is directly related to the decomposability of a program.

In this paper we first discuss the appropriateness of purely applicative languages for parallel processing and then give a scheme for decomposing applicative programs for multiple processor systems. It is assumed that each processor is capable of storing its own program and data and can communicate with any other processor in the system. The internal organization of a processor does not affect our scheme.

This research was supported by NSF Grant no. MCS-7902782

2 Impact of High level language on Decomposability:

Traditionally, Fortran and its extensions have been regarded as the only acceptable high-level languages for high performance systems such as CRAY-1, STAR-100 and Illiac IV. The main reason for programming in Fortran is to maintain compatibility with a large existing body of scientific software. This compatibility is of little use in practice because existing Fortran programs do not show significant performance gains on new machines with different architectures. In fact, a recoding of parts of the program either in machine language or in some new extension of Fortran is required to achieve high performance. Software tools such as vectorizing and optimizing compilers have been successful on a very limited class of Fortran programs, namely those programs that do not have undesirable "side-effects" (see [1] for an in-depth discussion of side-effects). In a maximally parallel program, statement executions are ordered only by data dependencies. It is difficult to detect data dependencies in a Fortran (in fact in any imperative language) program due to accesses to global variables and operations on data structures. A programmer often tries to minimize storage by reusing the same array over and over again. This further complicates detection of minimal data dependencies. A ban on the use of global variables or arrays seems absurd in view of the fact that for efficiency, a clever Fortran programmer often passes parameters to subroutines through *common* declaration. Potential side-effects of *common* declarations in Fortran are so intricate that most optimizing compilers will not optimize across subroutines.

Kuck and his associates [12, 13, 14] have studied and classified a large number of Fortran programs in an attempt to identify features that should be supported by high performance systems. They have given various statement execution orderings that potentially could be exploited by a compiler for a multiple processor machine or an array or pipeline machine. Various transformations of the source program are also suggested to enhance parallelism in a program. Even though the usefulness of studying existing algorithms[1] for designing high performance architectures is undeniable, we take issue with Kuck's acceptance of the adequacy of Fortran or its extensions. An efficient multiple processor architecture cannot be developed unless it supports the execution model of a parallel processing language systematically.

Our approach to program decomposition may be applicable to a class of Fortran programs structured along certain guidelines. However the basis of our programming is so radically different from Fortran that syntactic compatibility with Fortran is of little use. If a Fortran program has to be rewritten for a high performance architecture then it can just as well be written in a new language. We hope that eventually parallel processing languages will remove the constraints placed by Fortran-like languages on our thinking, encouraging us to develop yet faster algorithms.

3 Applicative Programming for Parallel Processing:

A parallel or asynchronous programming language for a multiple processor system should not incorporate the concept of an updatable storage cell [4, 8]. This is essential to avoid complex synchronization mechanisms and elaborate sequencing of operations. When all computation is based on values, as opposed to addresses where values are kept, the possibility of a race to read or write is not possible. The two most widely known languages that can support pure applicative (i.e., functional) programming are LISP and APL. However, both have such different syntax from conventional languages that the effort involved in learning either of them is quite substantial. The difficulty is further compounded by the fact that both LISP and APL also present entirely different programming paradigm. One has to almost unlearn Fortran programming to be able to think clearly in either of these languages. LISP due to its recursive nature and strange syntax is treated by scientific programmers as an amusing diversion for academicians. The inefficiency of these languages on conventional architectures also lends support to Fortran adherents.

We think that the "syntax" problem of applicative languages is completely solvable. Two languages, Id[4] and VAL[2], currently under development at the University of California at Irvine and MIT provide a syntax as well as a programming paradigm that is superficially quite similar to Algol–Pascal family of languages. Both of these languages are purely applicative, and we believe that a programmer familiar with Algol can learn Id in a few days.

Generally, an applicative language such as LISP allows the creation and use of data structures in a much more dynamic manner than Fortran. Hence a fair comparison of their efficiency is difficult. However, for most numerical algorithms this expressive power of applicative languages is not required. An applicative language with as restrictive a control and data structure as Fortran may still be less efficient than Fortran on a sequential computer. However, for a multiple processor machine the efficiency of a high level language will depend on the availability of program decomposition schemes, and due to this fact applicative languages may indeed turn out to be more efficient than imperative languages for such machines. A consensus seems to be emerging on this point [6,9,11].

[1] We prefer to study algorithms over programs because algorithms are more language independent.

The problem of decomposition can also be viewed as an exercise in program transformation. A fair amount of work has already been done on transforming applicative programs (see [7] for example). We illustrate the flexibility for decomposition provided by an applicative program through an example. Consider a classical relaxation algorithm in one-dimension. One computes the new values of the x elements repeatedly using the following equation.

$$new\,x_i = (x_{i-1} + x_i + x_{i+1})/3. \quad 1 \leq i \leq n$$

where x_0 and x_{n+1} remain constant.

A straightforward Fortran program would do this in the following way.

```
C  X IS AN ARRAY OF N+2 ELEMENTS
C  X(1) AND X(N+2) REMAIN CONSTANT
   N1=N+1
   DO 20 K=1, KMAX
   DO 10 I=2, N1
   Y(I) = (X(I-1) + X(I) + X(I+1))/3.
10 CONTINUE
   DO 15 I=2, N1
   X(I) = Y(I)
15 CONTINUE
20 CONTINUE                              (1)
```

A compiler can easily generate good code for a multiple processor machine from the above program. Even if a programmer is clever, and avoids copying array Y into X by switching back and forth between X and Y, a vectorizing compiler will be able to deal with it effectively. However, if array X is large, and a programmer decides to avoid using another array Y altogether, the following program may result.

```
   N1=N+1
   DO 20 K=1, KMAX
   T1=X(1)
   T2=X(2)
   DO 10 I=2, N1
   X(I)=(T1+T2 + X(I+1))/3.
   T1=T2
   T2=X(I+1)
10 CONTINUE
20 CONTINUE                              (2)
```

It would be extremely difficult for a compiler to detect a transformation in which all the elements of array X are relaxed simultaneously.

On parallel computers, programmers use the trick of relaxing only half the elements (i.e., odd or even) in one iteration to avoid excessive use of storage. It should be noted that the algorithm for relaxing odd and even elements alternatively is an entirely different one, and requires mathematical sophistication on the part of a programmer to prove its stability.

Now we contrast this situation with an applicative program written in Id.

```
(for k from 1 to kmax do
  new x ← (initial y ← <0:lb , n+1:rb>
                 ! lb and rb represent the boundary values at
                 selectors 0 and n+1 respectively!
           for i from 1 to n do
                 new y[i] ← (x[i-1] + x[i] + x[i+1])/3.
           return y)
 return x)                                                   (3)
```

We rely on readers intuition to understand the control structure of the above Id program. Manipulation of arrays (i.e. an example of structures) in applicative languages needs some explanation. One thinks of every array construction operation (such as *append*) as producing a new array. Hence *append* (a,i,v) produces a new array a' which differs from a only in position i. Even though *new* y[i] ←... looks like a conventional assignment statement, y[i] does not refer to a storage cell. Rather one should think of the whole array as a value, and y[i] as referring to a part of the value. Naturally if one changes a part of a value the aggregate value changes too. In this example since *i is taken from an unrepeated set of values* (i.e., 1 to n) it is possible to regard y as an I-structure [5]. In contrast to ordinary structures, an element of an I-structure can be used as soon as it is created. Thus I-structures allow greater freedom in manipulating programs for efficient execution on a parallel computer.

Using a vectorizing compiler it is as easy to generate code for a multiple processor machine from this Id program as it was with the first Fortran program. However, the same Id program allows us to generate code that may overlap several iterations of the outer loop. Note that since y is an I-structure, the k+1st iteration of the outer loop can begin as soon as the first three elements of x from the kth iteration have been computed. If we desire we can easily derive implementations of this Id program that will use the same minimum amount of storage as the second Fortran program and still allow concurrent execution of several iterations (see Figure 1).

Our premise is that *a high level language should permit coding of algorithms to show the maximal parallelism inherent in an algorithm.* Such languages have to be purely functional in nature. The task of decomposing and transforming maximally parallel programs for a parallel machine is considerably simpler than the task of decomposing Fortran programs. In the rest of this paper we will outline a scheme for decomposing Id programs for multiple processor machines. The scheme will be described through examples.

4 Decomposition Scheme:

Applicative programs that have loops as their primary control structure and that operate on bounded-size data structures can be decomposed into programs for a set of individual processors in three steps:

1. The nested loop structures are unrolled into a network of *computation cells*.
2. Data structure elements are assigned to the *cells*.
3. The network of *computation cells* is mapped onto the actual processors of the system, according to the size and structure of both the network and the computer system.

A *computation cell* can be regarded as a virtual processor to which a program and local data has been assigned. However, the virtual processor program may also refer to data that is not local, in which case a communication between this virtual processor and the virtual processor holding the data takes place. We will use programs written in Id language to illustrate the decomposition scheme. All expressions in Id have the property that for every set of inputs received they must produce exactly one set of outputs. Due to this property, the communication between *computation cells* is highly structured and its pattern can be determined *a priori*. In order to remain consistent with the data-driven nature of Id, we assume, without loss of generality, that a non-local value is sent to, rather than demanded by, a *computation cell*. We can draw a directed link from the cell that sends a value to the cell that receives it, and thus a network of virtual processors can be created. If an unbounded number of processors were available and if these processors could be interconnected in any desired pattern, then an ideal network topology for the physical system would be the topology of the *computation cell* network.

4.1 Defining Cells of Computation:

A programmer defines a *cell* by specifying what task is to be carried out by it. For example a task may be defined as the work done in the ith iteration of a loop, hence by unfolding a loop a number of computation cells may be defined. A program for the task carried out in the ith iteration of an Id loop can be generated automatically. There is in general more than one computation cell definition possible as we show below. Consider the following program for conventional matrix multiply algorithm.

```
procedure matrix_multiply (a, b, l, m, n)
 ! multiply matrix a of dimensions l × m by matrix b
   of dimensions m × n!
 (initial c← < > ! < > represents an empty structure!
  for i from 1 to l do
      new c[i] ← (initial d← < >
                  for j from 1 to n do
                         new d[j] ←
                               (initial s←0
                                for k from 1 to m do
                                    new s ← s + a[i,k]*b[k,j]
                                return s)
                  return d)
  return c)                                                  (4)
```

In Id a matrix is represented by an one-dimension array of one-dimension arrays. Hence output matrix c is constructed by appending together l rows, each represented by an array d. We note that this Id program when executed under the U-interpreter [5] can automatically carry out all the l × m × n multiplications in parallel. This effect is achieved without any global analysis of the program. As stated earlier the attempt in this paper is to perform certain functions of the U-interpreter at compile time and hence map concurrent activities statically onto processors.

Suppose we specify computation cells to carry out one iteration of the loop with index variable i. The network of Figure 2 will be produced by unfolding loop i. The program for the i^{th} cell can be written as follows.

```
append(c_{i-1}, i, d) where d is computed as follows
        d ← (initial d ← < >
              for j from 1 to n do
                new d[j] ← (initial s ← 0
                              for k from 1 to m do
                                new s ← s + a[i,k] * b[k,j]
                              return s)
              return d)
```

The subscript on a variable (i.e. c_{i-1}) indicates the cell where that variable will be computed. This program is valid for cell numbers 1 to l. The computation cell 0 should produce an empty array < >, and the result should be available in cell l+1.

The definition of *computation cells* can be critical to exploiting parallelism in a program. In the matrix multiply procedure if l is much smaller than the actual number of processors available then unfolding either both loop i and loop j, or only loop j may be more advantageous than unfolding only loop i. The network produced as a result of unfolding loop i and loop j of program (4) is shown in Figure 3.

There is no obvious advantage in unfolding the outer loop of program (3) for the relaxation algorithm. Computation cells produced in this manner will execute essentially in a sequential order. However, if loop i is unfolded concurrent relaxation of all the elements of x is possible. The process of unfolding an inner loop without unfolding the outer loop in a nested loop structure is somewhat tricky. The result of unfolding loop i of program (3) is shown in Figure 4. The cell programs are given below.

for cell 0

```
    y_0 ← <0:lb , n+1:rb>
```

for cell $1 \leq i \leq n$

```
    y_i ← append(y_{i-1}, i, t) where t is computed as follows
    t ← (x_{n+1}[i-1] + x_{n+1}[i] + x_{n+1}[i+1])/3.
```

Note that as before x_{n+1} means that x is defined in cell n+1. The program for cell n+1 is

```
  (for k from 1 to kmax do
       new x ← y_n(x)
   return n)
```

The meaning of $y_n(x)$ is that output from cell n is needed but it can only be obtained after x is supplied. In a dataflow interpretation the value array x is sent by cell n+1 to all the relevant cells as soon as x is produced. For every x value that cell n+1 outputs it receives an input value y_n which becomes the new value of x. The initial value of x has to be given to cell n+1 to start the computation. Such an input is implicit in program (3).

A reader at this point may consider the *computation cell* network of Figure 4 to be very wasteful because it makes many copies of array x. Indeed this is what we wish to avoid by mapping structures on computation cells.

4.2 Mapping Data Structures on Computation Cells:

Once *computation cells* have been defined, a mapping of data structures (i.e., matrices and arrays) onto these cells can be specified. For example a programmer may specify that x[i,j] for all j should be mapped onto cell k. Mappings may not be one to one. Consider again program (3) with the inner loop unfolded. A mapping that seems quite sensible is that elements x[i-1], x[i] and x[i+1] be mapped on to cell i. This mapping assigns each element of x to three *computation cells.* Treating a data structure as collection of elements--each one of which is assigned to a *cell* or *cells*--eliminates the *select* operation on data structures. Suitable mappings to reduce communication due to the *select* operation are straightforward to derive but unfortunately solve only half the problem.

Consider a mapping in which each x[i] is mapped onto only cell i. The program for cell i can be expressed as follows:

```
y_i ← append(y_{i-1}, i, t) where t is computed as follows
t ← (x[i-1]_{i-1} + x[i] + x[i+1]_{i+1})/3.
```

where each x[j] should be treated as one value and the meaning of $x[j]_k$ is as usual that x[j] is defined in cell k.

The value t computed by cell i becomes part of array y, and is passed on to cell i+1 which passes it to cell i+2 and so on. It finally reaches cell n+1 as a part of the value y_n. The new value of x is y_n, and it is y_n that is distributed to cells 1 to n. Hence *the x[i] that cell i receives is in fact the last t computed by cell i.* This makes the whole process of constructing x and then distributing it seem unnecessary. Every cell should compute t and store it for the next calculation of t. It must still communicate the value of t to cells i-1 and i+1 in order for these calls to compute their t values but most of the communication from cell n+1 to cell i will be avoided. There has to be some communication from cell n+1 to cell i to indicate if the computation has terminated or not (i.e., k>kmax?). Figures 5.1 and 5.2 depict the effect of simplification achieved by this data structure mapping.

In order to achieve the simplification suggested by Figure 5.2 we have to be able to determine the *cell* where a particular element of a structure is generated. This can be done easily in program (3) once we note that no element of y in the inner loop is ever redefined that is, y is an I-structure. As noted earlier an element belonging to an I-structure can be distributed as soon as it is generated.

For the kind of programs we are interested in, the selector for the append operation is often directly and simply related to the loop index. If the loop index is taken from an unrepeated set the condition of I-structures is automatically met. Now we give a rule for determining the number of the *cell* where the element of a data structure is generated in such cases.

Suppose we want to find the cell number where element c[i,j] is defined in program (4). First, find the cell that appends a value on selector i of c. Let k0 be the cell. Then

$c_{k0} \leftarrow \mathit{append}(c_{k1}, i, d_{k2})$

where subscripts k0, k1 and k2 refer to cell numbers. Once k2 is known find the cell that appends a value on selector j of d_{k2}. Let k3 be such a cell. Then

$d_{k3} \leftarrow \mathit{append}(d_{k4}, j, v)$

Mapping c[i,j] onto cell k will mean that cell k3 will send value v to cell k. Suppose we consider the cell definitions of Figure 3. Then mapping c[i,j] onto cell number <i,j> results in all the *append* operations being eliminated. Cell <i,j> would compute a value s and hold on to it. On the other hand, mapping c[i,j] on cell <i,j+1> would result in cell <i,j> sending the value s to cell <i,j+1>.

It is useful in a large program to map a data structure according to how it is used rather than how it is created. When matrix multiply is part of a larger program one will have to take into account the cells where matrix c will be used to specify efficient mappings. A common situation is that of unnested loops where one loop produces a structure and the other loop uses it. In such cases a cell definition may include one iteration of each loop.

5 Mapping Computation cells on processors:

In general one expects the network of *computation cells* to be larger than the number of processors available. The mapping in such cases takes the form of specifying a *folding* of the network of cells to fit the machine. Suppose we want to map the network of Figure 2 onto a p processor machine when l >> p. For the interconnections of Figure 2 we consider 3 mappings:

1. Map cell i on processor number i mod p (see Figure 6.1).

2. Map cells 0 to p-1 on processors 1 to p. Map cells p to 2p-1 on processors p to 1, and so on (see Figure 6.2).

3. Map cells 0 to f-1 on processor number 1, cells f to 2f-1 on processor number 2, and so on where $f = \lceil (l+2)/p \rceil$ (see Figure 6.3).

If the p processors are connected by a ring bus there may be no reason to choose between mappings 1 and 2. However the first two mappings are clearly inferior to the third mapping if the p processors have any kind of locality in their interconnection.

This small example only illustrates that a reasonable mapping of a network of cells onto an actual machine can be derived by simple reasoning. In fact we expect to do such simple folding of networks automatically.

6 Conclusions:

The success of large multiple processor machines crucially depends upon their programmability. Flexibility in programming such machines is ultimately limited by our ability to decompose a program into smaller programs suitable for execution on one processor. In the past, decomposition efforts have had limited success due to Fortran being the source language. It is suggested in this paper that applicative languages with restrictions on data and control structures are far more amenable to decomposition. It is generally quite easy to write a maximally parallel applicative program for a given algorithm. Undoubtedly the problem of decomposing a maximally parallel program is far simpler than detecting parts of a sequential programs that are suitable for concurrent execution.

A strategy for decomposing applicative programs for a multiple processor machine has been outlined. It creates a network of *computation cells* without relying on any information about the topology or the number of processors in the actual machine. The network is mapped onto the actual machine as the last step in the procedure. Our research efforts for the time being are concentrated on deriving efficient cell programs for the network of *computation cells*.

Acknowledgments

Bill Ackerman, Dean Brock, Randy Bryant and Vinod Kathail have provided invaluable help by reading several drafts of this paper.

References

1. Ackerman, W.B., "Dataflow Languages". *AFIPS NCC Proc.*, Vol. 48, (June 1979), 1087-1095.

2. Ackerman, W.B. and Dennis, J.B., "VAL -- A Value Oriented Algorithmic Language: Preliminary Reference Manual." TR-218, Laboratory for Computer Science, M.I.T. Cambridge, Mass. (1979).

3. Arvind, and Bryant, R.E., "Parallel Computers for Partial Differential Equation Simulation." Proc. Scientific Computer Information Exchange Meeting, Livermore, California, (Sept. 1979), pp. 94-102.

4. Arvind, Gostelow, K.P., and Plouffe, W., "An Asynchronous Programming Language and Computing Machine." University of California, Irvine, Technical Report 114a (Dec. 1978).

5. Arvind, and Thomas, R.E., "I-structures: An efficient data type for functional language." TM, Laboratory for Computer Science, M.I.T. Cambridge, Mass. (June 1980).

6. Backus, J., "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs." *Comm. ACM*, Vol. 21,N o. 8, (Aug. 1978), pp. 613-641.

7. Burstall, R.M. and Darlington, J., "A Transformation System for Developing Recursive Programs." *J.ACM* Vol. 24, No. 1, (January 1977)

8. Dennis, J.B., "First version of a dataflow procedure language." TM 61, Laboratory for Computer Science, M.I.T. Cambridge,Mass. (May 1975).

9. Friedman, D.P., and Wise, D.S., "The impact of applicative programming on multiprocessing." Proc. 1976 Intl. Conf. on Parallel Processing (Aug. 1976), pp. 263-272.

10. Gostelow, K.P. and Thomas, R.E., "Performance of a Dataflow Computer." *IEEE Transactions on Computers*, (to appear Oct. 1980).

11. Keller, R.M., Lindstrom, G., and Patil, S., "A Loosely-Coupled Applicative Multiprocessing System." *AFIPS NCC Proc.*, Vol. 48 (June, 1979), pp. 613-622.

12. Kuck, D., Budnik, P., Chen, S.C., Davis, E. Jr., Han, J., Kraska, P., Lawrie, D., Muraoka, F., Strebendt, R. and Towle, R., "Measurements of Parallelism in Ordinary FORTRAN Programs," *IEEE Trans. on Computer*, Vol. C-23, No. 1, (Jan., 1974), pp. 37-46.

13. Kuck, D.J., Muraoka, Y. and Chen, S.C., "On the Number of Operations Simultaneously Executable in FORTRAN-Like Programs and Their Resulting Speed-Up." *IEEE Trans. on Computer*, Vol. C-21, No. 12,(Dec., 1972), pp. 1293-1310.

14. Kuck, D.J. and Padua, D.A., "High Speed Multiprocessors and their Compilers." Proc. of International Conference on Parallel Processing. (August 1979), pp. 5-16.

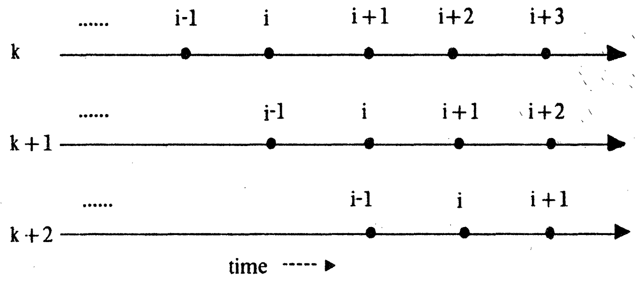


Figure 1 Overlapping iterations in program (3)

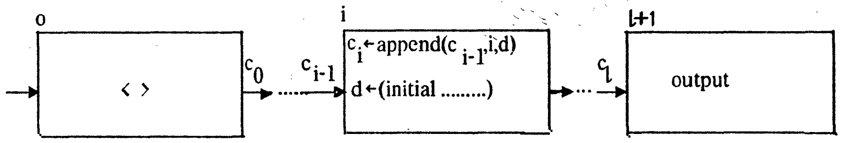
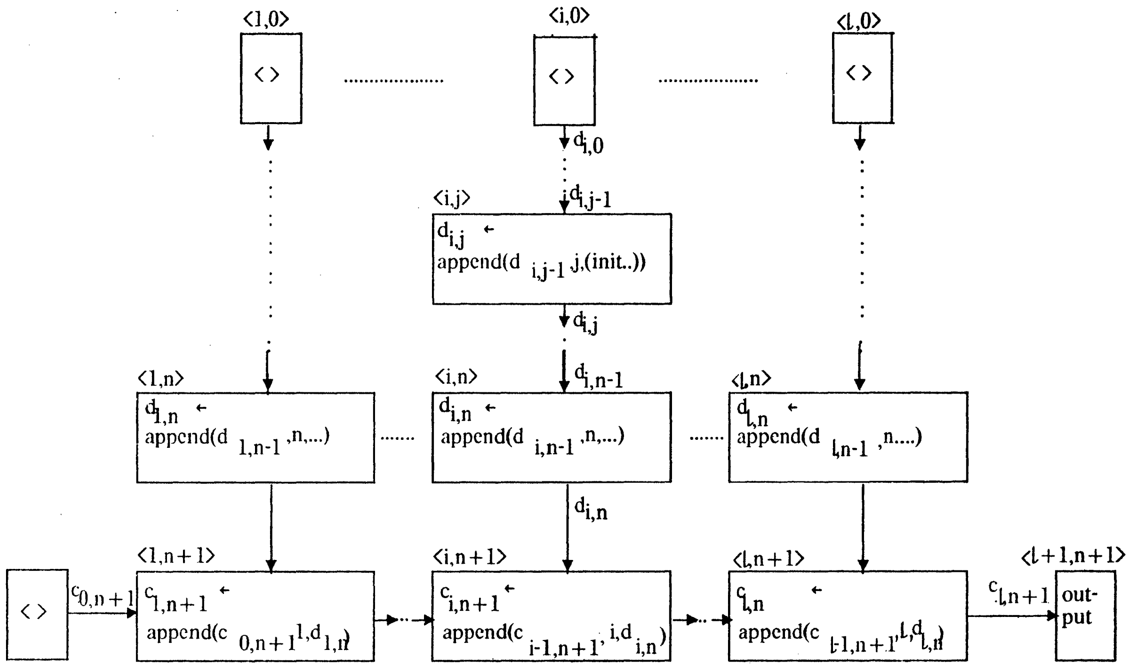
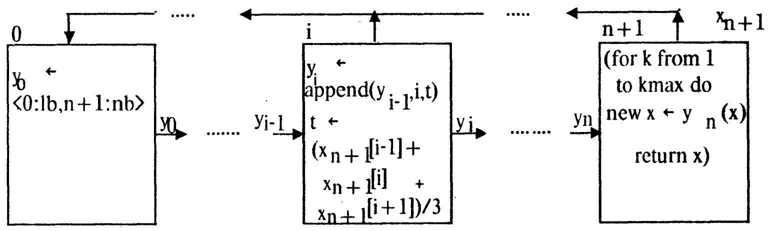

o
i
l+1
< >
c_0
c_{i-1}
$c_i \leftarrow append(c_{i-1}, i, d)$
$d \leftarrow (initial ....)$
c_l
output

Figure 2 Computation cell network when loop i is unfolded in the matrix multiply procedure



<1,0>
<i,0>
<L,0>
< >
$d_{i,0}$
$d_{i,j-1}$
<i,j>
$d_{i,j} \leftarrow$ append($d_{i,j-1}$, j, (init..))
$d_{i,j}$
$d_{i,n-1}$
<1,n>
$d_{1,n} \leftarrow$ append($d_{1,n-1}$, n,...)
<i,n>
$d_{i,n} \leftarrow$ append($d_{i,n-1}$, n,...)
<L,n>
$d_{L,n} \leftarrow$ append($d_{L,n-1}$, n....)
$d_{i,n}$
<1,n+1>
<i,n+1>
<L,n+1>
<L+1,n+1>
$c_{0,n+1}$
$c_{1,n+1} \leftarrow$ append($c_{0,n+1}$, 1, $d_{1,n}$)
$c_{i,n+1} \leftarrow$ append($c_{i-1,n+1}$, i, $d_{i,n}$)
$c_{L,n} \leftarrow$ append($c_{L-1,n+1}$, L, $d_{L,n}$)
$c_{L,n+1}$
out-put

Figure 3 Computation cell network when loop-i and loop-j are unfolded in the matrix multiply procedure



0
$y_0 \leftarrow$ <0:lb,n+1:nb>
y_0
y_{i-1}
i
$y_i \leftarrow$ append(y_{i-1}, i, t)
$t \leftarrow$ ($x_{n+1}[i-1]$ + $x_{n+1}[i]$ + $x_{n+1}[i+1]$)/3
y_i
n+1
x_{n+1}
y_n
(for k from 1 to kmax do new x ← y_n(x) return x)

Figure 4 Computation cell network when inner loop-i is unfolded in the relaxation program(3)

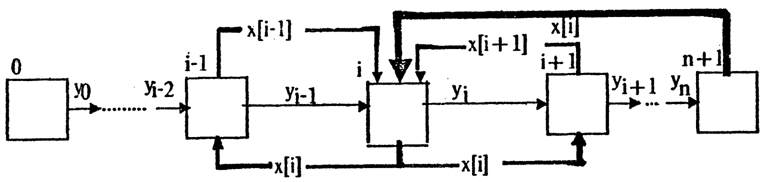


Figure 5.1 Mapping x[i] on cell i by distributing x in the network of figure 4. (—— show reference pattern, ▬ show mapping)

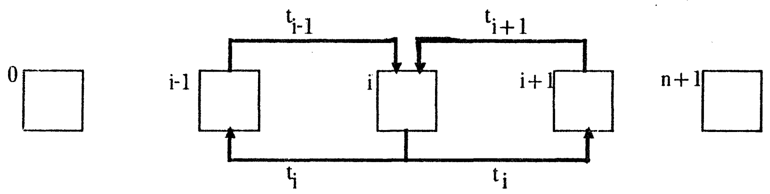


Figure 5.2 Mapping x[i] on cell i by sending it from the cell that generates x[i].Since cell i generates x[i] no dashed line with value x[i] is shown.

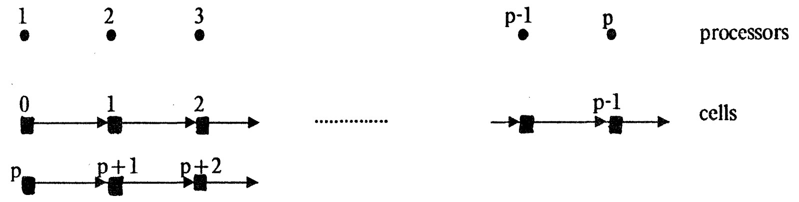


Figure 6.1 Mapping (1) - cell i on processor i mod p

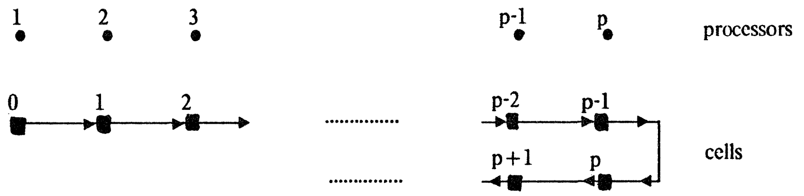


Figure 6.2 Mapping(2) cells 0 to p-1 on processors 1 to p, cells p to 2p-1 on processors p to 1 etc.

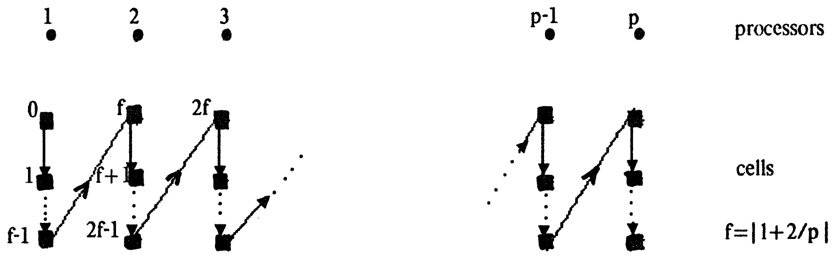


Figure 6.3 Mapping (3) cells 0 to f-1 on processor 1, cells f to 2f-1 on processor 2 etc.

Automatic Exploitation of Parallelism on a Homogenous Asynchronous Multiprocessor

Thomas L. Rodeheffer and Peter G. Hibbard
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania, 15213

Summary

This paper describes an investigation which is starting into the practical issues of automatically detecting parallelism in ordinary programs and exploiting it on a multiprocessor. We are looking at user programs written in Fortran and our target multiprocessor is Cm*, a distributed multiprocessor designed and built at Carnegie-Mellon University.

We have chosen Fortran because of the following reasons: we have a modern Fortran compiler written in C which is accessible and easy to modify; Fortran has a simpler implementation than other commonly-used high-level languages; much previous work has been concerned with analyzing Fortran programs, thereby allowing our results to be more easily compared with the results of others; and finally, Fortran is a language of much practical interest to the scientific computing community.

We have chosen Cm* [1] as our target multiprocessor primarily because it is a part of our research environment. Since Cm* is the subject of several projects, our work can enhance other research. Operating roughly as a classic, shared-memory multiprocessor with fifty identical, asynchronous processors, Cm* has the advantage that its memory accessing mechanism is implemented by a hierarchical switching network whose nodes can be microprogrammed to provide special operations in addition to simple memory mapping. Finally, Medusa [2], an operating system which supports a Unix-like environment but still allows almost a full exploitation of the Cm* hardware, has recently become available.

Measurements on Cm* [3] indicate that speedups near the theoretical limit are attainable for programs which have been carefully designed to take advantage of the available parallelism. Unfortunately, sufficiently careful and ingenious design has not proved to be a simple matter. Programming a multiprocessor is a difficult and tedious task, especially at the detailed levels of inter-processor coordination. Furthermore, multiprocessors are not generally available and thus much work has tended to be theoretical in nature.

Previous work on automatically detecting and exploiting parallelism has been directed primarily at architectures other than those of asynchronous multiprocessors. Kuck *et al.* [4] have studied extensively the ways in which programs can be transformed to extract parallelism under the assumption that the target architecture consisted of synchronous processors which perform exactly one operation every time step. Allan and Oldehoeft [5] have considered the same problems with a data-flow machine as a target architecture. In both cases, no consideration need be given to the problems of communication and synchronization between the processing elements, because such problems are assumed to be solved by the architecture at no cost. Gonzalez and Ramamoorthy [6] have studied through simulation the problems of scheduling on a multiprocessor parallel tasks of a program at the statement level.

We view the exploitation of parallelism as an optimization technique which is useful on a multiprocessor architecture. We are interested in automatically taking advantage of low-level parallelism—parallelism which would be difficult or just too tedious for a programmer to specify but which can be detected on a fairly local basis. The more global problem of designing a program or algorithm specifically to use parallelism falls beyond the scope of what we consider automatic optimization techniques.

We are building a prototype system which compiles Fortran programs into machine code for Cm*, detecting implicit, low-level parallelism and generating a schedule of tasks to minimize the time-to-completion of the program. All detection and scheduling is done during compilation. The run-time system on Cm* provides inter-processor communication and synchronization primitives which the compiler uses to effect its schedule.

For our purposes, each of the individual processors in Cm* contains a copy of the same code and shares access to the same data locations. As explained in [2], such an arrangement is not the most effective use of Cm*, but its simplicity and similarity to the normal manner of use of tightly-coupled multiprocessors is appealing. This arrangement is essentially the same as presumed in [6].

The compiler processes the Fortran source program on a subroutine-by-subroutine basis. Each subroutine is compiled into a directed graph of actions, in which each action represents an operation at the level of the individual operations of expression evaluation, and each edge represents a data- or control-flow dependency. The compiler then analyzes the flow graph to determine an execution schedule for the actions of the program.

The compiler uses approximate execution times for the various machine instructions and run-time system primitives in order to transform the flow graph to reduce the estimated time-to-completion of the final object code program. For example, a sequence of actions each of which is dependent solely upon its predecessor is probably best executed as a single task with no internal scheduling actions. Even actions that could be performed in parallel probably ought to be executed sequentially without scheduling if the overheads required of the run-time system to coordinate another processor are too large relative to the time that could be saved by parallel execution. These are only the simplest transformations, however.

Kuck *et al.* [4] have developed transformations applicable to assignment statements and common forms of Fortran DO-loops which exploit parallelism to reduce total execution time. Although the transformations were designed for a synchronous multiprocessor architecture such as an array machine, with proper consideration of inter-processor coordination costs it seems that these transformations could be useful in the environment of an asynchronous multiprocessor as well.

Another important class of transformations are those which act to defer or distribute overheads so that work is removed from the critical, limiting path of the computation. For example, instead of creating a new task at some point in the program (which involves the run-time overhead of locating a free processor and communicating the task start address to it) the compiler may be able to identify an earlier task whose completion had recently been awaited and arrange to re-use that task by passing signals for synchronization; this assumes that the task creation and completion primitives are more expensive than a signal between two existing tasks.

Our goal is to demonstrate a workable system for exploiting low-level parallelism on a multiprocessor. We are encouraged by previous results [7] which indicate that substantial low-level data parallelism is in fact available, although in that implementation the language run-time support was so complex that performing all analysis at run-time was feasible. Now we direct our attention to a language of much simpler requirements in order to address the practical issues of a workable system.

References

[1] R. J. Swan, S. H. Fuller, and D. P. Siewiorek, "Cm*—A Modular, Multi-Microprocessor", *AFIPS Conference Proceedings, 1977 National Computer Conference*, AFIPS Press, Montvale, New Jersey, (1977), pp. 637-644.

[2] J. K. Ousterhout, *Partitioning and Cooperation in a Distributed Multiprocessor Operating System: Medusa*, PhD dissertation, Computer Science Department, Carnegie-Mellon University, (Apr. 1980). Available as CMU Tech. Report CMU-CS-80-112.

[3] L. Raskin, *Performance Evaluation of Multiple Processor Systems*, PhD dissertation, Carnegie-Mellon University, (Aug. 1978). Available as CMU Tech. Report CMU-CS-78-141.

[4] U. Banerjee, S. C. Chen, D. J. Kuck, and R. A. Towle, "Time and Parallel Processor Bounds for Fortran-Like Loops", *IEEE Transactions on Computers*, Vol. C-28, No. 9, (Sept. 1979), pp. 660-670.

[5] S. J. Allan and A. E. Oldehoeft, "A Flow Analysis Procedure for the Translation of High Level Languages to a Data Flow Language", *Proceedings of the 1979 International Conference on Parallal Processing*, Oscar N. Garcia, ed., IEEE Computer Society, Long Beach, California, (1979), pp. 26-34.

[6] M. J. Gonzalez, Jr., and C. V. Ramamoorthy, "Parallel Task Execution in a Decentralized System", *IEEE Transactions on Computers*, Vol. C-21, Nop. 12, (Dec. 1972), pp. 1310-1322.

[7] P. G. Hibbard, A. L. Hisgen, and T. L. Rodeheffer, "A Language Implementation Design for a Multiprocessor Computer System", *Proceedings of the 5th Annual Symposium on Computer Architecture*, IEEE Computer Society and the Association for Computing Machinery, (1978), pp. 66-72.

SESSION 2: ARCHITECTURE

A CONTROLLABLE MIMD ARCHITECTURE[a]

by
Stephen F. Lundstrom
George H. Barnes
Burroughs Corporation
Paoli, PA 19301

Abstract -- A MIMD architecture targeted at 1000 Mflop/sec has been described to NASA. This system is targeted to be the Flow Model Processor (FMP) in the Numerical Aerodynamic Simulator. This paper describes the strategies adopted for making a many-processor multiprocessor controllable and efficient, primarily by decisions that are made at compile time. Hardware features include the division of memory into space private to each processor and space shared by all, and a hardware synchronization of all processors. The connection network, connecting 512 processors to 521 memory modules, is an essential element.

Two main constructs are needed in the language to control the architecture. First, an expression that a number of instances of a given section of code can be executed concurrently, and second, a determination as to whether variables are local to the instance or global to the entire program.

Performance validations used whole programs, not kernels. Simulation and analysis combine to demonstrate achievement of the goal of 1000 Mflop/sec on suitable programs and good performance on others.

Introduction

Present generation very-high-speed computers generally exploit vector algorithms for their highest performance. A study for NASA Ames Research Center was conducted to determine the feasibility of a "Flow Model Processor" (FMP) which could achieve a sustained computational rate of one billion floating point operations per second on complete aerodynamics flow programs [1]. It concluded that the dependence on vector operations for high throughput was no longer necessary.

Given that device technology has been fully utilized, parallelism can be used to achieve performance beyond that possible with a uniprocessor. Historically, two approaches have been used to achieve parallelism: a pipeline where parallelism is achieved by each stage of the pipeline operating on a different step of successive operation, or an array of identical execution units each simultaneously evaluating the same instruction on different data. References [2,3,4, and 5] have recent examples of both. In either case the result is a vector machine where the data comes from orderly addresses in memory and the same instruction acts on each data element.

The Flow Model Processor makes use of the parallelism of a MIMD (multiple instruction stream, multiple data stream) architecture. The architecture includes specific features so that a single program can be issued to all the processors and result in cooperative execution on a single application for a single user.

This paper describes motivations behind the design and some of the strategies used to ensure controllability. The architecture described here avoids or sidesteps the limitation observed in some MIMD architectures which are unable to utilize more than a few processors effectively. The result is an architecture that is somewhat specialized to a class of applications (although much less specialized than a vector machine would be). This architecture exploits any concurrency inherent in the problem, whether or not that concurrency can be described as vector operations.

The problem was approached by first studying the aerodynamic applications [6]. These applications have a large numerical component, much inherent concurrency, and simple control structures. Due to the wide variation in the amount of computation that may concurrently proceed between times at which synchronization is required, efficient implementation of the synchronization function is required. Due to the many different natural modes of accessing data, a large memory equally accessable to all processors is required. Due to the practical limitation on the speed attainable in a large common memory, and due to the need for speed, an architecture is required which allows many memory accesses to be from memory local to each processor.

Software strategy is based on the premise that source text submitted to the compiler should

[a]This work was done for NASA under Contract NAS2-9897 and reported to them in [1].

result in a single program being compiled for all processors in the array which will then execute it cooperatively. This premise is also advocated in [7]. From another point of view, the compiler emits a single program which is to be executed independently by each of the processors in the array. Included by this program are instructions which cause the processors to cooperate by sharing data and by synchronizing their actions appropriately when needed.

A second element of the strategy is to make certain decisions at compile time instead of run time. These decisions can then be supported by efficient hardware mechanisms, not by system software.

The functional constructs on which a language for this architecture is to be based can be compared to discussions previously found in the literature. A general discussion of parallel languages is found in [8]. Some proposed parallel languages are directed at the vector type of architecture, as in [9,10,11,12], others are not [13,14,15]. Some workers have proposed that the requisite parallelism can be found by starting from algorithms expressed in serial form [16,17] so that standard Fortran can be mapped onto various parallel architectures without language extensions. In the present case the architecture is such that the operations which can be done independently of each other and in parallel are whole sections of code, not restricted to single operations.

We believe that the architecture proposed here has several advantages over other parallel architectures previously proposed and that the simulations and performance validations reported below uphold this view. While no single feature of this architecture is by itself new, we believe the combination of features is. Some previously proposed architectures have all memory shared among all processors, [18, 19, 20, 21] but without processor private memory for data. In some cases, a central control processor is involved with the control of interconnections between processors, or from processors to memory [22]. N such centralized control is required here during execution of user programs. To our knowledge, fast hardware synchronization as seen here has not been proposed for MIMD architectures, although any SIMD machine, such as in [3], will be synchronized.

The development of the system concepts evolved from the applications to system architecture (involving both hardware and software) to a more detailed definition of both the hardware and software. In order to simplify the introduction of the software concepts, they will be preceded by a short summary of the hardware architecture. Following the software concept summary, a more detailed description of some parts of the hardware will be provided.

Hardware Overview

The block diagram of the proposed multiprocessor is shown in Figure 1. Salient features of this hardware are:

* A prime number of memory modules to reduce memory access conflicts.
* Separation of the memory space seen from each processor into a private part, and a section shared among all processors.
* A connection network whereby all processors can simultaneously request access to various memory modules.
* Hardware synhcronization, a P-way AND of the signal from each processor that marks its having gotten to a specific point in the program.

Each of the 512 processors has its own program counter, its own local memory for program and data, and its own connection to a shared memory. The shared memory is built of many (521) independently accessible modules. In order to provide connectivity between the processors and the memory modules, a connection network which has a complexity of O(P log(P)), instead of the $O(P^2)$ complexity expected for a fully general cross-point network, was chosen. This choice satisfies both the economic requirements and the bandwidth requirements of the system. For discussion of the connection network, see [23].

Software

The expense involved in application software development and maintenance over the life of a system now often exceeds the total costs of operations support and acquisition/ amortization of the computational equipment especially in development environments. The development of any new capabilities for such environments must, therefore, carefully consider <u>both</u> efficient utilization of the computational facility <u>and</u> the efficiency with which application development can proceed. In the past, unfortunately, the emphasis has been almost entirely on efficient hardware utilization. The provision of capabilities to embed assembly or machine code within high-level languages such as FORTRAN are an example of this approach. One recently introduced extended FORTRAN supports both development, with application-oriented vector forms, and efficient hardware utilization [12].

The major concern during the study was the feasibility of a hardware system with the required sustained performance. Automatic conversion of standard FORTRAN was not required. Rather, the project emphasized the definition of FORTRAN extensions that provided efficient control of the hardware capabilities ease in application definition.

Language Overview

The basic language construct chosen for this

MIMD system was one of computational processes that proceed concurrently between appropriate synchronization points. This type of construct is clearly compatible with a MIMD system. Such a construct is also convenient for application descriptions in that it is more general than the vector forms currently in use. The concurrent processes may include boundary value computations and central value computations simultaneously. Thus, each program for the FMP has a structure of pieces of normal serial code, which describe the details of what must be done at a given time, or at a given element of some index set, embedded in a control structure that expresses the location of concurrency and where the synchronization must occur.

Three extensions to standard FORTRAN are proposed. The primary extension is the construct described above which allows the definition of the inherent concurrency in a process. This construct is called "DOALL". The second extension is a construct to allow the definition of index sets, called "DOMAIN"s. The third extension is a means for identifying the data or variable dependencies between the instances of various processes and for differentiating which variables or data are independent of the global process structure and are therefore local to a particular instance.

Domains

A means for describing index sets to the compiler is needed. In FMP FORTRAN such sets are called DOMAINs. A DOMAIN has an associated name and can be interpreted as a one or multi-dimensional index set. For example, the declaration

```
        DOMAIN/EYEJAY/: I=1, IMAX; J=1, JMAX
```

declares that there are IMAX*JMAX elements, each consisting of one pair of values of I and J, with values in the range shown. Standard set operators are allowed. For example, if one has also declared

```
               DOMAIN/KAY/:  K=1,KMAX
```

then the cartesian product

```
             DOMAIN/IJK/: EYEJAY .X. KAY
```

defines a three-dimensional domain with extents in each dimension of IMAX, JMAX, and KMAX.

In the aero flow applications, only rectangular domains such as the example "IJK" were seen. Extensions to the domain concept will be needed for other applications. Simple modifications to domains can be implemented by conditional statements within the doall program segment.

In addition to their use in specifying the index sets for doalls as explained in the next section, domains can substitute for the iteration index sets in do loops, and for dimensionality in the declaration of arrays.

One convenient use of the DOMAIN construct is for the description of the geometry (or computational limits) of the problem. By naming the controlling index set, and referring to the index set by name throughout the rest of the program, changes relating to geometry need be made in only one place in the program.

DOALL Construct

The DOALL construct is the FMP FORTRAN extension for describing the inherent concurrency in a process. Figure 2 shows the conceptual flow of execution in this construct. Once the construct is entered, all individual parts may proceed simultaneously dependent on the availability of resources. Control is not allowed to pass beyond the construct until all individual parts (called instances) have completed whatever computation they are to do.

The doall construct consists of a "DOALL" header, followed by a doall program segment followed by a doall terminating delimiter. The header will contain a specification of a domain, perhaps by name. If the domain in the header is the domain "EYEJAY" as declared in the example of the previous section, and IMAX = 100 and JMAX = 50, then there are 5000 intances of the doall program segment to be executed. Each instance of the doall program segment can execute independently of, and without any interaction with, every other instance of the doall program segment. Within each instance, there may not be any references to computations within any other instance, but no restrictions on references to "old" values exist. The computation within each instance may be conditional on location in the model, on data, or on any other condition.

Hardware Support of the DOALL Construct

An issue is the mapping of the DOALL construct onto real processor resources. A DOALL construct execution begins when processors 0 through 511 pick up instance numbers 0 through 511. For a DOALL with I and J for instance variables as in the example above, each processor computes I and J values from the instance number by solving the equation

```
              instance number = J*IMAX + I
```

Specifically, I is instance number modulo IMAX and J is instance number DIV IMAX. When each processor has finished its instance of the DOALL program segments, it increments instance number by 512, computes new I and J values, and proceeds to iterate thus until the I and J values computed are outside the domain. Once the processor has completed all assigned instances, it drops down to a "WAIT" instruction. When all processors get to "WAIT", a 512-way AND of the WAITing state is used to create a "go" signal which causes all processors to step to the next construct or instruction. Thus, an essential feature to make the DOALL construct work is a fast hardware synchronization operation. DOALL program segments can be as short as a single statement. A single-statement DOALL with regular subscripting on variables exactly corresponds to a vector operation in a vector machine and hence this MIMD architecture includes vector computations as a subset of its capabilities.

Waiting implies processor idle time. In the aerodynamic flow and weather codes which were analyzed during the study, the amount of processing per processor was nearly equal for all processors, and hence processor efficiency was high, the first processor to finish being only slightly ahead of the last.

Memory Allocation

System control is simplified by making decisions at compile time rather than having them made by system software art run time. The distinction between the various sorts of memory is made in the compiler with help from programmer declarations.

The potential four types of memory allocation are:

1. A variable or array element is visible to any part of the program, can be accessed from within any instance of a doall program segment, or from any serial section of code between doall program segments.

2. A variable is a temporary variable which need not remain defined after the end of the instance in which it is used.

3. A variable is so frequently accessed that each processor deserves to have its own local copy.

4. A one-to-one relationship between the elements of an array and the elements of a domain holds. Within the instances of a doall program segment over that domain, elements of that array are accessed in correspondence to the relationship.

The exact form of the declarations for helping the compiler make appropriate assignments of different data to different types of space is still under discussion. It is clear that some analysis on the compiler's part is possible; an array which is subscripted with the instance variables inside a doall must be either type 1 or type 4, for example. If the language is to be an extended Fortran, each common area must contain variables of only one category.

The sets of memory declarations suggested to date contain some common features. First, there is a declaration to the effect that a variable is shared (type 1). Second, there is a declaration (or default) that a variable is temporary to the instance (type 2). Third, there is a means for declaring that a set of variables is of type 4. This last is the "INALL" declaration. The INALL declaration couples a variable or array with the dimensionality and index set of a domain. For example, the declaration

```
INALL/EYEJAY/ C1, C2, A(5)
```

declares that there is an element of C1, an element of C2 and five elements of A associated with each element of the domain "EYEJAY". When there is a doall construct over the domain "EYEJAY (I,J)" then these variables can be used with the doall program segment and each instance will have its own copy. Referring to a variable such as C2 either without subscripts, or with "centered subscripts" i.e., "C2(I,J)", is permissible and functionally identical. Outside of doalls over "EYEJAY", these three identifiers will identify arrays which have dimensionality
C1(IMAX,JMAX), C2(IMAX,JMAX), and A(IMAX, JMAX,5) respectively.

Given that there are two kinds of memory space, memory private to each processor and memory shared by all processors, variables of type 2 and type 3 will be found in processor private memory, and type 1 would be in shared memory. If a variable of type 4 is only accessed within doalls over the appropriate domain, and always on centered subscripts, it can be held in the private memories of the processor that will compute the instances that are in one-to-one correspondence with the appropriate array elements.

Parallel Functions

Some common parallel operations and first-order linear recurrences would be supported by new intrinsics.

Parallel sum. Consider a variable defined within each instance at the end of a doall. The parallel sum of all those variables is created, which will then be accessible after the end of the doall. 512 such variables can be summed in 9 steps using interprocessor communication. Similar parallel functions are parallel AND, parallel OR, and MAXIMUM across all instances.

First-order linear recurrence. Given quantities B(I) and C(I) in each instance of a doall whose index set is I=1, IMAX, form the sequence A(I) = A(I-1)*B(I) + C(I). A(0) is given as an initial value. As with the parallel sum, this function can be implemented in N steps when IMAX = 2^N. [24]

Other Software Issues

Although the mechanisms shown demonstrate that one can design a langauge to enhance control of the MIMD machine by imposing structure and regularity on the MIMD interprocessor interactions at compile time, there are certain issues which have to be resolved before fixing on a final design for the language.

One issue is a trade between making memory allocation decisions based on programmer declarations and making allocation decisions by compiler analysis. Many users of high-throughput machines insist on being able to control every detail of machine action, out of fear that the vendor's compiler will be inefficient if left to its own devices.

Using Fortran as a starting point raises an issue that might not arise with some other starting point because of the requirement in Fortran for separate compilation. At compile time the compiler must distinguish between a subroutine called within a doall program segment where each instace of the doall calls its own copy, and a subroutine called outside the doall which runs on the array as a whole. The simplest solution would be to distinguish between the two kinds of subroutine by a difference in the SUBROUTINE statements.

"Every instance of the doall program segment must be independent of and free from any side effects that would interfere with any other instance of the same doall program segment". This over-simplified statement is true at the first level of understanding of the working of the machine. However, steps taken to enforce this rule are subject to a trade between authoritarian and libertarian schools of programming. There is no hardware limitation on the processors fetching or storing any variable in shared memory at any point in the program. Since the relative timing between actions that occur in different instances of the doall is not controlled, this allows for data accesses and definitions to occur in an uncontrolled order. Hence there is a question about the enforcement of data precedence. Absolute enforcement by the compiler, so that code which is emitted is guaranteed to be free of data precedence violations, may be undesirable. First, such a compiler will be unable to detect all cases in which the instances are independent of each other and as a result will forbid certain useful functions. Second, for some applications [25] a change in the sequence of performing the computations will change the result to another, different, but still acceptable result. One does not wish to forbid such programs. However, if the compiler made no check, gave the user no help, unnecessary errors might be committed. The following rule is observed to cover all cases that arise in the aero flow and weather codes, and appears simple to implement. "If an array element in shared memory is used on the right side of an assignment statement within a doall program segment then any assignment to that array in the same doall program segment must be on centered subscripts and will be held in a "new" copy of the array. The "new" copy will replace the old copy of the array at the time of synchronization at the end of the doall."

Hardware Details

Instead of implementation details, discussion below will concentrate on how hardware features support the langauge extensions.

Processor

Analysis of the aerodynamic flow and global weather model programs (provided by NASA Ames during the NASF Feasibility Study as samples of typical application programs) showed that up to several thousand processors could efficiently work in parallel. In these cases, the actual number of processors supplied is irrelevant over a large range; only total throughput matters. The design intent was to supply a processor that had maximum throughput at minimum cost. The trade-off evaluation was based on assumptions of the technology suitable for 1983 delivery and on the desire to limit complexity to control project risk. The result was 512 processors, each having capability of about 3Mflop/sec.

Each processor has independent integer and floating-point execution units with limited instruction look-ahead. To hide access time of the shared memory, each processor has a one-slot queue, called the "CN Buffer", which manages accesses to the shared memory while other processor operations go on concurrently. A processor-local memory of about 32K words is appropriate to the applications studied.

Shared Memory

In reference (1), the shared memory is called "Extended Memory" (EM). It consists of a prime number of memory modules (521) in order to reduce conflicts for the case that the pattern of accesses from the processors forms a regular pattern [26,27].

All processors independently compute accesses in shared memory, and independently access memory. Given that processor no. i is to access shared memory address A(i) the processor will compute address-within-module given by

$$L(i) = A(i) \text{ DIV } 512$$

and module number

$$M(i) = A(i) \text{ modulo } 521$$

When the addresses being accessed by the processors form a vector with constant stride the formula for the A(i) is

$$A(i)=A(0)+p*i$$

Here the M(i) fall into 512 different memory modules because p and the number of memory modules are relatively prime. This is the basis for claiming that a prime number of memory modules makes certain kinds of accessing "conflict-free".

Features for Fault Tolerance

Because of the flexibility of the connection network, a simple method of providing spare processors and memory modules is planned. Each

CN buffer contains a "replacement unit directory" to redirect connections around spare units. Single error correction, double error detection (SECDED) code covers all memory and transfers through the connection network. The connection network, being duplexed, has a simplex mode of operation as backup.

Staging Memory

Staging memory is called "Data Base Memory" in (1) where a size of 128 Mwd is assumed. Later discussions have centered on a size of 256 Mwd. Transfer rates must be on the order of 50 Mwd per second to and from shared memory. Access time requirements make disk undesirable. If staging memory were to be built of semiconductor components, then 256k-bit chips would be desirable.

The design and control of the staging memory has no surprises. The structure is one of a dual port memory. One port responds to requests from the coordinator for high-speed transfers between staging memory and Extended Memory. The other port is externally controlled and provides the high-speed data path to the rest of the system.

Connection Network

The connection network is used like a dial-up network, with any processor requesting connection to any memory module at any time, with the concommittant "message" being an address plus one word of data either stored to or fetched from the memory module involved. All processors could request simultaneously. Blockage must be low enough that the average added delay due to blockage is small compared to the time due to cable delays, access time of the memory module and memory conflicts. In addition processors must be treated "fairly". In the intended applications all processors have an equal amount of work to do. If any processor had a low probability of making its connections through the connection network, then that slower processor would tend to be the last processor arriving at the synchronization points, thereby slowing up the whole system.

The chosen configuration (Figure 3) is called the "baseline" network by Wu and Feng 28]. We first derived it as an isomorphism to the Omega network of Lawrie [29]. A parallel paper [22] discusses the design and validation of the connection network showing that it indeed performs as desired.

The time it takes to make a connection from any one of the 512 processors to any one of the 521 memory modules is estimated at 120 ns., barring conflicts or blockage. The throughput analysis of the FMP assumed a path width of 11 bits. During throughput analysis of the FMP, a particular distribution of shared memory conflicts and of blockage in the connection network was assumed. After the simulations to evaluate performance were nearly finished, simulation of the connection network [23] showed that the assumed delays were in fact correct.

Synchronization

Synchronization is mechanized by the WAIT instruction. A processor continues to execute WAIT until a "go" signal is received. The "go" signal is the 512-way AND of a signal emitted by each waiting processor. Synchronization ensures that no processor tries to fetch new data until that data has in fact been produced, perhaps by the slowest processor, in the preceding DOALL construct.

Figure 4 shows a mechanism whereby the 512-input AND gate is implemented as a tree-form cascade of 8-input AND gates (Figure 4 is actually drawn for a 27-input AND gate implemented as a cascade of 3-input AND gates; the number of levels in the tree comes out the same in either case). The root node of the tree reflects the "GO" signal back to all processors when the "AND" output is true at the root node. Note that the spare processors must always appear to be waiting even when being serviced or checked off-line from the primary problem.

The total delay from the last processor accessing a WAIT instruction until the "go" signal reaches all processors has been estimated at 160 ns.

Performance Validation

NASA had supplied two complete three-dimensional aerodynamic flow codes, solutions of the time-averaged Navier Stokes equations, and some weather codes. Three of these programs were completely analyzed. The method of analysis was to determine the calling sequence, the path of execution through the entire program, with notations as to how often each section of the code was called. Appropriate DO loops were converted into concurrent "DOALL" constructs in which DO iterations are converted into DOALL instances. Representative sections of the programs were exercised in simulation to determine running time. Other sections had their running estimated based on how their parameters were related to the parameters of the simulated sections. The most significant parameter was the number of floating point operations per reference to the shared memory. The running time and number of floating point operations in each section are each summed to give the running time for the whole program and the number of floating point operations for the whole program. The quotient of these two totals is then the throughput for the entire program in terms of floating point operations per second. Details are in [1] in Appendix A.

The results of this analysis are summarized in Table I. In brief, performance met the target of 1.0 Gflop/sec for favorable aerodynamic applications, and varied from 0.5 Gflop/sec on up for other suitable applications. The chemistry and radiation portions of the global circulation model were not vectorized, but consisted of a doall with one instance at each point on the globe; the doall program segment having much data dependent branching within it.

Conclusion

A generalization of vector architectures for high-throughput numerical computing has been presented. The lack of any need to vectorize the application should make it more widely applicable than are the current generation of vector machines. Validation using actual application programs supports the expectation of high throughput.

The three programming constructs are the parallel execution of many instances of the same code, the use of named index sets, and the concept of two types of memory, one private to a single instance, the other shared across the entire program.

Acknowledgements

In any project of this size, many people contribute. The authors have singled out, for special acknowledgement of their contributions, Howard Pearlmutter and Philip E. Shafer.

References

[1] Final Report, Numerical Aerodynamic Simulation Facility Feasibility Study, Contract No. NAS2-9897 Burroughs Corporation, Paoli, PA, for NASA Ames, March 1979.

[2] R. M. Russell, "The Cray-1 Computer System", Communications of the ACM, Volume 21, No. 1 January 1978, pp. 63-72.

[3] P. M. Flanders, D. J. Hunt, S. F. Reddaway, D. Parkinson, "Efficient High Speed Computing with the Distributed Array Processor", in High Speed Computer and Algorithm Organization, ed. D. J. Kuck, et al, Academic Press, 1977, pp. 85-89 (SIMD).

[4] R. A. Stokes, "Burroughs Scientific Processor", in High Speed Computer and Algorithm Organization, ed., D. J. Kuck et al, Academic Press, 1977 pp. 85-89.

[5] L. Fung, "A Massively Parallel Processing Computer", in High Speed Computer and Algorithm Organization, ed., D. J. Kuck et al, Academic Press 1977, pp. 203-204 (MPP).

[6] D. R. Chapman, "Computational Aerodynamics Development and Outlook", Dryden Lectureship in Research, 17th Aerospace Sciences Meeting 1979 NASA Technical Report 79-0129.

[7] T. Christopher, O. El-Dessouki, M. Evens, P. Greene, A. Hazra, W. Huen, A. Rastogi, R. Robinson, and W. Wojciechowski, "Uniprogramming a Network Computer", 1978 International Conference on Parallel Processing IEEE, Computer Society, Long Beach CA, 1978, pp. 312-138.

[8] D. J. Kuck, "A Survey of Parallel Machine Organization and Programming", Computing Survey, Volume 9, No. 1 (March 1977), pp. 29-60.

[9] D. H. Lawrie, T. Layman, D. Baer, J. M. Randal, "Glypnir - A Programming Language for Illiac IV", Communications of the ACM, Volume 18, No. 3, March 1975, pp. 157-164.

[10] E. W. Davis "STARAN Parallel Processor System Software", AFIPS National Computer Conference, 1974, pp. 17-22.

[11] J. R. Dingledine, H. G. Martin, and W. M. Patterson, "Operating System and Support Software for PEPE", Sagamore Conference on Parallel Processing, Proceedings, 1973 IEEE, pg. 170-178 (claims to describe PFOR).

[12] Burroughs Corporation, Burroughs Scientific Processor (BSP) Fortran Reference Manual, Ref. No. 1118338, February 1980, Paoli, PA.

[13] J. B. Dennis, D. P. Misunas, and C. K. Leung, "A Highly Parallel Processor Using a Data Flow Machine Language", Computation Structures Group Memo. 134, MIT, January 1977.

[14] P. Brinch-Hansen, "The Programming Language Concurrent Pascal", IEEE Transactions on Software Engineering, June, 1975, pp. 199-207.

[15] J. P. Anderson, "Program Structure for Parallel Processing", Communications of the ACM, Volume 8, No. 13 (December 1965), pp. 431-155. (Very early discussion of "conventional" multiprocessors).

[16] D. J. Kuck, P. P. Budnick, S. C. Chen, E. W. Davis, Jr., J. C. Han, P. W. Kraska, D. H. Lawrie, Y. Muraoka, R. E. Strebendt, and R. A. Towle, "Measurements of Parallelism in Ordinary Fortran Programs", IEEE Computer, Vol. 7, No. 1, pp. 37-46, Jan, 1974.

[17] Leslie Lamport, "Parallel Execution of DO Loops", Communications of the ACM, Volume 17, No. 2, February 1974, pp. 83-93.

[18] R. J. Swan, S. H. Fuller, D. P. Siewiorek, "Cm*, a Modular, Multiprocessor", in "Collection

of Papers on Cm*", Technical Report, Computer Science Dept., Carnegie-Mellon University, February, 1977.

[19] W. A. Wulf, C. G. Bell, "C.mmp - A Multi-mini-processor", AFIPS Conference Proceedings Vol. 14, Part II, FJCC 1972, pp. 765-777.

[20] H. J. Siegel, P. T. Mueller, Jr., and H. E. Smalley, Jr., "Control of a Partitionable Multi-microprocessor System", Proceedings of the 1978 International Conference on Parallel Processing, IEEE Computer Society, 1978.

[21] Burton J. Smith, "A Pipelined, Shared Resource MIND Computer", Proceedings of the 1978 International Conference on Parallel Processing, IEEE Computer Society, 1978.

[22] R. Kober, C. H. Kunzia, "SMS - A Multi-processor Architecture for High Speed Numerical Calculations", Proceedings of the 1978 Interatnional Conference on Parallel Processing, IEEE Computer Society, 19781.

[23] G. H. Barnes, "Design and Validation of a Connection Network for Many-processor Multi-processor Systems", this conference.

[24] S. C. Chen, D. J. Kuck, "Time and Parallel Processor Bounds for Linear Recurrence System", IEEE Transactions on Computers, Volume C-24, No. 7, July 1975, pp. 701-717.

[25] Gerald M. Baudet "Asynchronous Iterative Methods for Multiprocessors", Journal of the ACM, Volume 25, No. 2, April 1978, pp. 226-244.

[26] P. Budnick and D. J. Kuck, "The Organization and Use of Parallel Memories", IEEE Transactions on Computers, December 1971.

[27] Roger C. Swanson, "Interconnection for Parallel Memories to Unscramble p-ordered Vectors, IEEE Transactions on Computers, November 1974.

[28] C. Wu and T. Feng, "Routing Techniques for a Class of Multistage Interconnection Networks", Proceedings of the 1978 International Conference on Parallel Processing, IEEE Computer Society, 1978.

[29] D. H. Lawrie, "Access and Alignment of Data in an Array Processor", IEEE Transactions on Computers, C-24 (1975), pp. 1145-1155.

Table I

Performance Summary

Case	Grid Size	No. Time Step	Thru put, Gf/s	Run Time min.
Implicit	100x 50x200	100	1.01	6
Explicit	100x100x100	100	0.89	9
Weather	89x144x 9	1008	0.53	4.5
FFT	512 to 4096	-	0.45-0.7	-

Implicit	= Implicit Aero Flow Code
Explicit	= Mixed Explicit/Implicit Aero Flow Code
Weather	= Global Circulation Model
FFT	= Fast Fourier Transform

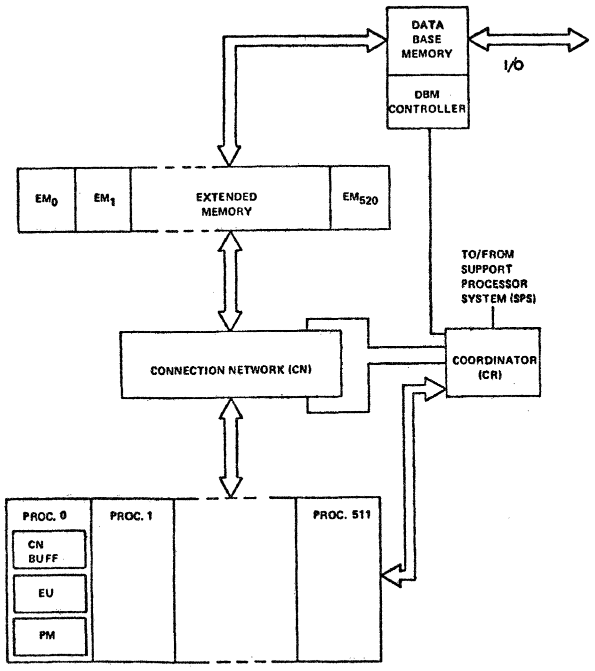


Fig. 1. Block Diagram

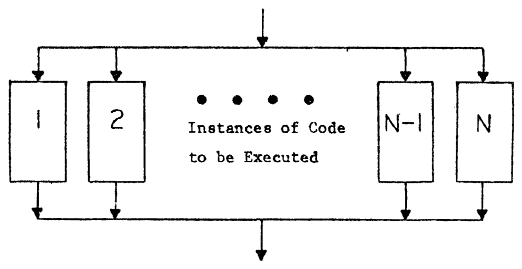


Fig. 2. Flowchart, Concurrent Construct

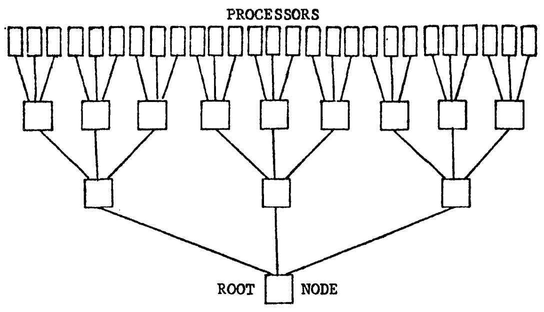


Fig. 4. Tree Form of AND Implementation

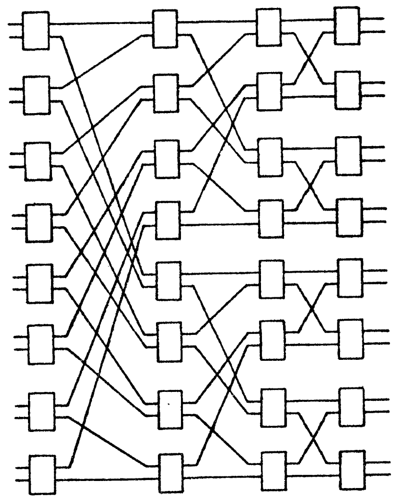


Fig. 3. Form of Connection Network

ARRAY MACHINE CONTROL UNITS FOR LOOPS CONTAINING IFs*

U. Banerjee,** D. Gajski and D. Kuck
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

Abstract -- The control unit is the interface between the compiler and the processing part of a computer. A number of array (parallel or pipeline) machines have been built with scalar or array instruction sets. Most such machines do a poor job of handling sparse data arrays and this paper addresses how such computations may be better handled. We emphasize two areas:

1. Conditional statements can lead to Boolean recurrences that must be solved to generate control bits. We discuss hardware for the solution of Boolean recurrences.

2. Sparse array computations lead to difficult memory access and data alignment problems. We discuss an efficient bit string approach to handling such computations.

1. Introduction

The control unit of any computer is the point at which the compiler meets the rest of the computer system. Thus, a well-designed control unit is necessary in achieving good system performance. In an array processor, the control unit is relatively more important because the system is more complex. Furthermore, compiler algorithms should be designed hand-in-hand with the control unit to achieve high system performance for ordinary user programs.

In this paper we discuss a subject that has seldom been handled well in existing parallel or pipeline machines, namely, the processing of sparse arrays in an efficient manner. We will present ideas that can be applied to array machines that execute single-array operations, which we denote SEA (single execution, array), and can be regarded as simple parallel or pipeline machines. The ideas are also useful in MEA multiple execution, array) machines which can execute several array operations simultaneously, and MES (multiple execution, scalar) machines, which can be regarded as tightly coupled multiprocessors whose goal is the speedup of one program at a time [KuPa79]. For more discussion of the above notation, see [Kuck78].

Specifically, we will discuss three hardware aspects of executing programs: accessing data in parallel memory units, alignment networks that pair proper array elements, and the processing pattern of the elements. In a parallel machine the problem is pairing elements in different processors, while in a pipeline machine the problem is pairing elements to be fed into the pipeline.

We assume that a traditional serial language is used to specify array operations, and that arrays are stored densely in a parallel set of memory units. The problem arises when conditional statements in loops cause the selection of only a limited, random set of the array operations to be performed. We will show that there are simple synchronous ways of accessing and aligning such arrays that should give high performance in most programs.

Two aspects of programs will be discussed. First, IF-statements contained in the scope of iteration statements (e.g., DO loops) give rise to mode bits that are used to control the execution of subsequent statements. We will discuss the fast generation of such mode bits, even when cycles of dependence are involved. This gives rise to new algorithms for the fast execution of Boolean recurrences.

Secondly, we discuss the use of mode bits in executing arithmetic array statements. Here the problem of accessing sparse arrays in parallel memories arises. We will present some theoretical results, sketch some hardware and give an example of the operation of our ideas. Formerly, high degrees of vectorization have been achievable in these cases, but the sparseness of the vectors led to poor efficiency unless the arrays

*This work was supported in part by the National Science Foundation under Grant No. US NSF MCS77-27910.

**Current address is:
Honeywell Information Systems - M/S C87
P. O. Box 6000
Phoenix, AZ 85005

CH1569-3/80/0000-0028$00.75 © 1980 IEEE

were first compressed [Kuck76].

We do not discuss the handling of compressed arrays. Most serial languages do not have explicit ways of specifying compress and expand operations; however, they may be useful operations when arrays are extremely sparse or indexing patterns are such that the methods we describe perform poorly. Some languages and software systems do allow the specification and manipulation of sparse arrays, and these are useful in many applications. In [Kuck70], this subject was dealt with for a few special cases and the ideas of this paper can be extended to this area as well, but are beyond our present scope.

The remainder of the paper contains five sections. In Section 2, some background ideas are presented. Section 3 discusses Boolean recurrences and Section 4 is about arrays and mode bits. Section 5 contains a detailed example and Section 6 gives some remarks and conclusions.

2. Theoretical Background

Here we briefly discuss the theoretical foundations of our work. For more details, see [Bane79]. Earlier results about compilation with conditional statements may be found in [Towl76] and [Kuck76].

Consider an arbitrary program consisting of loops, assignment statements, and conditional statements. Because of the presence of the conditional statements in the program, some instances of some of the statements may fail to get executed. For each assignment statement S, we define a Boolean valued function F_S of a suitable set of (loop) index variables, such that

(1) F_S has a value for each instance of S;

and (2) the value of F_S corresponding to a given instance of S is 1, iff that instance must be executed. We call F_S the mode function of S and its values the mode bits for S. Clearly, the mode function of a statement is determined by the conditions of all the conditional statements whose scopes contain that statement. The efficient generation of mode bits and their proper use is our main concern.

The statements in the program are dependent upon one another in a certain way. Using this dependence structure, we can break up the given program into a partially ordered set of subprograms. The same results would be obtained, if instead of executing the given program serially we execute the subprograms in any parallel way, as long as a subprogram is never started until all of its predecessors have finished.

No subprogram can be further decomposed along similar lines. Moreover, these subprograms can be grouped into several classes according to their characteristics, among which are the classes of cyclic mixed subprograms, acyclic subprograms, and cyclic arithmetic subprograms.

A cyclic mixed subprogram is such that some of the variables defining the conditions of its conditional statements are evaluated within the subprogram itself. This leads to the design of programmable hardware for the solution of Boolean recurrences (Section 3). A Boolean recurrence $B\langle n,m\rangle$ of degree n and order m ($1 \le m < n$) is a set of equations of the form

$$x_k = \phi_k(x_{k-1}, x_{k-2}, \ldots, x_{k-m}) \quad (1 \le k \le n)$$

where $x_1, x_2, \ldots, x_n$ are Boolean variables and $x_0, x_{-1}, \ldots, x_{-m+1}$ Boolean constants.

Consider now a subprogram where all variables defining the conditions of all the conditional statements are computed outside the subprogram. If in addition there is exactly one assignment statement, all of whose instances can be executed independently of one another, then we have an acyclic subprogram. Thus the mode bits for the unique assignment statement are known at execution time. This leads to the use of mode bits to control the execution of array assignment statements, involving the accessing of memory and aligning of data to and from memory. We will see in Section 4 that hardware for this can easily be added to standard indexing hardware, and this extends the earlier work on conflict-free array access [BuKu71], [Lawr75].

A cyclic arithmetic subprogram has one or more arithmetic assignment statements which are dependent upon one another or on themselves; except for that, it is similar to an acyclic subprogram. Here also the mode bits are known at execution time, but the instances of the assignment statements can no longer be executed independently. A subprogram of this kind is equivalent to an arithmetic recurrence with mode bits. A comment is made on the solution of linear arithmetic recurrences with mode bits in the final section; we do not discuss this problem in detail. (The definition of an arithmetic recurrence is obtained from that of a Boolean recurrence given above by making the obvious changes. An arithmetic recurrence is linear, if each ϕ_k is a linear function of its arguments.)

3. Generation of Mode Bits

3.1 Cyclic Mixed Subprograms

Consider the following example.

```
     DO  k = 1, 100, 1
     IF [C(k) > U(k) + C(k-1)]
     THEN BEGIN
S1              C(k + 1) = V(k + 1)
S2              Y(k)     = C(k + 2) + Y(k - 1)
          END
     ELSE BEGIN
S3              C(k + 1) = W(k + 1)
          END
```

Let x_k denote the condition of the IF statement.

Then the Boolean variables $x_1, x_2, \ldots, x_{100}$ satisfy the Boolean recurrence B<100,2> described below.

$$x_k = a_{k0}\,\bar{x}_{k-1}\,\bar{x}_{k-2} + a_{k1}\,\bar{x}_{k-1}\,x_{k-2} + a_{k2}\,x_{k-1}\,\bar{x}_{k-2} + a_{k3}\,x_{k-1}\,x_{k-2} \qquad (1 \le k \le 100)$$

where

$$x_{-1} = 0,\ x_0 = 0;$$

$$a_{10} = [C(1) > U(1) + C(0)],\ a_{11} = a_{12} = a_{13} = 0;$$

$$a_{20} = [W(2) > U(2) + C(1)],\ a_{21} = 0,$$

$$a_{22} = [V(2) > U(2) + C(1)],\ a_{23} = 0,$$

$$a_{k0} = [W(k) > U(k) + W(k-1)]$$

$$a_{k1} = [W(k) > U(k) + V(k-1)]$$

$$a_{k2} = [V(k) > U(k) + W(k-1)]$$

$$a_{k3} = [V(k) > U(k) + V(k-1)] \qquad (k = 3, 4, \ldots, 100)$$

(Here [...] represents a Boolean valued expression.)

The Boolean coefficients a_{kt} $(1 \le k \le 100,\ 0 \le t \le 3)$ of this recurrence can be computed in parallel on a vector machine like one shown in Fig. 1. They are all stored in the Boolean-coefficient memory. After n sets of coefficients are stored, the Boolean-recurrence solver generates mode-function bits for n iterations of the loop. Those bits are stored in the mode-function register and control the parallel execution of the true and false branches of the conditional statement in the loop. In our example, the statements S_1 and S_2 are executed in each processor that has a mode-function bit equal to 1. Processors with mode-function bit equal to 0 are turned off. After S_1 and S_2 have been executed, the content of the mode-function register is complemented and statement S_3 is executed.

If the upper limit of index k is much larger than the number of processors n, the execution of the loop can be partitioned into n-iteration slices. In this case, the computation of mode-functions by solving Boolean recurrence can be overlapped (pipelined) with the computation of Boolean coefficients and execution of the IF statement. This way IF statement control becomes time-transparent to the original vector machine. Thus, we must be able to solve such Boolean recurrences.

We now consider a general cyclic mixed subprogram. From this program we extract a Boolean recurrence. To evaluate the kth variable x_k of this recurrence, we need to know certain values computed inside the subprogram itself and certain values coming from outside. The values (arithmetic and Boolean) coming from outside will be

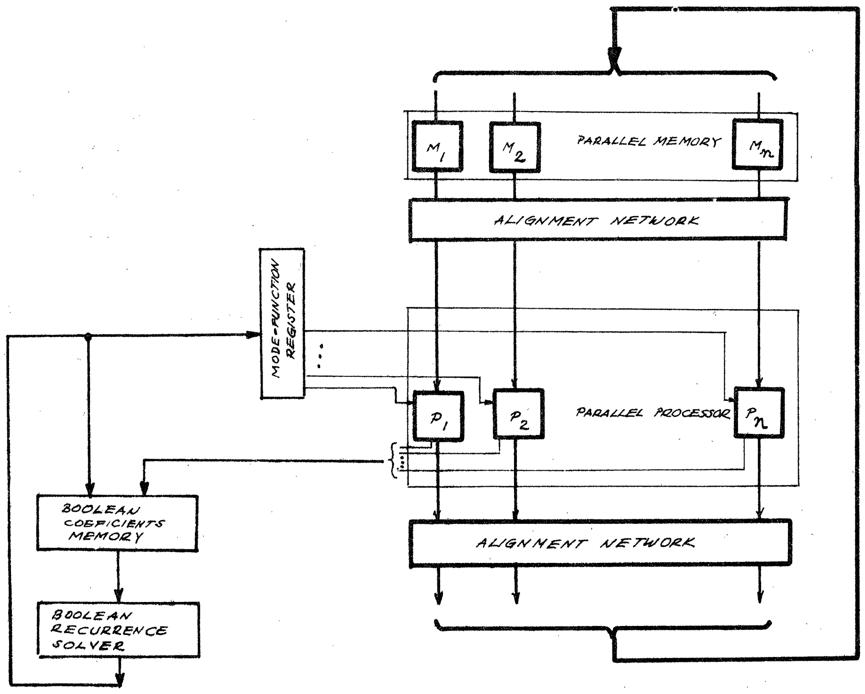


Fig. 1. Control hardware for the loops with IF statement

completely known at run-time, and they are to be treated as constants. The formula defining x_k can be expressed in terms of the constants in at most 2^{k-1} different ways, but frequently requires much less than that, since several different paths may lead to the same expression and hence can be combined.

3.2 Solution of Boolean Recurrences

In what follows, n and m denote two integers such that $1 \leq m < n$. Consider an arbitrary set of m Boolean variables $\{y_1, y_2, \ldots, y_m\}$. The 2^m minterms of these variables are numbered 0, 1, 2, ..., 2^{m-1} in the usual way, and the t^{th} minterm is denoted by $P_t(y_1, y_2, \ldots, y_m)$. We will use AND and OR gates, such that each gate has a gate delay of one unit of time. It is assumed that each gate gives true and complemented outputs with no time or cost penalty. The sole purpose of this assumption (which holds for ECL circuit family gates) is to keep our formulas simple; extension to the general case is easy and straightforward. For any positive integer k, we write log k to denote $\lceil \log_2 k \rceil$.

Let us define a Super Cell (SC) (Fig. 2(b)) to be a piece of combinational logic which takes $(m+1)^{2m}$ inputs $\{a_s | 0 \leq s \leq 2^m-1\} \cup \{b_{rt} | 1 \leq r \leq m, 0 \leq t \leq 2^m-1\}$ and produces 2^m outputs c_t defined by

$$c_t = \sum_{s=0}^{2^m-1} a_s P_s(b_{1t}, b_{2t}, \ldots, b_{mt}) \qquad (0 \leq t \leq 2^m-1)$$

where each c_t is realized by the logic in a Basic Cell (BC) (Fig. 2(a)). The following lemma is obvious.

Lemma 1

If fan-in and fan-out considerations are ignored, then an SC can be implemented in 2 gate delays with $2^m(2^m+1)$ gates. ■

Consider now a general Boolean recurrence B<n,m> of degree n and order m, defined by the equations

$$x_k = \sum_{t=0}^{2^m-1} a_{kt} P_t(x_{k-1}, x_{k-2}, \ldots, x_{k-m}) \qquad (1 \leq k \leq n)$$

where the a's and $x_0, x_{-1}, \ldots, x_{-m+1}$ are known Boolean constants.

Theorem 1

If fan-in and fan-out considerations are ignored, then the Boolean recurrence B<n,m> can be solved in

$2(\log n + 1)$ gate delays

with

$[(\frac{n}{2} \log n) 2^m (2^m+1) + n(2^m+1)]$ gates. ■

For a proof of this theorem and for results in the limited fan-in, fan-out case, see [Bane79]. As an example, the solution of B<8,2> is shown in Fig. 2(c).

(a)

(b)

(c)

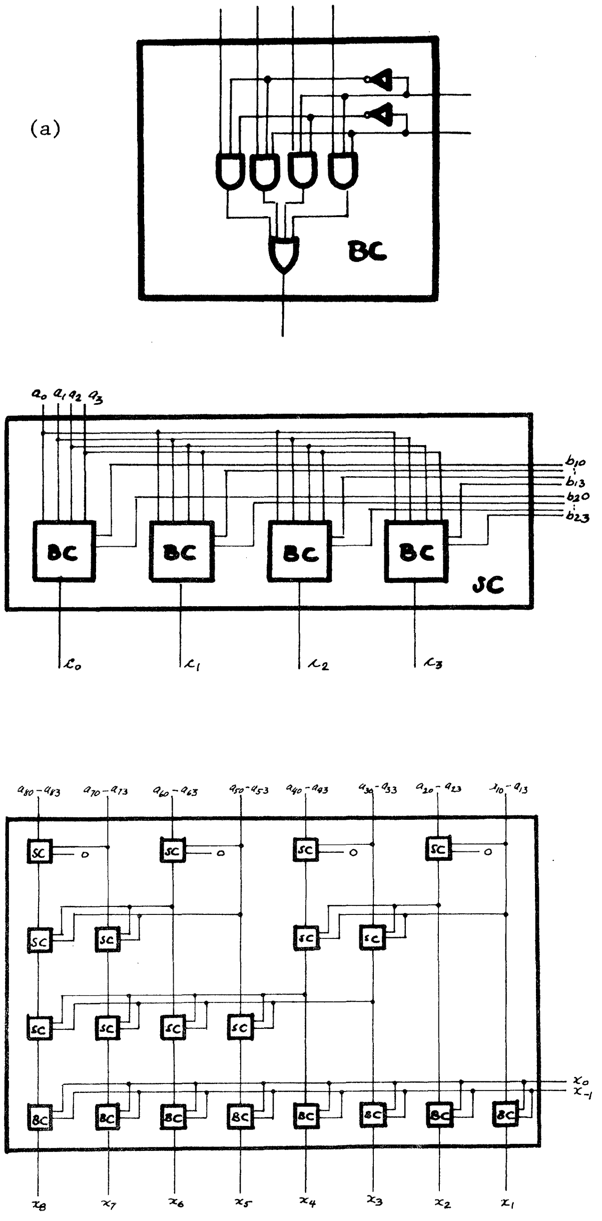


Fig. 2.

(a) Ba[illegible]c Cell (BC) for m = 2

(b) [illegible]per Cell (SC) for m = 2

([illegible] [illegible]tion of Boolea[illegible] recurrence B<8,2>

4. Arrays and Mode Bits

Assuming that mode bits exist, we now discuss their use in memory accessing for arrays. We will discuss alignment later. Its implementation is straightforward with a crossbar switch but less costly with an extended omega network [Lawr75].

Assume a storage scheme such that the element X(I) of an array X is stored in memory module number f(I), is given by

$$f(I) = I + \text{Base}_X) \bmod m$$

where Base_X is the number of the module that contains X(1) and m is the total number of memory modules. The following two results are crucial for our discussion; for proofs, see [Bane79]. (The notation of this section is somewhat different from those of the previous sections.)

Lemma 2

Let A_0, a denote integers such that gcd(a,m) = 1. Then the elements $X(a_0 + ai)$ and $X(a_0 + aj)$ of an array X are stored in the same memory module, iff (j - i) is a multiple of m. ∎

An immediate consequence of this lemma is the following corollary.

Corollary 1

Let a_0, a, n denote integers such that gcd (a,m) = 1 and $0 < n \leq m$. Then for any i, the set of elements $\{X(a_0 + aI) | i \leq I \leq i + n - 1\}$ of an array X can be accessed from memory without any conflicts. ∎

Consider now the program

```
     DO  I = 1, u, 1
S        $Z(c_0 + cI) = X(a_0 + aI)$ op $Y(b_0 + bI)$
     END
```

where X, Y, Z are one-dimensional arrays, u, c_0, c, a_0, a, b_0, b are integer constants, and op some valid operation. (The conditional statements are not shown explicitly; we deal with the mode function of S instead.) Let us assume that m is a prime number and that none of a, b, c is a multiple of m. If the value of the mode function F_S of statement S is equal to 1 for each value of I, then everything works just fine. We can fetch $X(a_0 + aI)$ and $Y(b_0 + bI)$ and store the result of op in $Z(c_0 + cI)$ for any m consecutive values of I, without ever getting a conflict. However, in general, $F_S(I) = 1$ only for a random set of values of I. And only those instances of statement S are to be executed for which $F_S(I) = 1$. We may still fetch the full set $\{X(a_0 + aI)|1 \leq i \leq m\}$ without any conflicts, but now probably only a few of these values need to be sent to the processors.

The above lemma points to a way of avoiding this potential inefficiency. We look at a number of values of $F_S(I)$, much larger than m. A set of m or fewer 1's are selected from these values, such that if $F_S(i)$ and $F_S(j)$ are in this set and $i \neq j$, then (j - i) is not a multiple of m. The values of the index I corresponding to these bits are guaranteed not to produce any conflicts in the memory addresses of the elements $X(a_0 + aI)$ of any arbitrary array X, as long as gcd(a,m) = 1. Our scheme fails when gcd(a,m) > 1, but then nothing can be done in that case; $X(a_0 + aI)$ will lie in the same memory module independently of I. If m is a large prime number, these instances of failure should occur very rarely.

The selection of m or fewer mode-bits with value 1 is accomplished by the Mode-Function Compressor (MFC) shown in Fig. 3. The MFC has two outputs: mode bits and their corresponding indices, and it can be thought of as consisting of m content-addressable memories, each storing pairs of the form $(F_S(i), i)$. Any two pairs $(F_S(i), i)$ and $(F_S(j), j)$ have (mod-m)-equivalent index values; that is, $i = j (\text{mod } m)$. Each memory when enabled, reads out the first value $(F_S(i), i)$ with $F_S(i) = 1$, or the pair $(F_S(i) = 0, i = 0)$ is issued when there is no pair with $F_S(i) = 1$. Mode bits are stored in Mode-Function Register as before. $F_S(i) = 0$ will turn off the corresponding processor P_i which will generate a null result that is never stored in any module of the Parallel Memory. The corresponding index values are sent to the memory address generator which generates memory address for each memory unit from the common vector descriptor containing a_0, a, and Base_X for each vector $X(a_0 + aI)$.

The set of m associative memories may become prohibitively costly for reasonable values of m (16 to 64) and index set I (1024 to 4096). For this reason, we will now describe a less costly but slower design of the MFC (Fig. 4).

Part of this design is similar to a paging system. Suppose that the set of all values of the mode function F_S (in the Boolean Coefficient Memory) has been broken up into a number of "pages," each page being m bits long. Page 1 consists of the values $\{F_S(1), F_S(2), \ldots, F_S(m)\}$, page 2 consists of $\{F_S(m + 1), F_S(m + 2), \ldots, F_S(2m)\}$, and so on. There are L "page frames," where L is some suitably chosen number, and a frame is an m-bit register. The "page table" consists of L lines, where line ℓ gives the number of the page residing in frame ℓ, and a test bit which is 1 iff frame ℓ

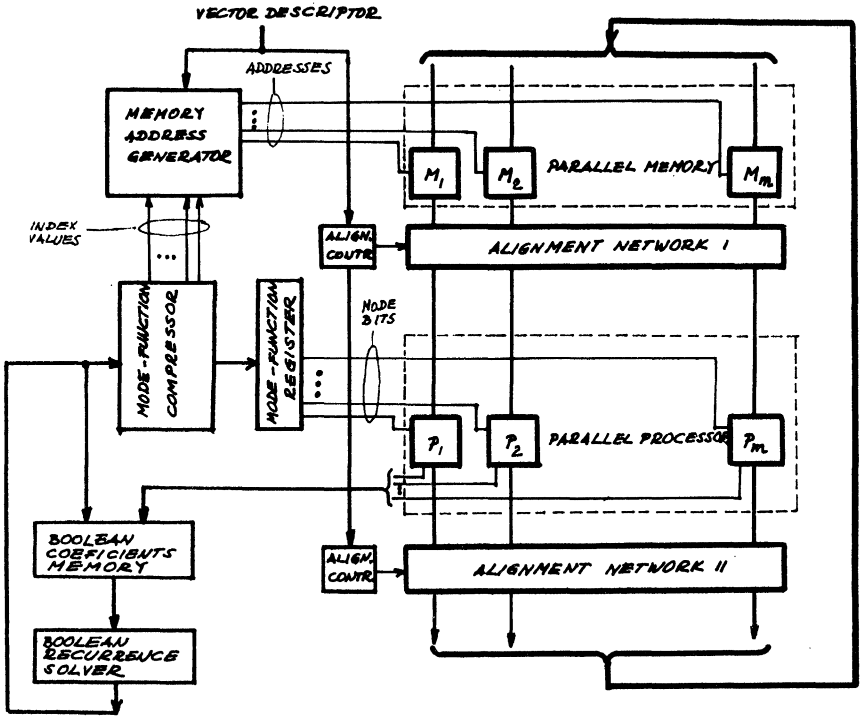


Fig. 3. Control for the high speed execution of loops with IF statement

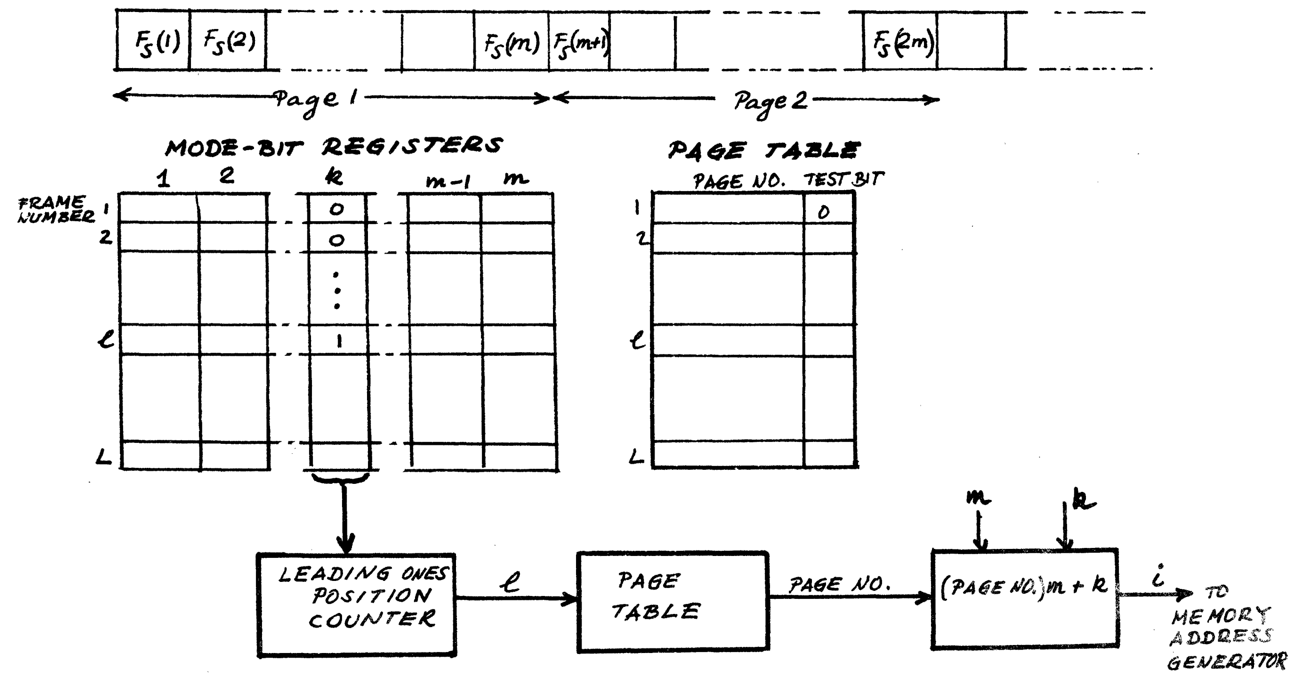


Fig. 4. Implementation of the MFC

has at least one 1 ($1 \leq \ell \leq L$). A page is brought to the frames iff it has at least one 1. (We assume that the sum of all the bits in a given page is also stored in memory, and that this bit is tested before the page is brought out.) No page is brought to the frames more than once. Any frame can hold any page. When the time comes to bring new pages into frames, a frame is refilled iff all the 1's of the page originally residing in this frame have been used up (as indicated by its test bit). We will see that the 1's in frame 1 are always used up before refill-time, and hence its test bit should be permanently fixed at 0.

We start by bringing L pages into the L frames. Then we choose the leading 1 in each of the m columns, i.e., the leading 1 among the 1st bits of all frames, the leading 1 among the 2nd bits of all frames, and so on. The values of the index I corresponding to these bits lead to executable instances of statement S, and they do not cause memory conflicts. If the position of the leading 1 in the k^{th} column is ℓ, then the value i of index I corresponding to this bit is given by

$$i = (\text{number of page in frame } \ell - 1)m + k$$
$$(1 \leq k \leq m,\ 1 \leq \ell \leq L,\ 1 \leq i \leq u)$$

If the k^{th} column has at least one 1, then the index value i corresponding to the leading 1 goes to the Memory Address Generator.

Before the process is repeated, we must reset the leading 1-bit in each column and update the test bit for each frame. New pages are brought into the frames whose test bits are equal to 0. And we start all over again. If the loop-size is large and the steady stage is reached, we should be able to get out m (or close to m) conflict-free index values from the MFC, for a number of times.

5. Example

In this section we present an example of handling sparse array operations using the method of the previous section. As was mentioned earlier, the idea can be used for a register-to-register pipelined processor as well as for a parallel machine as sketched here.

Consider the program of Fig. 5(a), a segment of a larger program, in which the X array is tested and C(I) is updated whenever X(I) is non-negative. Fig. 5(b) shows those index values (I) for which this test is true. Given a memory system with five memory units, conflict-free access to array elements is guaranteed except for such subscripts as 5I, 10I + 3, etc. Such a memory is shown in Fig. 6.

A snapshot of the system in Fig. 3 is shown in Fig. 7. It is assumed that the entire mode function has been computed and stored in mode-bit registers inside the MFC. The contents of the mode-bit registers are shown in the first row in Fig. 7.

We now have the problem of accessing only hose elements of arrays for which the mode bits

```
DO  I = 1, 15
    IF (X(I) ≥ 0)
    THEN C(I) = A(2I + 1) + B(I + 3)
END
```

(a) The program segment

1, 3, 4, 7, 8, 9, 11, 12, 13, 15

(b) Values of I for which X(I) ≥ 0

Fig. 5. Program with IF in loop

Fig. 6. Memory units with stored arrays

MEMORY ADDRESS	M_1	M_2	M_3	M_4	M_5
1	A_1	A_2	$\enclose{circle}{A_3}$	A_4	A_5
2	A_6	$\enclose{circle}{A_7}$	A_8	$\enclose{circle}{A_9}$	A_{10}
3	A_{11}	A_{12}	A_{13}	A_{14}	$\enclose{circle}{A_{15}}$
4	A_{16}	$\boxed{A_{17}}$	A_{18}	$\boxed{A_{19}}$	A_{20}
5	A_{21}	A_{22}	$\boxed{A_{23}}$	A_{24}	$\boxed{A_{25}}$
6	A_{26}	$\triangle{A_{27}}$	A_{28}	A_{29}	A_{30}
7	$\enclose{circle}{A_{31}}$	A_{32}	B_1	B_2	B_3
8	$\enclose{circle}{B_4}$	B_5	$\enclose{circle}{B_6}$	$\enclose{circle}{B_7}$	B_8
9	B_9	$\enclose{circle}{B_{10}}$	$\boxed{B_{11}}$	$\boxed{B_{12}}$	B_{13}
10	$\boxed{B_{14}}$	$\boxed{B_{15}}$	$\triangle{B_{16}}$	B_{17}	$\enclose{circle}{B_{18}}$
11	B_{19}	B_{20}	B_{21}	B_{22}	B_{23}
12	B_{24}	B_{25}	B_{26}	B_{27}	B_{28}
13	B_{29}	B_{30}	B_{31}	B_{32}	$\enclose{circle}{C_1}$
14	C_2	$\enclose{circle}{C_3}$	$\enclose{circle}{C_4}$	C_5	C_6
15	$\enclose{circle}{C_7}$	$\boxed{C_8}$	$\boxed{C_9}$	C_{10}	$\boxed{C_{11}}$
16	$\boxed{C_{12}}$	$\triangle{C_{13}}$	C_{14}	$\enclose{circle}{C_{15}}$	C_{16}
17	C_{17}	C_{18}	C_{19}	C_{20}	C_{21}
18	C_{22}	C_{23}	C_{24}	C_{25}	C_{26}
19	C_{27}	C_{28}	C_{29}	C_{30}	C_{31}
20	C_{32}				

are 1. The 15 bits (one per loop iteration) are folded over in rows of length 5 (one column per memory unit). After one leading one's detection in each column, mode bits corresponding to index values 1, 7, 3, 4, and 15 are selected and they appear at the outputs of MFC. At the same time the MFC outputs five 1's to the Mode-Function Register. The array elements A(3), A(15), A(7), A(9), and A(31) correspond to index values of 1, 7, 3, 4, and 15.

Using the code ○ for this first set of elements and referring to Fig. 6, we see that all ○ elements in the A array can be fetched without conflict. Similar statements can be made about accessing the ○ elements in the B and C arrays. The second cycle in Fig. 7 shows the mode bit registers after the first set of 1's are deleted and the results of a second leading one's detection are presented with the □ code: the elements are also marked in Fig. 6. On a third cycle, only one element, marked △ , would be accessed. Note that five elements are accessed on the first cycle, four on the second and one on the third. The effective memory bandwidth will always drop off toward the end of a vector access, but will remain high on earlier cycles as long as the addresses are uniformly distributed across the memory units.

Next, consider the processing of data for this program using the five processor parallel machine of Fig. 3. The Memory Address Generator calculates from index values supplies from MFC and vector descriptor supplied by the control unit the proper addresses of array elements. For example, an array indexed as $A(a_0 + ai)$ has a_0, a, $\text{Base}_{\text{Unit}}$, and $\text{Base}_{\text{Addr}}$ included in the vector descriptor, where $\text{Base}_{\text{Unit}}$ and $\text{Base}_{\text{Addr}}$ are memory unit number and address of A(1). For each index value i the address $(\lceil (a_0 + ai + \text{Base}_{\text{Unit}} - 1)/m\rceil - 1) + \text{Base}_{\text{Addr}}$ is supplied to memory unit $1 + (a_0 + ai + \text{Base}_{\text{Unit}} - 2) \bmod m$. For details, see [LaVo80]. We see that the array elements from A and B arrays are not paired properly for processing.

This leads us to our final point, consideration of data alignment between memory units and processors. It is obvious that if a crossbar switch is provided between processors and memories, then the proper alignments would be possible. Instead of an $O(n^2)$ gate switch between n memory and processor units, however, we can employ an $O(n \log n)$ gate omega network [Lawr75], because only uniform shifts and squeezes are involved. Thus, an array indexed as $A(a_0 + ai)$ can be aligned with an array indexed as B(i), by a shift

	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
MODE-BIT REGISTERS	1 0 1	0 1 1	1 1 1	1 1 0	0 0 1	0 0 1	0 0 1	0 1 1	0 1 0	0 0 0	0 0 0	0 0 0	0 0 1	0 0 0	0 0 0
MODE-FUNCTION COMPRESSOR (index values)	1	7	3	4	15	11	12	8	9	-	-	-	13	-	-
MODE-FUNCTION COMPRESSOR (mode bits)	1	1	1	1	1	1	1	1	1	0	0	0	1	0	0
MEM. ADDRESS GENERATOR (A Array)	7	2	1	2	3	-	4	5	4	5	-	6	-	-	-
PARALLEL MEMORY OUTPUT	A(31)	A(7)	A(3)	A(9)	A(15)	-	A(17)	A(23)	A(9)	A(25)	-	A(27)	-	-	-
ALIGNMENT I OUTPUT	A(3)	A(15)	A(7)	A(9)	A(31)	A(23)	A(25)	A(17)	A(19)	-	-	-	A(27)	-	-
MEM. ADDRESS GENERATOR (B Array)	8	9	8	8	10	10	10	9	9	-	-	-	10	-	-
PARALLEL MEMORY OUTPUT	B(4)	B(10)	B(6)	B(7)	B(18)	B(14)	B(15)	B(11)	B(12)	-	-	-	B(16)	-	-
ALIGNMENT I OUTPUT	B(4)	B(10)	B(6)	B(7)	B(18)	B(14)	B(15)	B(11)	B(12)	-	-	-	B(16)	-	-
PARALLEL PROCESSOR OUTPUT	C(1)	C(7)	C(3)	C(4)	C(15)	C(11)	C(12)	C(8)	C(9)	-	-	-	C(13)	-	-
ALIGNMENT II OUTPUT	C(7)	C(3)	C(4)	C(15)	C(1)	C(12)	C(8)	C(9)	-	C(11)	-	C(13)	-	-	-
MEM. ADDRESS GENERATOR (C Array)	15	14	14	16	13	16	15	15	-	15	-	16	-	-	-
	First cycle. All memory elements read or written in this cycle are denoted by ○ in Fig. 6.					Second cycle. All memory elements used in this cycle are denoted by □ in Fig. 6.					Third cycle. All memory elements used in this cycle are denoted by △ in Fig. 6.				

Fig. 7. A snapshot for example in Fig. 5

of a_0 and a squeeze of a. Additionally, a shift by the difference in their base memory unit number is required.

These ideas can be clarified in our example. Notice that the A array is stored beginning in unit 1, whereas the B array begins in unit 3. Thus, B must be left-rotated by distance 2 because of its base address, plus 3 because of its subscript (I + 3), for a total of 5, which is precisely the number of memory units. A rotation of distance 5 (mod 5) is no rotation at all.

The A array, on the other hand, requires a left rotation of 1 (mod 5) because of its subscript (but none due to its base address in memory unit 1) and a squeeze of distance 2 (mod 5) because of its subscript. This combination is precisely that between input and output of Alignment Network I. Since pair elements are correctly accessed by the scheme described earlier, they are correctly aligned using methods for dense arrays. More discussion of dense arrays and omega networks can be found in [Lawr75]. It may be observed that the scheme we are using for sparse arrays can be regarded as substituting for one element of a dense array, another desired element that happens to be stored in the same memory unit. For example, A(15) is substituted for A(5) and A(31) is substituted for A(11). Thus, the omega network handles the alignments properly.

6. Remarks and Conclusion

The solution of linear arithmetic recurrences with mode bits will be studied somewhere else. Here it would suffice to make a few comments. Since linear arithmetic recurrences of low order can be processed in time proportional to the log of the serial time, breaking a recurrence into two parts to be processed consecutively could actually slow down a computation. In certain cases, however, breaking up a large recurrence is quite profitable. If a very large number of small recurrences arise, an MES machine (or an MEA machine with very many instruction sequences) could execute each one serially (or using limited processor algorithms [ChKS78]). For register-to-register pipeline processors with vector registers (e.g., CRAY-1), register-contained recurrences are desirable since no memory access is needed other than at the beginning and end. Also, on any machine, remote term recurrences can be speeded up by only computing the final sequence required to obtain the remote terms.

To illustrate the basic idea, consider an R<n,1> recurrence defined by

$$x_i = a_i \, x_{i-1} + b_i \qquad (1 \le i \le n)$$

with appropriate initial conditions. Suppose this appears in a loop with an IF statement, so a mode bit pattern controls its execution. If one mode bit is zero, then this may be computed as two independent recurrences, using an initial value for x in the zero mode bit position. Similarly, if some a_i happens to be zero, the recurrence can be broken into two recurrences.

A new scheme for handling array operations inside DO loops with IF statements has been presented in this paper. The idea of Mode Function Compressor can be easily extended to processing of any type of sparse arrays on a parallel machine. We also gave a new result on solving Boolean recurrences.

Acknowledgment

We thank David Padua for his helpful discussions.

References

[Bane79] U. Banerjee, "Speedup of Ordinary Programs," Ph.D. thesis, University of Illinois at Urbana-Champaign, Dept. of Computer Science Rpt. No. 79-989, Oct. 1979.

[BuKu71] P. Budnik and D. J. Kuck, "The Organization and Use of Parallel Memories," IEEE Trans. on Computers, Vol. C-20, No. 12, pp. 1566-1569, Dec. 1971.

[ChKS78] S. C. Chen, D. J. Kuck and A. H. Sameh, "Practical Parallel Band Triangular System Solvers," ACM Trans. on Mathematical Software, Vol. 4, No. 3, pp. 270-277, Sept. 1978.

[Kuck70] D. J. Kuck, "A Preprocessing High-Speed Memory System," IEEE Trans. on Computers, Vol. C-19, pp. 793-802, Sept. 1970.

[Kuck76] D. J. Kuck, "Parallel Processing of Ordinary Programs," in Advances in Computers, Vol. 15, pp. 119-179; Ed. by M. Rubinoff and M. C. Yovits (Academic Press), 1976.

[Kuck78] D. J. Kuck, The Structure of Computers and Computations, Vol, I, John Wiley & Sons, Inc., NY, 1978.

[KuPa79] D. J. Kuck and D. Padua-Haiek, "High-Speed Multiprocessors and Their Compilers," Proc. of the 1979 Int'l. Conf. on Parallel Processing, pp. 5-16, Aug. 1979.

[LaVo80] D. Lawrie and C. Vora, "The Prime Memory System for Array Access," Proc. of the 1980 Int'l. Conf. on Parallel Processing, Aug. 1980.

[Lawr75] 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.

[Towl76] R. A. Towle, "Control and Data Dependence for Program Transformations," U. of Ill., C. S. Rpt. 76-788, Mar. 1976.

VASTOR: A Microprocessor Based Associative Vector Processor for Small Scale Applications*

W.M. Loucks, W.M. Snelgrove and S.G. Zaky

Dept. of Electrical Engineering,
University of Toronto

ABSTRACT

A word-parallel, bit-serial associative processor built around an array of 1-bit wide microprocessors is introduced. It is intended as a low-cost auxiliary processor in small scale computer systems. Data are organized in an array of fixed number of elements, variable word-length vectors. Processing proceeds in parallel on all elements of a vector. Information about the location and word-length of these vectors is stored in a small general-purpose computer which is used to control the storage and processing array.

I. INTRODUCTION

The parallel processing capabilities of an associative processor are highly attractive in many non-numeric applications. Operations such as searching and sorting are inherently parallel in nature, since they may be regarded as a sequence of basic operations such as compare, shift, and mark performed in parallel on a large number of operands. Many organizations have been proposed for associative processors [8, 10]. Of these, the word-parallel, bit-serial, or vertical [9], organization has received considerable attention. This is due to the fact that the bit-serial organization leads to a considerable simplification of the hardware in comparison with fully parallel schemes.

Because of the hardware intensive nature of associative processors, they tend to be economically viable only in large, high capital cost systems. The purpose of this paper is to introduce an associative processor that is meant for relatively small applications. It is based on an array of commercially available 1-bit wide microprocessors. Machine organization is word-parallel, bit-serial. Data is stored and processed in the form of vectors consisting of a fixed number of elements. The machine has been dubbed VASTOR for Vector Associative Store TORonto.

VASTOR is intended as a special purpose processor to be attached to a conventional mini-computer system. In what follows, the minicomputer will be referred to as the host. In such a system, VASTOR would handle those parts of the work load that can benefit from its associative and vector capabilities. Use of associative processors in this manner has been suggested by many authors, e.g. [5]. Also many potential applications have been studied [3]. The main feature of VASTOR is that it represents an associative structure and its implementation that are economically viable in a minicomputer system environment. A prototype processor has been constructed and tested.

The main constraints in the design of VASTOR were low cost and modularity. This required that readily available components be used, that internal communication and control be kept simple, and that VASTOR should not overload the computer to which it is attached. Modularity also meant that backplane interconnections between modules should be kept simple and easily expandable.

The VASTOR processor, figure 1, consists of two main components, namely the processing array and the controller. The processing array contains all the storage and processing elements of VASTOR. The controller translates high level commands received from a scalar machine -the host- into sequences of control signals for the processing array. This paper presents a practical implementation of the array and its controller, and describes input/output transfers between the array and the host computer. Algorithms that may be implemented on vector oriented machines such as VASTOR are readily found in the literature [2, 3 and 7].

II. MACHINE STRUCTURE

The organization of the VASTOR array is illustrated in figures 2 and 3. The storage section in the array is an n-word memory, with a word length of several kilobits. Operations are performed on vectors of data elements, figure 2, when the elements of a given vector occupy the same bit positions in all words. While the number of bits per element is the same for all elements of a given vector, it may vary from one vector to another. A 1-bit wide processing element PE is a part of every word. Shift-register SH provides the main mechanism for data transfer

----

* This work was partially supported by the Natural Sciences and Engineering Research Council of Canada under research grant #A8994

among VASTOR words, as well as between the array and the outside world.

VASTOR's architecture, depicted in figures 2 and 3, has the properties both of an associative processor and of an array processor, in the sense in which those terms are defined in [10]. It is an SIMD machine, as are both of these types (note that opcode lines are shared by all cells in figure 2). Each cell contains a storage element which may be used to mark individual words. The I/O structure enables the host to read from and write to marked words in the memory. This allows VASTOR to be used as a content-addressable memory for the host machine. Each cell also has the ability to perform logical and arithmetic operations on its memory under the control of the mark bit, so that one may operate (in parallel) on all data elements satisfying some arbitrary condition. The above features give VASTOR the properties of an associative processor.

On the other hand, one may leave all words selected and use VASTOR as an array of processors. Its I/O structure allows large quantities of data to be transferred to and from the host machine via the parallel port on the right of figure 2. I/O data transfer rate ranges from 0.5 to 8 Mbit/s, as will be discussed in section V. Each cell C can perform data manipulation operations on one word of the memory M. From this point of view, VASTOR is an array processor. Inter-processor communication within the array enables handling of data organized in the form of a one-dimensional array, hence the word "vector" in the machine's name. Thus associative operations may be seen as a particular case of array processing, in which a preliminary computation is used to select data in certain cells for further processing or output to the host machine.

VASTOR operations are essentially word-parallel, bit-serial. The major differences between VASTOR and other serial machines, e.g. STARAN [10], stem from pragmatic considerations: component cost and backplane complexity. STARAN's memory is multi-dimensional: data may be accessed either by row (horizontally) or column (vertically) of a 256 row by 256 column memory array. These two modes of access involve a relatively complex interconnection network, which is referred to as a "flip network". Such a network is not required in VASTOR.

VASTOR uses 256 conventional 1024 by 1 bit random-access memories, all driven by the same address lines (cf. figure 2). Operations can be performed only on columns of memory. Because of this it is a "vertical" computer similar to that proposed by Shooman [9]. The I/O structure has been designed to compensate for the resulting difficulty in communicating with the "horizontal" host machine.

When the number of elements in a data vector is greater than the number of cells in a column of memory, operations can be carried out on "sub-vectors" of 256 elements each. This compromise exists in Shooman's machine also.

As mentioned earlier, development of the structure of VASTOR has been heavily influenced by interconnection considerations. The array has been designed to use only "daisy-chained" and "bused" connections between circuit boards. This allows new boards to be added at any time to increase the size of the array with minimal modifications to the existing backplane. The structure is also well suited to large-scale integration because of the small number of interconnections required between modules.

The main implication of the above restriction on backplane complexity is that it limits the inter-word and associative facilities that may be used. Hence, inter-word communication is accomplished via a shift-register, which involves a daisy-chain connection between circuit boards for both data and control information. Moreover, a single bused connection common to all words of the array combined with an analogue to digital converter (not shown) are used to provide limited accuracy associative testing.

The structure of VASTOR may be discussed in terms of three separate features: the intra-word storage and computation, the inter-word communication, and the associative testing capabilities. Each of these features is discussed briefly below.

2.1 INTRA-WORD FACILITIES

Figure 4 shows the components of a VASTOR word: two kinds of storage, a 1-bit processor and one bit of a shift register.

The random-access memory referred to in the figure as WK constitutes the 'working store'. Data are taken from this memory and returned to it during computation. A second memory, referred to as BK, for backing store, is a serial memory. Its contents are swapped with the contents of the working store in pages containing 256 bits per word. One more bit of storage is available for each word in its part of the shift-register SH. This may be used for temporary storage of operands. It should be noted that the intra-word facilities can be expanded through the use of the line marked 'B' on the figure.

The 1-bit processing element PE with which VASTOR has been implemented is the Industrial Control Unit - Motorola MC14500B. It performs a limited set of primitive operations on external data and a 1-bit internal accumulator called RR (the result register). Another internal register, output enable or OEN, contains a mask which is used to enable selective write-back into either the working or the backing store. The collection of the OEN registers in all words constitutes the output enable vector.

2.2 INTER-WORD COMMUNICATION

The shifter SH is the primary medium for inter-word communication. It is the only machine feature that defines any order to the words. The shift-register SH is divided into 8-bit segments as shown in figure 5. Each segment of SH has two parallel bidirectional ports A and B. The B port is connected to one "phrase" of eight VASTOR words.The A ports of all segments are connected together to form an 8-bit I/O bus.

Two multiplexers CIRC and SHMODE connect the serial inputs of the segments of SH to any of a number of sources. This allows data transfer between the shifter and VASTOR words to take place in one of the following modes.

1. VASTOR to shifter - parallel mode through the B port: in this mode the source of data may be the processing element PE, the working store WK or the backing store BK.

2. VASTOR to shifter - serial mode through the SI port: in this mode up to eight bits of data may be loaded from any word of a phrase into the shifter segment. This operation takes place in parallel for all phrases.

3. Shifter to VASTOR - parallel mode: VASTOR words may be loaded in parallel from port B of the shifter SH via the processing element PE.

4. Shifter to VASTOR - serial mode: 8 bits of data can be moved serially from a shifter segment to any word in the corresponding phrase. This is accomplished via the combined use of the output enable vector OEN and the ability to circulate data within each of the 8-bit segments of SH.

We should note that in the two serial modes 2 and 4, only one word of each phrase is involved in data transfer. This reduces the parallelism in the array by a factor of eight. However, the serial modes are necessary to simplify byte-oriented data transfer between VASTOR and the host machine, as will be discussed in section V.

2.3 ASSOCIATIVE TESTS

All VASTOR operations may leave a result in register RR of the processing element. Contributions from all RR registers are summed, in an analogue fashion, onto a single line. This is a simple scheme to obtain a limited accuracy estimate of the number of responders S, i.e. the number of words with RR=1. The most useful values for this number are zero, one and more than one. A simple analogue to digital converter is used to extract this information from the analogue sum.

III. EXAMPLES OF VECTOR OPERATIONS

This section presents two examples of vector operations in order to illustrate the capabilities of the VASTOR array. In the first example vector addition is described. The second example deals with an associative search for the largest element of a vector.

Let A and B be two vectors that are resident in the VASTOR array, Figure 6a. It is required to obtain a third vector R which represents the arithmetic sum of A and B. Information regarding the two vectors A and B is stored in a table in the controller. The table stores the relevant parameters for each vector, e.g. starting address in the array, number of elements, number of bits, etc. The ADD operation is initiated by the host computer by sending a high level command specifying the function to be performed and the two operands A and B. It is not necessary for the host computer to specify such details as the addresses of the operands, the number of elements or the element lengths. Operands are identified by means of pointers into the operand table stored in the controller. When the operation is completed, the controller returns to the host the value of the pointer corresponding to the result vector R.

Addition is performed in a bit serial, word parallel manner. The sequence of operations is given in Figure 6b. As indicated in the figure, control of the sequence of operations and address calculations are performed in the controller, while vector operations are performed in the array. The optional masking operation at the beginning of the sequence disables those words of the array for which the mask contains "0"s. This may be needed when the vectors involved contain fewer elements than the number of VASTOR words. The mask used in such operations is set up at the time vectors A and B are created.

An implementation of the binary search algorithm [3] for positive or unsigned integers is given in Figure 6c. In this case the elements of the vector are scanned starting with the MSB. A one-bit wide vector TEMP masks out the words that have been rejected at any stage of the search. The associative sum S is used to determine the first bit position where one element of TEMP contains a "1" while all other elements contain "0"s. At the end of the search TEMP contains "1"(s) in the word(s) containing the largest element(s).

The above examples illustrate the operation of VASTOR on short vectors with all bits contiguous in fields. When there are more elements in a vector than words in the array, the vector may be broken into several subvectors. Each subvector is operated on independently. It is also possible that the elements of a vector may occupy two or more non-contiguous fields in a word. In this case the controller repeats the operations on the different fields of the vector.

IV. THE CONTROLLER

The function of the controller is to reduce the control overhead required from the host machine to drive VASTOR. In order to keep the VASTOR array continuously active, 50 control bits are needed every microsecond. That is, a control bandwidth of 50 bits/microsecond must be supported. This rate exceeds the bandwidth of the entire PDP-11 UNIBUS. Hence, it must be reduced to a level which does not prevent the host from performing operations not related to VASTOR. The controller receives high level commands from the host machine, requiring a much lower control bandwidth. These commands are then translated into the sequences of control signals needed to drive the VASTOR array.

The complexity of the commands that have to be interpreted by the controller is represented by the examples given in section III. In order to support such operations, a hierarchical approach has been adopted. Each level in the hierarchy serves to reduce the bandwidth required from the higher levels. Furthermore, interpretation of high level commands has been made relatively simple because of the use of well defined interfaces between various levels.

The hierarchical approach led to the controller organization shown in Figure 7. It consists of three distinct units. The microcontroller which performs low level looping control operations, the buffer memory which is used as a communications medium, and the microprocessor which is responsible for interpreting high level commands received from the host and for space allocation within the VASTOR array. As such, the microprocessor performs functions similar to that of the "interpreter" in ECAM [1]. The microcontroller corresponds to the iteration control logic in ECAM. The three subsystems of VASTOR's controller are discussed briefly below.

4.1 THE MICROCONTROLLER

The microcontroller UC serves to remove some of the redundancy at its output, the array control lines, in order to reduce the bandwidth required at its input. Its sophistication, and therefore cost, can be selected to provide almost any desired bandwidth at its input. We have chosen to implement a device that executes sequences of microcode stored in an internal Read Only Memory, with primitive branching and looping capability. Input commands to the microcontroller come from a buffer memory M which, in turn, is filled by the microprocessor UP.

Linear microcode sequencing provides a large reduction in the control bandwidth. Hence, it was adopted as the main sequencing mechanism in the microcontroller. The starting address for a given microcode sequence is loaded from the buffer M. Since data can be made to appear in the VASTOR array in fields of consecutive locations, further compression of the control information is obtained with a simple loop counter/index register. This counter is decremented and tested to control microprogram loops. It also serves as an index register to modify the addresses transmitted by the controller to the array memory.

Some further control bandwidth compression is obtained by introducing a data-dependent branch. The associative sum of responders is compared to a reference in the microcode. One of two branch addresses is then selected from the buffer M.

4.2 THE BUFFER MEMORY

The buffer memory is divided into sixteen separate task control blocks. These blocks are filled by the microprocessor and interpreted by the microcontroller. Whenever the microcontroller finishes a task it interrupts the microprocessor to request the address of the next control block. Task control blocks contain up to 26 bytes of information. This includes starting and loop control information for the microcode of the microcontroller. It also includes specifications for the operands in the VASTOR array.

4.3 THE MICROPROCESSOR

Controller algorithms represented by one control block in the buffer memory take from 1 to several hundred microseconds to complete and to interrupt the microprocessor. These interrupts are usually quite simple to service but would be uneconomically frequent for the host machine. The microprocessor is therefore included to provide further compression of the control bandwidth. It simplifies the interfacing software by translating high-level operations into sequences of microcontroller tasks.

In addition to sequencing control, the microprocessor performs the storage management function. This includes allocating and freeing fields of storage, garbage collection, paging variables into the working store from the backing store, allowing the widths of elements (e.g. integers) to expand and contract, and segmenting vectors longer than the VASTOR array into manageable components.

V. INPUT/OUTPUT

Data transfer between VASTOR and the host machine is generally difficult because of the incompatibility of the addressable units in the two machines. While a host machine generally obtains all bits of a single element of a vector with one reference to its memory, VASTOR obtains one bit of each element. The transposition required to match the two machines is the source of the difficulty.

The simplest type of vector to transfer is a boolean vector, which is only one bit wide, figure 8a. In order to transfer such a vector from the host into the VASTOR array, its elements may be shifted serially by bit into the shift register SH. This is followed by a transfer from SH to a column of WK using the parallel mode (mode 3, section 2.2). If elements of the boolean vector are packed into bytes in the host machine, as is the case in some versions of APL, shift register SH may be loaded serially by byte through its 'A' port. In the current implementation, data rates for the bit-serial and byte-serial modes are 1 Mbit/s and 1 Mbyte/s respectively.

Consider now the case where data is presented to VASTOR so that some number of consecutive bits must be loaded into a single word, figure 8b. This may be achieved by first loading register RR of the ICU from the CONST line, figure 4, and then storing the content of RR in the enabled word. Due to that two-step sequence and the fact that only one word is enabled at a time, the transfer rate is limited to 500 Kbits/s.

The phrase structure may be used to increase the transfer rate of byte-organized data, as shown in figure 8c. This corresponds to mode 4 of section 2.2. The data rate achievable in this case is 2.5 Mbits/s. In this approach consecutive words from the host machine are not loaded into consecutive words of VASTOR. Rather, they are loaded into the same relative positions in consecutive phrases. A sentence structure consisting of two phrases per sentence also exists and may be used for 16-bit wide I/O transfers. The detailed procedure is given in reference [6].

VI. PERFORMANCE IN APPLICATION AREAS

This section discusses potential applications of a VASTOR processor. The primary application of VASTOR is as an auxiliary processor in a minicomputer system. In this case, it would serve to enhance the performance of the system in vector and associative operations. A second, and equally important, potential application derives from the fact that VASTOR can be regarded as a collection of 1-bit wide controllers driven in parallel by a host computer. Each of these two application areas is discussed briefly below.

Table 1.

Performance Comparison

Between VASTOR and a PDP-11/45
with Bipolar Memory in Vector Operations Involving
256-Element Vectors, with 16 Bits per Element.

Operation	Result	VASTOR Execution Time Microseconds	PDP-11/45 Execution Time Microseconds
Compare	Vector	4 us/bit * 16 bits = 64	3.225 us/word * 256 words = 825.6
Addition	Vector	10 us/bit * 16 bits = 160	1.9 us/word * 256 words = 486.4
Mark Largest Element	Vector	3 us/bit * 16 bits = 48	2.5 us/word * 256 words = 640
Compare to Scalar	Vector	3 us/bit * 16 bits = 48	2.5 us/word * 256 words = 640
Sum Reduction	Scalar	336 us/bit * 16 bits = 5376	1.5 us/word * 256 words = 384

Vector and associative operations are performed quite frequently in the operating system software of a computer. Symbol table manipulation and file management are two such examples. Also, some computer languages, such as APL and SNOBOL, are based upon the organization and manipulation of data in the form of vectors [4] or character strings [7]. A VASTOR processor is ideally suited to such tasks, and hence can take a considerable load off its host computer. Table 1 gives an estimate of VASTOR's performance in this area. The table gives execution times for a number of operations on 256-element vectors, where each element is 16 bits wide. These times are based on the current implementation using a processing element, the ICU, which runs at a 1 microsecond cycle time. For comparison, the times required to perform the same operations in a PDP-11/45 minicomputer are given. As can be seen from the data in Table 1, VASTOR is an order of magnitude faster than a PDP-11/45 when executing tasks that involve parallel operations on all elements of a vector. However, operations such as sum reduction (adding all elements of a vector) take much more time. In this case, VASTOR's performance is limited by its inter-word communication facilities. However, when dealing with much longer vectors VASTOR's performance on sum reduction approaches its performance on vector addition. This is due to the fact that many elements of the vector would be stored in the same word of the array.

At the present stage of development of the VASTOR processor, it is very difficult to obtain an accurate estimate of the gain in performance that would result from adding a VASTOR processor to a minicomputer system. While the data in Table 1 indicate that considerable gain can be realized, this gain will be partially offset by the overhead resulting from transferring data between VASTOR and its host computer. This overhead is expected to be of the same order as that involved in transferring data between the main memory of a computer and a disk file. Therefore, VASTOR is most suited for use in applications where a number of vector operations have to be performed before a given vector is transferred back to the host machine.

Stand-alone ICU's have applications in process control and monitoring. VASTOR may be used in situations where a number of ICU's performing similar tasks are to be interfaced to a common host computer. In this case, VASTOR represents an organized way of performing I/O and control functions. Each ICU is capable of sampling data from and controlling an external device at data rates of the order of a few kilohertz. Status information and data such as minimum values, maximum values, averages, setpoints and enabling bits for each device may be kept in the corresponding working storage. The main limitation to this approach is that it is necessary to synchronize data transfer between the ICU's and the various devices.

VII. CONCLUSIONS

The VASTOR processor presented in this paper represents a trade-off between the capabilities and cost of the inter-word communication facilities in an associative processor. The result of this trade-off is a processor that allows a nontrivial associative processing capability to be incorporated in small scale minicomputer systems. The communication hardware provided in the VASTOR array enables data transfer among the words in the array without requiring costly and complicated hardware. It also results in simple backplane interconnections between different modules. The modular structure of VASTOR allows its capabilities to be expanded easily and economically.

Some of the limitations of the current implementation of VASTOR are due to the slow speed of the processing element used (the ICU). A faster and more powerful 1-bit wide processing element can lead to a considerable increase in performance without the need for any changes to the architecture. In fact, because of the low number of interconnections involved, the structure is well suited to integration. Some of the possibilities would be the implememtation of an array of 1-bit processors, or processors and memory on a single chip. Another possibility which is currently being investigated by the authors is the use of a table driven processing element made of memory only. Some other limitations of VASTOR, such as the difficulty of reordering a vector, are more fundamental. In order to perform such operations at high speed, a more complex, and hence more costly, inter-word communication scheme must be provided.

REFERENCES

1. Anderson, G.A., and Kain, R.Y., "A Content-Addressed Memory Designed for Data Base Applications", Proc. 1976 International Conf. on Parallel Processing, IEEE, New York, 1976, pp. 191-195.

2. Baudet, G. and Stevenson, D., "Optimal Sorting Algorithms for Parallel Computers", IEEE Trans. Comput., vol. C-27, pp. 84-87, Jan. 1978

3. Foster, C.C., Content Addressable Parallel Processors Van Nostrand Reinhold Co., New York, NY, 1976.

4. Grey, L.D. A Course in APL\360 with Applications, Addison-Wesley Publishing Co., Reading, Mass., 1973

5. Kaplan, A., "A Search Memory Subsystem for a General-Purpose Computer", Proc. AFIPS 1963 Fall Jt. Comp. Conf., Vol. 24, Spartan Books, Inc., Baltimore, Md., 1963, pp 193-200.

6. Loucks, W.M. and Snelgrove, W.M., "VASTOR 1978", Univ. Toronto Computer Engineering Report 13, June 1978.

7. Mukhophadhyay A., "Hardware Algorithms for Nonnumeric Computation", Proc. 5th Ann. Symp. Comp. Arch., April 1978, Palo Alto CA., pp. 8-16.

8. Parhami, B., "Associative Memories and Processors: An Overview and Selected Bibliography", Proc. IEEE, Vol. 61, pp. 722-730, June 1973.

9. Shooman, W., "Parallel Computing with Vertical Data", Proc. 1960 Eastern Jt. Comp. Conf., Eastern Jt.:c Computer Conf. 1960, pp 111-115.

10. Yau, S.S. and Fung, H.S. "Associative Processor Architecture - A Survey", ACM Computing Surveys, Vol 9, No. 1, pp. 3-27, March 1977.

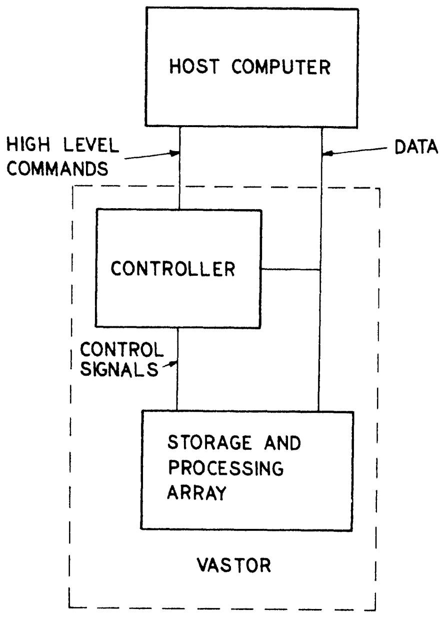


Fig. 1. The VASTOR processor

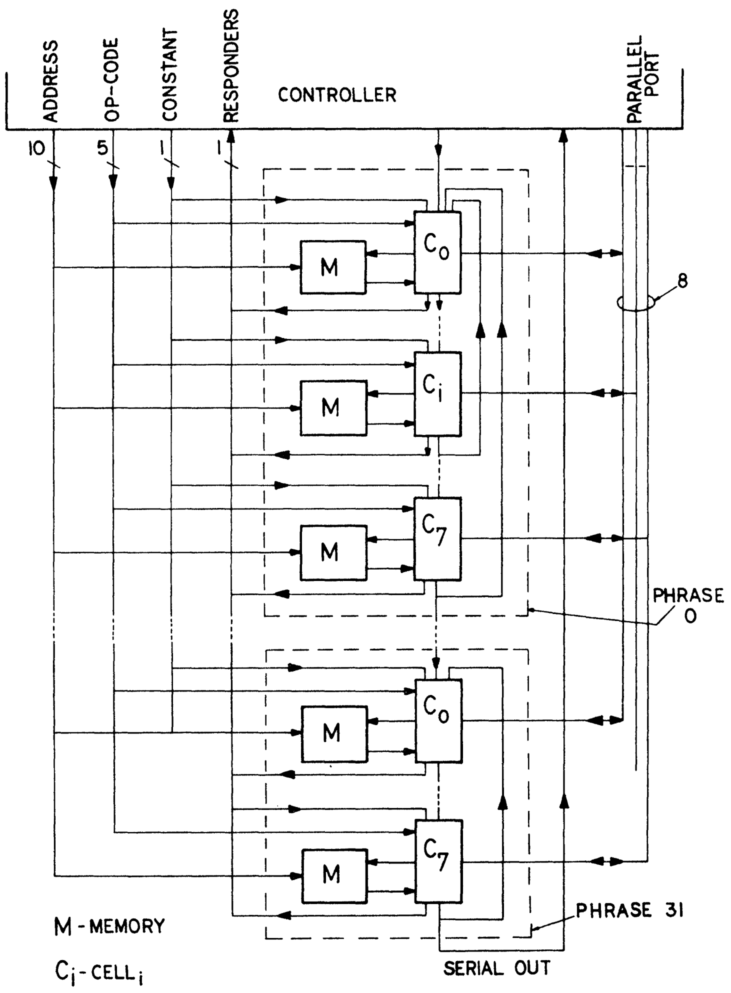


Fig. 2. Control and data paths

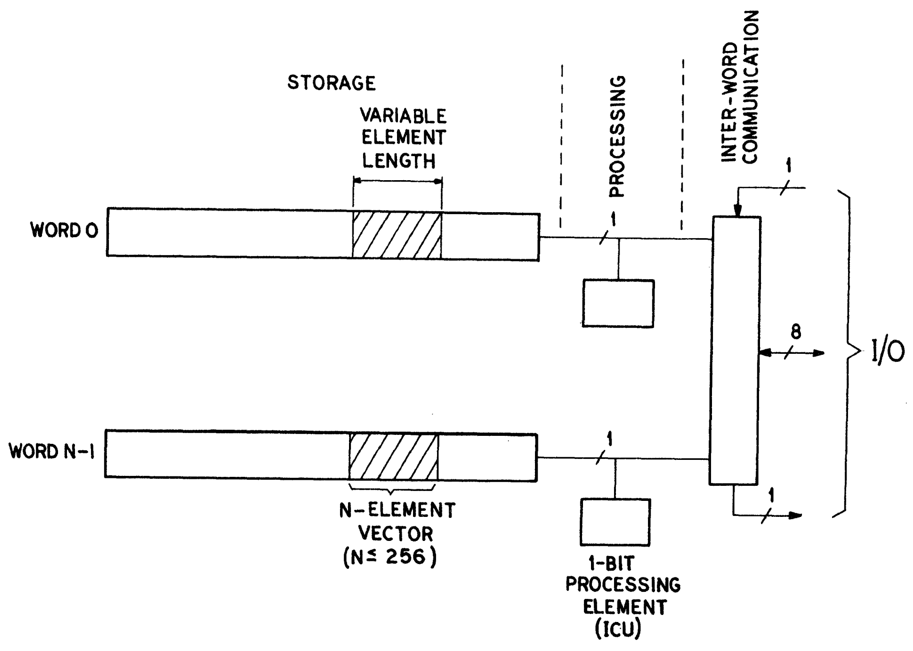


Fig. 3. Organization of the VASTOR ARRAY

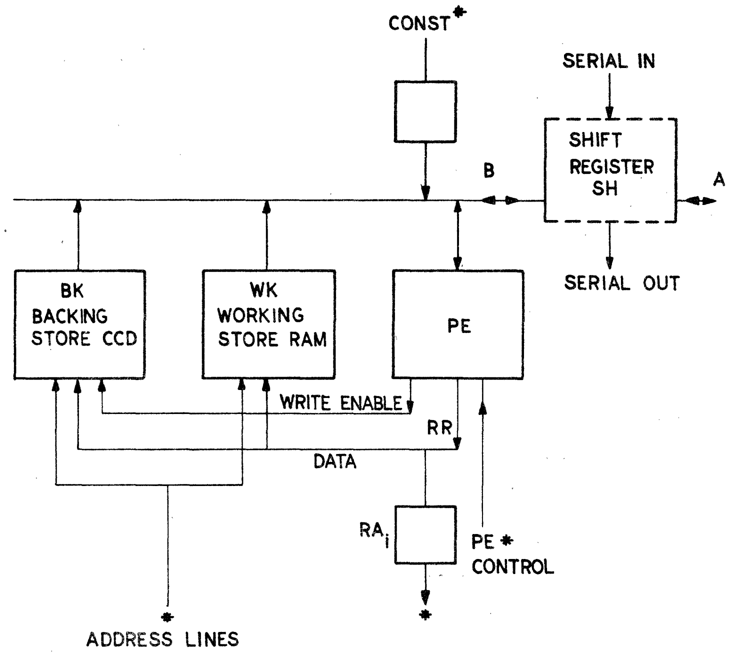


Fig. 4. One word of the storage and processing array

* COMMON TO ALL ARRAY WORDS

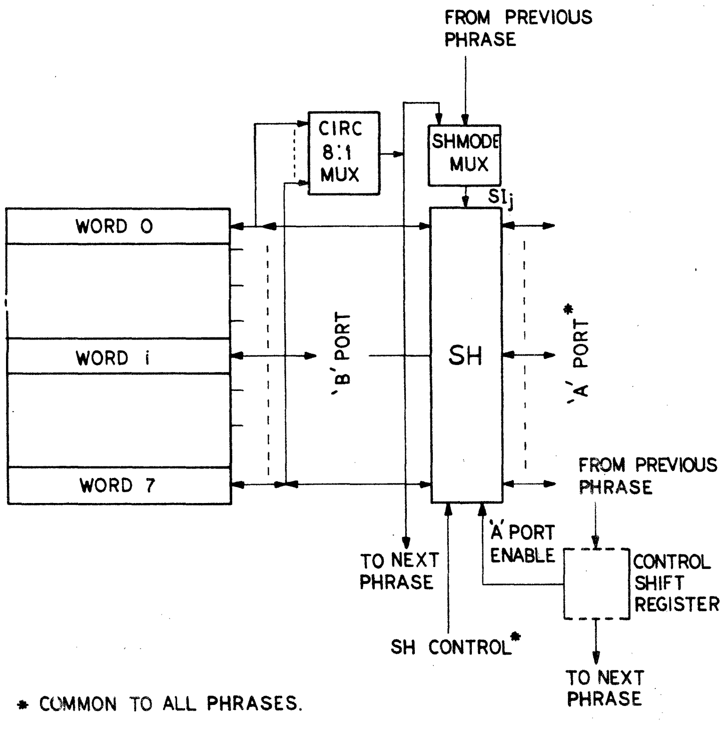


Fig. 5. The phrase structure

* COMMON TO ALL PHRASES.

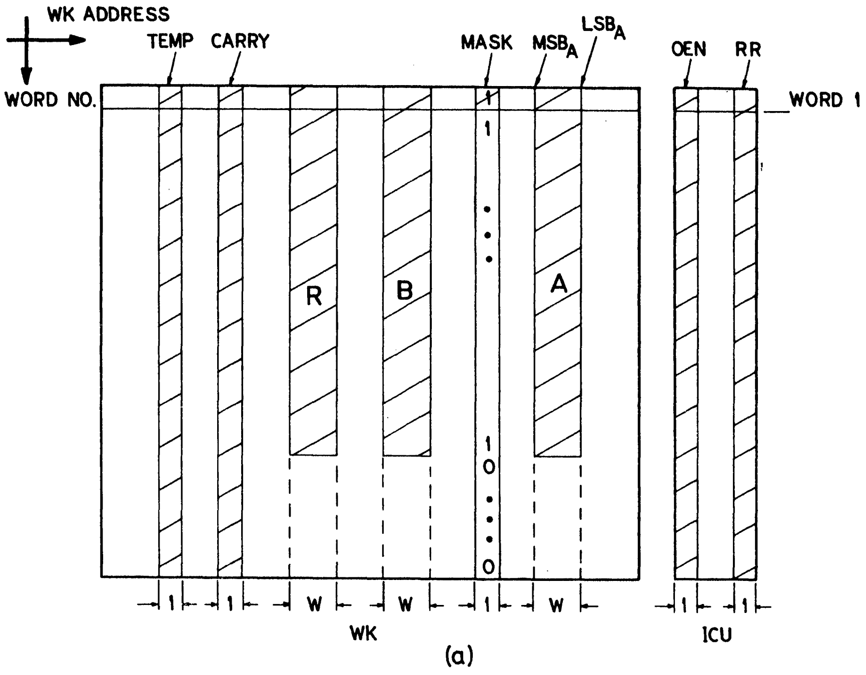


Fig. 6a. Vector addition example

```
CLEAR C                                ; Clear carry vector - array operation (optional)
OEN ← MASK                             ; Vector array operation
FOR   i = 0, W - 1                     ; Controller operation
      ADDRESS ($A_i$) = $LSB_A$ + i    ; Address calculation - controller operation
      ADDRESS ($B_i$) = $LSB_B$ + i    ; Address calculation - controller operation
      ADDRESS ($R_i$) = $LSB_R$ + i    ; Address calculation - controller operation
      ADDRESS (C)  = $LSB_C$           ; Address calculation - controller operation

      $R_i \leftarrow A_i \vee B_i \vee C$                            ; Vector array operation

      $C \leftarrow A_i \wedge B_i \vee A_i \wedge C \vee B_i \wedge C$       ; Vector array operation
```

Fig. 6b. Implementation of vector addition

```
TEMP ← MASK                            ; Vector array operation
FOR   i = 0, W - 1                     ; Controller operation
      ADDRESS ($A_i$) = $MSB_A$ - i    ; Address calculation - Controller operation
      RR ← TEMP.$A_i$                  ; Vector array operation
      IF (S = 1)                       ; Controller operation
                EXIT                   ; Controller operation
      ELSE IF ($S \neq 0$)             ; Controller operation
                TEMP ← RR              ; Vector array operation
```

Fig. 6c. Search for the largest element

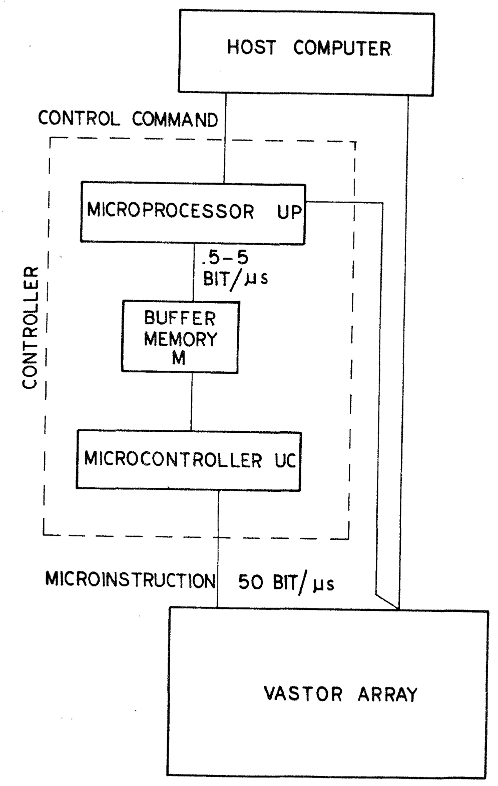


Fig. 7. Controller hierarchy

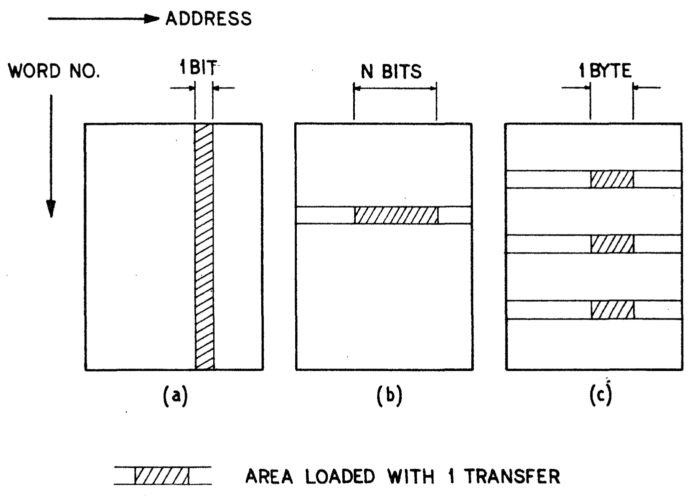


Fig. 8. Alternative modes for input/output transfers

An Outline of the Computer System with Associative Pipelining

Simon Ya. Berkovich

Department of Electrical Engineering and Computer Science
The George Washington University
Washington, D.C. 20052, USA

Summary

The fundamental ways for increasing the productivity of computer systems are parallelism and pipelining. In both cases for the sake of efficiency the computing processes should be decomposed into possibly small and uniform parts. The most appropriate elementary computing operations from this point of view are provided by a fully parallel word-organized associative processor [1]. Unfortunately, the successful application of the associative processors comes across two limitations: the implementation of such devices of sufficiently large scale is rather difficult and the necessity to make supplementary moves of data in and out of the working area cut down the gain in their fast processing.

In this work we consider a new type of computer system - dual to the associative processor. Its main component is a homogenous array of cells [2], which realizes pipelining transformations in space, isomorphic to parallel transformations realized by the associative processor in time (fig. 1). The algorithms of the associative processing are based on the alternation of two types of commands: (1) Φ - the isolation of the subset of words having a given indicator and (2) A - the multiwriting of given codes simultaneously in certain digits of all the words of the isolated subset. The program in (Φ-A) form for the processor controls the pipeline elements as well. The data are processed during transmission and the number of the pipeline elements is equal to the length of the program rather than to the amount of these data. The above-mentioned limitations on the size of the device and speed of the computing process fall away, and it gives fresh impetus to the application of the long and well developed theory of associative processing.

The principle of associative pipelining can be applied to different types of computers from relatively small specialized devices to very large data processing systems. The computing process can be constructed as a succession of the uniformly organized data transmissions; if the program is longer than the available pipeline length, the processing can be arbitrarily divided into successive steps. A general purpose architecture is shown in fig. 2.

The pivotal part of the computing system is the associative pipeline in the form of a closed curve to decrease possible losses due to fragmentation. Information storage is spread over a number of some devices with cyclic access, which are called <u>DLS</u> - "Drum-Like Storage," because a drum presents a clear view of word stream supply. Main functions of the <u>control processor</u> are the presentation of control <u>programs in (Φ-A)</u> form and the dynamic allocation of the DLS and pipeline resources. The <u>switching circuit</u> establishes the necessary paths <u>between DLS and output units</u> through some segments of the <u>pipeline and directing interfaces</u>. The control program can be <u>sent to such a path</u> essentially simultaneously with the data stream.

In the framework of this architecture it is simple to achieve multiprogramming facilities by an interleaving technique for data transmissions. The solution of the concurrency problems can be organized in such a way that as soon as some information starts out to transfer from one DLS to another DLS, all the requests to the former should be reassigned to the latter, and the access to the updated information will be available right away, before the whole process of updating will be completed.

Associative transformations of isolated words should be extended to some operations concerning their collective properties. These operations can be applied to sets of short words considered as long-word packets, and to data collection as a whole for sorting, eliminating duplicates, max/min and so on. It is more easy to provide such facilities for the associative pipeline than for the associative processor, because the processor requires extra circuitry in bulk, while the pipeline needs only some additional equipment for its individual devices - directing interfaces and output units.

The pipeline operations are efficient for manipulating with different types of information structures, especially in a table form. They may be used in sublanguages based on relational algebra as SEQUEL. Associative pipelining is adjustable for most reasonable table functions as MAX, MIN, COUNT, TOTAL and for transformation operators like SELECTION, PROJECTION, DIVISION, and JOIN. The computer system with associative pipelining is beneficial for inverted file directories, which can be organized by presenting the keys of records in packet form. The access may be accelerated by an order of magnitude and even more. Associative pipelining provides not only all necessary information, corresponding to simple key matching, but more complex searching criteria, including logical functions and partial name matching can be accomplished in the same time.

The most crucial question for system applica-

CH1569-3/80/0000-0047$00.75 © 1980 IEEE

tions is the pipeline length, i.e., the number of (Φ-A) elements to be implemented. Estimates show that one such element with word length - r about 40 bits should contain approximately ~$0.5 \cdot 10^3$ gates. A moderate system of about 10^5 logic circuits may present a pipeline with ~200 elements. This is fairly enough for most information retrieval procedures, for which are typical the algorithms with O(r) number of (Φ-A) elements. A larger system on the order of ~10^6 logic circuits may present a pipeline with ~2,000 elements. Such systems may be used for computational problems for some kind of algorithms with $O(r^2)$ number of (Φ-A) elements.

The idea of associative pipelining is in accord with data-flow concept [3]. The advantages of this approach are hardware/software uniformity, high speed, ease of operational control and multi-access using a communication computer. This system naturally integrates into network environment. It is worthwhile to notice that a reply processing can be initialized before the completion of the request, when it is in (Φ-A) form. Because any algorithm can be realized at the rate of word transmission with no bottle-neck situations, associative pipelining is appropriate for code conversion by sendings and receivings of data, e.g. for any kind of encryption and error correction, and for rather more complicated real-time signal processing.

Associative processors are known to be useful for different parallel algorithms, but their most powerful applications are in information retrieval. The associative pipeline as a dual structure has the similar properties too. Hence, this computer system may be in particular considered as a sort of a database machine [4]. In this case, a unified approach to different types of information systems can be developed, including features of information retrieval and database management.

References

[1] K.J. Thurber and L.D. Wald, "Associative and Parallel Processors," ACM Computing Surveys (December 1975), pp. 215-255.

[2] S.Ya. Berkovich, Yu.Ya. Kochin and G.M. Lapir, "Algorithms for Group Word Processing in an Associative Memory and their Realization by Homogenous Computing Arrays," Automation and Remote Control, (August 1974), pp. 1342-1349.

[3] P.J. Denning, "Operating Systems Principles for Data Flow Networks," Computer (July 1978), pp. 88-96.

[4] G.G. Langdon, Jr., Guest Editor, "Special Issue on Database Machines," IEEE Trans. on Computers (June 1979).

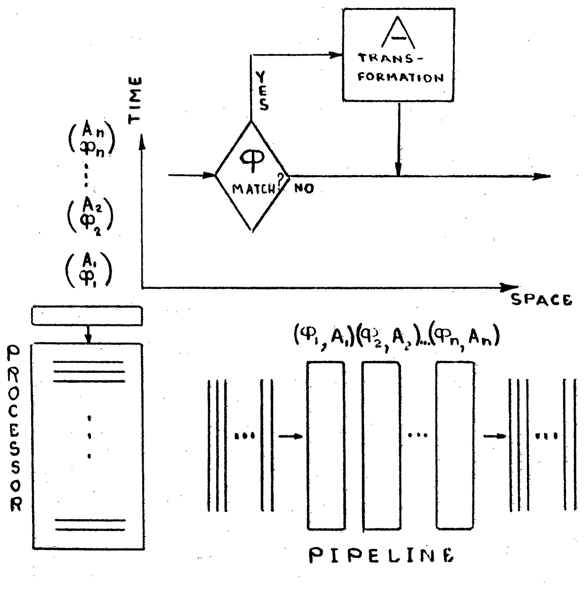


Fig. 1

ASSOCIATIVE PROCESSOR - PIPELINE DUALITY

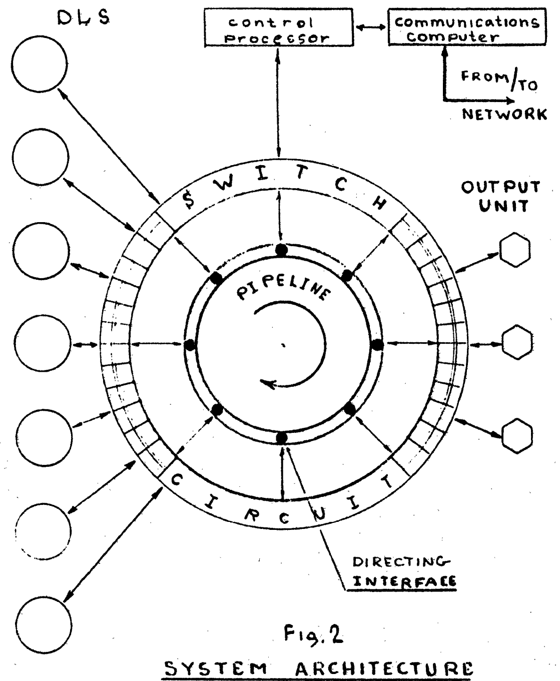


Fig. 2

SYSTEM ARCHITECTURE

FRAMEWORK FOR COMMUNICATION IN LOOSELY COUPLED MULTIPLE PROCESSOR SYSTEMS

VASON P. SRINI

COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF SOUTHWESTERN LOUISIANA
LAFAYETTE, LA 70504

The problem of communication between processes in multiple processor systems is addressed. Three high level communication mechanisms are presented. The first mechanism is based on sequential processes consisting of modules, procedures and processes that communicate via procedure calls and input/output statements. The second mechanism is based on message passing consisting of modules with conditional send message and receive message primitives. The third mechanism is based on structured message-passing consisting of blocks that receive messages at the beginning of a block and send messages at the end of a block. Programming language constructs for supporting each of the three mechanisms are outlined. The structured message-passing approach (or abstract dataflow approach) has features that facilitate automatic scheduling of blocks to processors, brings out all parallelism at the block level, facilitates synchronization without using semaphores, and facilitates a design approach using abstractions and refinement.

I. INTRODUCTION

Most of the multiple processor systems that have been developed during the past several years can be divided into three categories:

a. Tightly coupled systems such as multiprocessors with shared memory or a shared bus (e.g. C.mmp , UC Berkeley's PRIME, Burroughs 6700/7700, and PLURIBUS).
b. Loosely coupled systems such as distributed systems, and systems which communicate by passing messages (e.g. HP's 3000, IBM's 8100).
c. Networks of computers (e.g. ARPANET, ALOHANET, Ethernet).

The development of each of the above systems has required significant software development and maintenance. Since software is more expensive than hardware or firmware, the implementation of the strategies and policies used in the above systems is not possible in inexpensive multiple processor systems. In this paper, an outline of strategies for interprocess communication in multiple processor systems is presented. A detailed discussion is presented in [1,2].

This work is supported in part by TI Inc., Corporate Engineering Center, Dallas, Tx.

STRATEGIES FOR INTERPROCESS COMMUNICATION

A fundamental concept useful in loosely coupled multiple processor systems is the distributed process, dp. A dp is a collection of blocks, called dp_blocks. The dp_blocks communicate either by using messages or by sharing data structures. The details of dp_blocks are discussed later on. A dp consists of the following:

1. Its own address space.
2. Its own resource environment.
3. A list of all other dp's it can access (capability list) and a list of other dp's that can access it (access list).

All communications between dp's take place using relatively short messages. The code and data associated with a dp is stored in the address space. The address space of any dp is symbolic (e.g. a collection of named objects). The code of a dp is executed using the data in the address space and the resources available in the environment. The resource environment of a dp provides the runtime support for the dp. It contains standard library programs, a runtime stack(s) for supporting activation records, a heap(s) for supporting storage needs, a processor(s), virtual or real devices, and special functional units such as a floating point processor, and a FFT processor.

REQUIREMENTS OF INTERPROCESS COMMUNICATION

One of the major areas to be addressed in the support for dp's is the interprocess communication. The following dp support requirements are noted:

a. Ability to share large data structures and to communicate short messages.

b. Ability to block requests on various conditions.
c. Ability to refuse requests for resources.
d. Facility to employ different strategies in accessing resources.
e. Facility to handle local and system exception conditions.
f. Facility to prevent deadlocks.

Asynchronously executing dp's can communicate using three distinct approaches. The first approach is based on the procedure call or the use of monitors [3]. In this approach, a program is partitioned into process,es by the programmer. In each process, the programmer makes decisions regarding the sequence of statements. An OMODULE construct is introduced to realize the dp concept. The OMODULE consists of a collection of PROCESSs, MODULEs, IMODULEs for shared objects, DMODULEs for device or control dependent activities, procedures, initialization part, and a module body. The constructs MODULE, IMODULE, DMODULE, PROCESS, and procedure represent the dp_block. Each MODULE consists of a collection of procedures, MODULEs, an initialization part, and a body. A MODULE establishes a scope rule for its local variables. An IMODULE is the extension of the interface module in MODULA [4]. It encapsulates shared objects and operations allowed on these objects. An IMODULE consists of a collection of procedures, one or more DMODULEs, an initialization part, and a body. Nesting of IMODULEs is not allowed. A DMODULE is an extension of device modules in MODULA. The syntax of the above constructs and the informal semantics of the constructs are shown in [2].

The second communication strategy is based on message passing. A program is partitioned into modules by the programmer. In each module, the programmer makes decisions regarding the sequence of statements. Communication between modules is by sending messages and receiving messages. This approach to communication avoids the delay inherent in procedure calls when the called procedure cannot be entered.

The third communication strategy is a structured message-passing approach based on dataflow with high level primitives [1,2]. The highlights of the third strategy are shown in the next section.

STRUCTURED MESSAGE-PASSING

The structured message-passing approach to communication between asynchronous processes uses principles of dataflow [5]. Basic dataflow has been used in computer systems organization and in the specification of algorithms. All activities that can be performed in parallel are expressed as nodes without data dependencies. Activities are sequenced only when there is data dependency. One problem with using basic dataflow is that the resulting graphs are complex, containing all the details. Another problem is the rather restricted set of primitives. Basic dataflow has been extended so that users can define operations suitable to their applications. Each of these operations is a procedure in a highlevel language. This abstract dataflow approach is supported by a dataflow simulator [6,7]. Dataflow programs using basic dataflow primitives and user defined operations can be run on the dataflow simulator. This has opened up several possiblities in analyzing algorithms for parallelism and functional partitioning of programs.

Each node in abstract dataflow waits for the arrival of tokens on the required input arcs. If space is available on the output arcs for tokens, then the node can be enabled for firing. Thus, communication between nodes is accomplished using tokens on arcs. Tokens can be thought of as messages and specified arcs as data paths. This communication facility is different from the message passing approach in several aspects:

a. Using a standard firing rule, once a node starts firing, it cannot be interrupted by other nodes sending tokens to it and the node cannot wait for tokens from other nodes.
b. Using a nonstandard firing rule, a node starts firing when tokens arrive on a specified subset of input arcs and continues to accept token(s) on a specified subset of input arcs.
c. All tokens generated by a node are sent as output on the designated arcs either at the end of firing, if a standard firing rule is used, or during firing if a nonstandard firing rule is used.

There are several advantages to the above mentioned communication facility:

a. The communication mechanism for each node is the same.
b. Each node has a specified set of output arcs for sending tokens to other nodes and a specified set of input arcs for receiving tokens from other nodes.
c. The communication structure is regular and comprehensible. The resulting program is well structured.
d. Synchronization is achieved by using enabling conditions which

require at least one token on each of the required input arcs. There is no need for semaphore variables and P and V operations on semaphores.

In the structured message_passing approach, a program is an abstract dataflow graph (ADG). Each ADG is a labelled and directed graph which is an interconnection of subgraphs. Each subgraph consists of nodes which are interconnected by arcs. Each node and arc has a number of attributes. Some of the attributes of a node are label, operation, and input/output (I/O) arc specification. The label of a node is a unique identifier for the node. The operation attribute specifies the semantics associated with the node. The I/O arc specification attribute specifies the input arcs and output arcs of the node. Each I/O arc specification of a node has a set of conditions that must be met before the node can be fired. can be fired. This set of conditions is called the enabling condition and is represented as a set, called the firing semantics set (FSS). Some of the attributes of an arc are label, token type that the arc can carry, and arc capacity.

A simple example is shown in Figure 1. The READER reads a job, copies it into an empty buffer, and outputs a filled buffer to PROCESS_JOB. The PROCESS_JOB manipulates and fills an empty buffer with its results. The buffer received from READER is returned as an empty buffer to the BUFFER_POOL_MANAGER. The WRITER receives the filled buffer from PROCESSOR, outputs the contents of the buffer, and returns the empty buffer to the BUFFER_POOL_MANAGER. The node BUFFER_POOL_MANAGER in turn removes an empty buffer from one of its input arcs and outputs the empty buffer on one of its output arcs using the policy specified in the semantics of BUFFER_POOL_MANAGER. Initially, empty buffers are on the arc EMPTY_BUFFERS.

A detailed discussion on ADG, extensions to ADG, and several examples are shown in [9]. In order to use the structured message-passing approach in multiple processor systems, we need either an environment supporting the execution of ADGs or a high level language with constructs for representing nodes, arcs, and tokens. In this paper, the high level language approach is pursued. We propose a construct for representing nodes. This construct is called a dp_block. We now draw an analogy between the ADG and the dp concept, and describe the details of dp_blocks.

If we treat a dp as analogous to the execution of a subgraph in an ADG, the dp_blocks are analogous to the nodes in an ADG. Since any number of nodes can be fired in parallel depending on the FSS and the availability of data, any number of dp_blocks in a dp can potentially be executed simultaneously. The syntax of this construct is shown in Table 1.

The name of the dp_block corresponds to the name of the operation assigned to a node. The label of the dp_block corresponds to the node label. The input arc descriptions correspond to the description of all arcs incident to the node. An arc description consists of arc name, arc capacity, label of the source block, and the type of token the arc can carry. The condition part corresponds to an FSS and specifies the set of input arc names that must have at least one token in order to enable the node for firing. Those arcs that can receive a token(s) during the firing of the node are specified in INPUT. The condition part can be a Pascal IF statement using any of the input arc names or constants, or logical operators such as AND, OR, or NOT. If the condition part is absent, then a token must be present, on each of the input arcs in order to execute the block and the block should not contain any arc names in INPUT. The body of the dp_block represents the node semantics. The constant, type, and local variable declarations have the usual Pascal syntax and semantics. The statements in the block can be any of the executable Pascal statements except procedure calls, and blocks.

Assignment statements in a dp_block use the single assignment rule which is similar to those rules proposed by Tessler and Chamberlin [10].

The result of a node's firing is provided by the last line in the block. If values have been calculated in the block for the output arc names, then these values are sent as output by the last line in the block provided the condition part is true. Outputs can also be sent during the node firing. The output arcs that receive tokens in this manner are shown in OUTPUT.

PROGRAM DECOMPOSITION

Since a program is a directed and labelled ADG using nodes and arcs with an operation assigned to each node, this operation can be the name of the ADG. Such a node is called a recursion node, and the graph is called a recursive graph. Each recursive graph is denoted as a dp. Each invocation of a recursive graph has a distinct graph color. This color is used by all the tokens in the invocation of the graph. All the invocations of a recursive graph can reside in one dp address space, use the dp environment, and have the capabilities of the dp. Each invocation can also use a distinct copy of the dp's address space, environment, and capabilities.

There is a special kind of operation, called APPLY, that can be assigned to a node [6,7]. This operation builds a graph dynamically using tokens on input arcs. Each APPLY node is denoted as a dp. A nonrecursive graph can be partitioned into subgraphs using cluster detection algorithms [11] or by computing cut sets [12,13], satisfying a given objective function. The arcs in the cut set represent the data paths for communication between subgraphs. Each subgraph is denoted as a dp. If a graph contains nodes representing recursive subgraphs or nodes with the operation APPLY, then each such node is denoted as a dp. Nodes that have not been denoted as dps are analyzed in the above manner to identify all dps. In other words, program decomposition can be performed algorithmically.

ACKNOWLEDGEMENTS

The author is thankful to Drs. Bruce Shriver and Pat Carr, Steve Landry, Daren Appelt, Doug Johnson, Laurence Brevard, the members of Project Kappa for stimulating discussions, and to Mary Davis for preparing the manuscript.

REFERENCES

1. V.P.Srini,"Framework for communication in loosely coupled multiple computer systems", Technical Report,TR-80-3-3, Computer Science Dept., University of Southwestern Louisiana, Lafayette, LA 70504, March 1980.
2. V.P.Srini and B.D.Shriver, "Abstract dataflow protocol for communication in distributed systems", Proceedings of Compcon, Sept. 1980.
3. C.A.R.Hoare, "Towards a theory of parallel programming", Operating Systems Techniques, Academic Press, New York, 1972.
4. N.Wirth, "MODULA: A language for Modular Multiprogramming", Software - Practice and Experience, Vol. 7, No. 1, Jan. 1977, pp 3 - 35.
5. J.B.Dennis, and J.B.Fosseen, "Introduction to data flow schemas", Computation Structures Group Memo: 81, Project MAC, MIT, Cambridge, MA, 1973.
6. S.P.Landry, and B.D.Shriver, "A dataflow simulation research environment", Workshop on Data Driven Languages and Machines, Toulouse, France, Feb. 1979, pp V 1 - 15.
7. B.D.Shriver, and S.P.Landry, "An overview of dataflow related research at the University of Southwestern Louisiana", Workshop on Data Driven Languages and Machines, Toulouse, France, Feb. 1979, pp 1 - 15.
8. P.Brinch Hansen, "The nucleus of a multiprogramming system", CACM, Vol. 13, No. 4, April 1970, pp 279 - 288.
9. V.P.Srini, "Extended abstract dataflow models for reconfigurable systems", Technical Report, TR-80-3-5, Computer Science Dept., University of Southwestern Louisiana, Lafayette, LA, 70504, May 1980.
10. D.D.Chamberlin,"Parallel implementation of a single assignment language", Technical Report No. 13, Jan. 1971. Stanford Electronics Labs., Stanford University, Stanford, CA.
11. M.Cornish, D.W.Hogan, and J.C.Jensen, "The Texas Instruments Distributed Data Processor", Proceedings of Louisiana Computer Exposition, Lafayette, LA, March 1979, pp 189 - 193.
12. F.Harary, Graph Theory, Addison-Wesley Pub. Co., Reading, MA, 1974.
13. A.V.Aho, J.E.Hopcraft, and J.D.Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesleyh Pub. Co., Reading, MA, 1974.
14. V.P.Srini, "Programming language specification by using three forms", Proceedings of CSC Conference, Detroit, Feb. 1978.

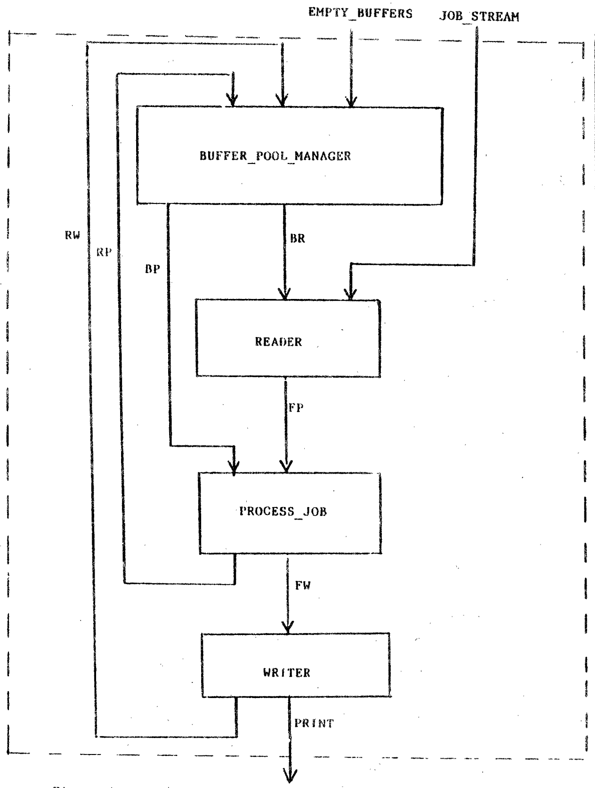


Figure 1 Example of structured message-passing approach

Table 1 Syntax of dp_block

TYPE	DESCRIPTION	MODE DELIVERED
closed form	<label>: BEGIN <name> (<input_arc_descriptions>) {<condition>}$^{1}_{0}$ {INPUT <arc_descriptions>}$^{1}_{0}$ {<constant_declaration>}$^{1}_{0}$ {<type_declaration>}$^{j}_{0}$ {<local_variable_declaration>}$^{k}_{0}$ {<statement>}$^{\ell}_{0}$ {<OUTPUT <arc_descriptions>}$^{1}_{0}$ END (<output_arc_descriptions>) {<condition>}$^{T}_{0}$	<dp-block>
expression form	{(<arc_name>, <arc_capacity>, <label_of_source_dp_block>, <token_type>, <arc_mode>) },$^{m}_{1}$	<arc_ descriptions>
expression form	<Any Pascal statment other than procedure calls, or dp_blocks>	<statement>
expression form	<constant declaration in Pascal>	<constant_ declaration>
expression form	<Type declaration in Pascal>	<type_ declaration>
expression form	<variable declaration in Pascal>	<local-variable declaration>

SUITABILITY OF BUBBLE MEMORIES IN PARALLEL PROCESSOR ARCHITECTURES

Edward W. Davis
Department of Computer Science
North Carolina State University
Raleigh, North Carolina 27650

Summary

Advances in computer architecture result from creative organizational ideas, improvement and innovation in components, and the requirements of new applications. Architectural creativity has led to various parallel processor organizations. Technological inventiveness has produced magnetic bubble memories. When bubble technology was a new item in the research labs, the major anticipated application was mass memory for large computer systems [5]. Now that commercial bubble devices are available, the applications have in reality been microprocessor oriented [9]. Certain properties of bubble storage devices make them quite suitable as components in a memory hierarchy for parallel processors. Five aspects of this suitability are outlined in this paper.

Parallel Processor Memory Considerations: A parallel processor follows the definition that there is a single control unit with the responsibility of driving a set of identical processors. These machines have primary memories from which processing elements (PEs) operate. Secondary memory is typically disk storage interfaced to primary memory, the control unit, or even a host computer. Clearly this definition includes real machines such as ILLIAC IV [1], PEPE [8], and STARAN [2]. It also includes newer ideas such as data base machines [4,7].

There are two aspects of the memory systems that need to be mentioned. First, the amount of primary memory per processor is typically much smaller in parallel processors than in uniprocessors. For example, ILLIAC IV has a 2K word by 64 bits memory associated with each processing element, PEPE has 1K by 32 bits per PE, and STARAN with its more global, multi-dimensional access memory has a 256 by 256 bit memory array associated with 256 PEs or 256 x 9216 in the STARAN E. These numbers reflect memory component technology available at the time the machines were built. They also illustrate the comparatively small primary memory size used in parallel processors. Movement of data in and out of primary memory is an important part of total system performance. The second aspect is secondary memory and its interface to primary. The usual device is a disk. This provides good storage capacity but is poor with respect to access time and interface path.

Bubble Memory Characteristics: Magnetic bubble memory technology became commercially available in the late 1970's. An introductory reference is [9]. Bubble memories are essentially shift register storage structures. Several shift path organizations are possible. One of these, the major-minor loop organization, offers a good compromise between capacity, access time, and simplicity of operation. Figure 1 shows the basic structure. Information flows serially in or out of the device on the major loop but can be transferred in parallel as a "page" to or from the minor loops.

Given a major-minor loop organization, the access time components for a read are (a) position the page at the transfer gates, and (b) transfer to the major loop and shift to the detector. Presently available devices have page position times of 3 to 10 msec and major loop shift times of 4 to 30 msec. At the device level, capacities range from 92K bits to 1M bits. Expectations are that device capacity will double annually and that new techniques will improve shift rates by a factor of ten in the near term.

Support circuits are needed to implement memory systems. These circuits include a controller which provides an interface to other equipment through useful features and functions such as page buffering, format conversion between bits and bytes, maintaining page position, error detection and correction, and indicating status. The controller exercises control over individual bubble devices attached to it. Functions activated at one device are independent of other devices. That is, bubble memory devices are individually operable.

Bubble memories clearly represent a new choice for designers. The properties and characteristics of this new choice need to be examined.

Five Suitability Factors: The five subsections that follow are intended to provide a contrast between bubble and disk devices when used in parallel processor memory hierarchies. Figure 2 shows the model for bubble memory usage as an intermediate level in the hierarchy.

(1) Access Time. With the small primary memory capacity in typical parallel processors, fast access to a secondary storage is important. Access to randomly located information in currently available bubble memories is about ten times faster than access times to random information using movable head disks. Fixed head disks match bubble access times but are not competitive in terms of cost or modularity.

Storage allocation techniques for reducing access times are applicable to both. However, bubble memories have a performance improvement due to their unique capability to "stop" the rotation. Pages can be positioned at the transfer gate waiting for an I/O command.

CH1569-3/80/0000-0053$00.75 © 1980 IEEE

(2) Selectable Input/Output. The ability of an individual processor within a parallel processing system to execute instructions sent from the common control unit, or else do nothing, represents one form of local, individualized control. This control exists because it is useful, or even essential, for devising parallel algorithms. In previous architectures, the ability to enable local entities applied only to processors and the primary memory associated with them. Control of a bubble memory system is readily exercised at the level of individual components. By using such memory components, local control can extend to secondary memory. This is a logical extension of the need for local control which becomes practical through bubble memory technology.

(3) Localized Memory Addressability. In most parallel organizations an additional local control feature is memory reference modification. Addresses supplied to all processors from the control unit can be modified individually within the processors. It is this feature that enables simultaneous access to rows or columns of arrays through skewed storage schemes [6]. It is a variation of this feature that produces multidimensional access memories [3]. In previous machines, local control of addresses was limited to the primary memory. Now, with device level control of bubble storage devices it is possible to extend local addressability to the secondary memory.

(4) Customized Configurations. Bubble memory components are ideally suited to customized design of secondary memory configurations. Modular components allow memory design customized to the number of PEs and capacity requirements. For example, the number of modules can match exactly the number of PEs for bit stream operations. It can be a multiple for byte wide or other size I/O operations. If I/O transfer rates are less demanding, a controller can operate more than one memory module. Essentially, the technology allows a great deal of flexibility.

(5) Fault Tolerance. First note that nonvolatile bubble memories are manufactured using integrated circuit techniques. There are no moving parts or mechanical adjustments. The devices are inherently more reliable than disk storage units. As further protection against failures, storage loops can be provided for single error correction and double error detection. When an uncorrectable failure occurs, the system remains operable with reduced parallelism. It is reasonable to assume the fault can be located to a replaceable unit providing a minimal mean time to repair.

Conclusion: Research is needed to develop a better understanding of the data structures and algorithms for efficient use of bubble memories in parallel processing environments.

References

[1] G. Barnes et al, "The ILLIAC IV Computer," IEEE T. C., (Aug. 1968), pp. 746-757.

[2] K. E. Batcher, "STARAN Parallel Processor System Hardware," Pro. of the NCC, (1974), pp. 405-410.

[3] K. E. Batcher, "The Multidimensional Access Memory in STARAN," IEEE T. C., (Feb. 1977), pp. 174-177.

[4] P. B. Bera and E. Oliver, "The Role of Associative Array Processors in Data Base Machine Architecture," Computer, (March, 1979), pp. 53-61.

[5] P. I. Bonyhard et al, "Applications of Bubble Devices," IEEE Trans. on Magnetics, (Sept. 1970), pp. 447-458.

[6] P. Budnik and D. Kuck, "The Organization and Use of Parallel Memories," IEEE T. C., (Dec. 1971), pp. 1566-1569.

[7] G. A. Champine, "Current Trends in Data Base Systems," Computer, (May 1979), pp. 27-41.

[8] A. J. Evensen and J. L. Troy, "Introduction to the Architecture of a 288 Element PEPE," Proc. of the 1973 Sagamore Computer Conference, (1973), pp. 162-169.

[9] J. E. Juliussen, D. M. Lee, and G. M. Cox, "Bubbles Appearing First as Microprocessor Mass Storage," Electronics, (Aug. 4, 1977), pp. 81-86.

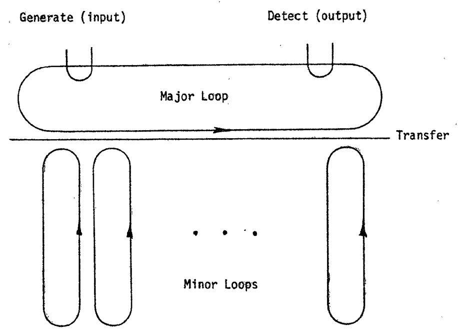


Figure 1. Major - Minor Loop Bubble Memory Organization

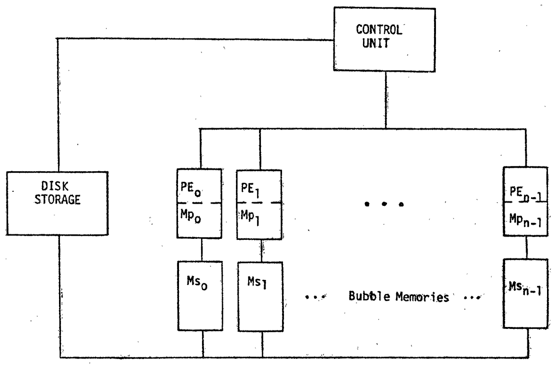


Figure 2. Parallel Processor with Distributed Secondary Memory

ON THE PERFORMANCE OF ON-LINE ARITHMETIC*

Miloš D. Ercegovac and Aksenti L. Grnarov+

UCLA Department of Computer Science
University of California,
Los Angeles, California 90024

Abstract -- An analysis of the performance and effectiveness of on-line arithmetic structures is provided. A relative comparison with structures based on the conventional arithmetic in computational problems such as the evaluation of scalar and vector expressions and recurrence systems indicates speedup and cost benefits of on-line arithmetic structures.

1. Introduction

The purpose of this research is to analyze the performance of on-line arithmetic structures and provide a relative comparison with the conventional arithmetic in computational problems such as the evaluation of scalar and vector expressions and recurrence systems. On-line arithmetic algorithms have been investigated by a number of authors [1-6]. Here we review only the basic definitions and characteristics that are used in the following discussion.

An algorithm is on-line if the j-th leftmost output digit is computed using no more than $(j+\delta)$ leftmost input digits.

* Supported in part by the ONR Contract No. N00014-79-C-0866 (Research in Distributed Processing)

+ On leave from the Department of Electrical Engineering, University of Skopje, Skopje, Yugoslavia.

Thus, an on-line algorithm is performed always in a digit-serial manner from the most to the least significant digit. In order to compute the first digit of the result, the inputs have to be known to $\delta+1$ digits of precision. Thereafter the next most significant digit of the result can be obtained for each additional input digit. The on-line delay δ is a small integer, typically 1 to 5 for the basic arithmetic operations. The algorithms for addition, subtraction, multiplication and square root with $\delta=1$ have been described in the literature [1,5]. The on-line division algorithms require $\delta=3$ to 5, depending on the radix [1,4,5]. Interestingly, there is an algorithm for fast polynomial and rational function evaluation with an on-line delay of -1 [2].

The use of a redundant number system in the representation of the variables is necessary and desirable in on-line arithmetic. Computation of results from left to right in all operations requires the use of a redundant number system in the representation of the results. Consequently, the input operands should also be acceptable in the redundant form. A redundant number system system can be used conveniently in the on-line algorithms. The time required to compute one output digit, t_d, can be made independent of the length of operands by using internally a redundant representation of the partial results. Alternatively, an internal carry-save structure can achieve the same effect.

The on-line representation of a number x is defined as

$$X_j = X_{j-1} + x_{j+\delta} r^{-\delta-j}$$

and

$$X_0 = \sum_{i=1}^{\delta} x_i r^{-i}$$

The digits x_i belong to a redundant digit set

$$\{-\rho,\ldots,-1,0,1,\ldots,\rho\}$$

where $r/2 \leq \rho \leq r-1$ determines the amount of redundancy.

In general, an on-line algorithm is specified recursively in terms of the on-line representations of operands, results and some internal values. The recursion is of the form

$$A_j = f(A_{j-1}, x_{\delta+j}, y_{\delta+j}, z_j)$$

where A_j denotes the internal vectors required by the algorithm. For example, in the case of multiplication A_{j-1} contains the scaled residual w_{j-1}, and on-line representations of the operands X_{j-1} and Y_{j-1} [1]. In general, the internal vectors at the j-th step require j radix r digits in the representation. The primitive operations used in the recursion are addition, multiplication by a single radix r digit, one position shift and concatenation. The output digit is determined by a limited precision selection function[1,2,4,5,7,8]:

$$z_j = S(\hat{A}_{j-1}, x_{\delta+j}, y_{\delta+j})$$

where $\hat{A}$ is a truncated value of A. Since only a small number of most significant digits is required for the selection of the output digit, the recursion can be performed using totally parallel operations, i.e., carry-propagation limited operations. Thus the recursion step time or the time t_d to obtain one output digit is independent of the length of the operands and an on-line algorithm can be implemented in a highly modular manner without speed degradation. An organization of on-line unit as a linear array of identical modules operating in parallel is shown in Figure 1.

The number of modules is determined by the precision s of the selection function $S(\hat{A}_j)$ and the number of digits n:

$$p = \lceil (n + s)/2d \rceil$$

assuming that each module has internally d digits of precision. Detailed descriptions of modular organizations of on-line units are discussed in [2,3,5].

The on-line algorithms are interesting for several reasons. Since the results are always computed from left to right, a sequence of operations can be sped up by overlapping the operations at the digit level. Furthermore, the interconnections in an on-line arithmetic network are much simpler than in a conventional one since only single digits are transferred between the operation units. Therefore, the structures using on-line arithmetic can be implemented in a highly modular manner. The on-line arithmetic realizes by definition a variable-precision arithmetic with a built-in significance indication: for the inputs of k significant digits the output has at least k-δ significant digits.

The on-line algorithms can be used in a floating-point system without difficulties. The exponent arithmetic should be implemented using a conventional approach. One apparent advantage of the on-line floating-point arithmetic is in the operand alignment phase. It can be performed in on-line manner and, thus, overlapped with the mantissa operation. However, in the present discussion we are assuming that, given the same resources, the floating-point exponent operations, operand alignment and mantissa normalization require the same time in on-line and conventional arithmetic. Therefore, our analysis of relative performance of these two approaches is restricted to mantissa operations.

We first consider the performance of on-line and conventional arithmetic unit structures (networks) in evaluating scalar expressions. In this case we are interested in the total delays of networks, their costs and effects of of the interconnection bandwidth on the speedup. The arithmetic units, on-line as well as conventional, are not pipelined. Later we discuss the relative performance and cost-effectiveness of on-line and conventional networks of pipelined units in evaluating vector expressions, i.e., scalar expressions repeated on sets of operands.

2. Evaluation of Scalar Expressions

We consider a scalar expression to be of the form

$$z = E(\underline{x})$$

where z is a scalar, $\underline{x}$ is an argument vector of n-digit elements and E is an arithmetic expression formed with the floating-point operators {+,-,*,/, square root } and the elements of $\underline{x}$.

Assume that a network to evaluate E consists of non-pipelined arithmetic units, connected as a tree network of L levels. In the case of conventional arithmetic we assume that the units at the i-th level begin operating only when all units at the level i-1 have finished. In an on-line network the units are synchronized with a common digit clock. An on-line unit at level i can generate the first output digit as soon as the coressponding δ+1 input digits are available. Therefore, a network of L levels of on-line arithmetic units has the delay (latency):

$$T_{OL} \leq [n + \sum_{i=1}^{L} (\delta_{imax}+1)]t_d$$

where δ_{imax} is the largest on-line delay at the i-th level, n is the number of digits and t_d is the time to compute or load one digit.

Similarly, a network of L levels of conventional arithmetic units has the following delay:

$$T_{CONV} \leq \sum_{i=1}^{L} (T_{imax} + t_{LOAD})$$

where T_{imax} is the time of the slowest operation unit at the level i and t_{LOAD} is the time to transfer operands between two levels in the network.

We assume in our analysis that δ_{imax} is 3 on the average. In the case of conventional arithmetic units, we assume that

$$T_{imax} = cnt_d$$

where c=1 if the conventional arithmetic operation time is T=O(n) and $c = (\log n)^2/n$ if the operation time is $T = O(\log^2 n)$.

The on-line and the conventional

networks, consisting of the same number of units, are compared using the speedup factor S:

$$S = \frac{T_{CON}}{T_{OL}} = \frac{L(cn + 1)}{n + 4L}$$

assuming that $t_{LOAD} = t_d$. The minimum number of levels for which an on-line network is faster than a conventional network is

$$L_{min} = \left\lceil \frac{n}{cn - 3} \right\rceil$$

For example, let n=32 and $c=5^2/32$. Then a network with two or more levels is faster in on-line arithmetic than in the conventional arithmetic. For large L, $S \rightarrow (cn+1)/\delta_{max}$. In particular,

$$\frac{(\log n)^2}{4} < S(oo) \leq \frac{n + 1}{4}$$

The number of levels required to achieve k percent of the maximum speedup is:

$$L = \frac{kn}{4(1 - k)}$$

The relation between the speedup S and the number of levels L is illustrated for n=32 and c=25/32 in Figure 2.

The minimum number of digits for which an on-line network is faster than the conventional one is:

$$n_{min} = \left\lceil \frac{3L}{Lc - 1} \right\rceil < \left\lceil \frac{3}{c} \right\rceil$$

In the previous analysis the difference in the bandwidth requirements of on-line and conventional networks was ignored. If a conventional arithmetic unit has a bandwidth of B digits per variable, its delay is increased to:

$$T_i = (n/B + cn)t_d$$

and the speedup becomes

$$S = \frac{L(n + cBn)}{B(n + 4L)}$$

The additional speedup, due to the bandwidth limitation, is n/4B for large L.

3. Organization of On-Line Structures

A pipelined on-line unit consists of $(n+\delta)$ stages with the stage delay t_d. In the steady state, the unit is computing up to n different results, the first stage producing the first digit of the i-th result and the last stage producing the last digit of the (i-n)-th result. As mentioned before, to implement the recursion of an on-line algorithm, the working precision that increases with the number of steps must be provided. If the result is to be computed to a maximum precision of n digits, the recursion requires at the j-th step a precision of j digits for $j < n/2$ and a precision of n-j digits for $j \geq n/2$. Therefore, n simultaneous operations in various stages of completion require a total working precision of about $n^2/4$ digits.

This indicates that a one-dimensional array of modules, shown in Figure 1, would not be suitable for pipelining since the modules (their internal precision) and the inter-module bandwidth would depend on the relative position in the array. We suggest a two-dimensional array that uses identical modules as illustrated in Figure 3. This array, if implemented with d-digit wide modules, requires $\lceil n/d \rceil$ rows with a variable number of modules per row with the maximum number of modules in a row as indicated above. The total number of d-digit modules for a maximum precision of n digits is approximately $(n/d)^2/4$. In terms of digit circuits, the pipelined one-dimensional and array units have equivalent complexities, the later scheme having more uniform implementation.

4. Evaluation of Vector Expressions

Consider vector expressions that have V vector operands and one vector result, each of M elements:

$$\underline{Z} = E(\underline{X}_1, \ldots, \underline{X}_V)$$

$$\underline{Z} = (\underline{Z}_1, \ldots, \underline{Z}_M)$$

and

$$\underline{X}_i = (\underline{X}_{i1}, \ldots, \underline{X}_{iM})$$

Each vector element is represented with n significant digits.

A conventional pipelined unifunctional unit is assumed to have N stages with the stage delay t_s [9]. The time required to compute M results using a network of L levels of pipelined conventional units, shown in Figure 4, is:

$$T_{COP} = [NL + M - 1]t_s$$

In this analysis we are ignoring the time required to "chain" pipelined units.

A pipelined on-line unit of array type discussed in the previous section, has n + δ stages for a precision of n digits. The stage delay is t_d. The time required to compute M results using a network of L levels of pipelined on-line units, shown in Figure 5, is:

$$T_{OLP} = (L\delta_{max} + n + M - 1)t_d$$

We are assuming that the latencies of a conventional and an on-line pipelined unit satisfy the following condition:

$$Nt_s = cnt_d$$

where c is defined in Section 2. The speedup factor in this case is:

$$S_P = \frac{T_{COP}}{T_{OLP}} = \frac{cn(LN + M - 1)}{N(L\delta_{max} + n + M - 1)}$$

The speedup factor for several cases in which L=4 is given below:

c	n	N	M	S_P
25/32	32	4	100	4.9
1	32	4	100	6.2
1	32	8	1000	3.9
1	64	4	1000	15.0

For a large number of operands, i.e., when $M \rightarrow \infty$, the speedup is:

$$S_P = cn/N$$

and in the case of networks with a large number of levels L, i.e., when $L \rightarrow \infty$:

$$S_P = cn/4$$

These results indicate that the additional speedup due to on-line arithmetic is between 2 and 16 for typical precision.

One distinct advantage of on-line arithmetic is that it can be easily applied in cases that are known to be difficult to speed up using pipeline or parallel computer organizations. For example, non-linear recurrences [10,11] cannot be sped up by algebraic transformations and thus a parallel or a pipeline system organization is not useful. Consider an m-th order non-linear recurrence

$$X(i) = F(X(i-1), \ldots, X(i-m))$$

for $1 \leq i \leq M$ where F requires L levels of operations. Using a network of pipelined on-line units, F can be evaluated in time

$$T_{OLP} = (M \sum_{i=1}^{L} \delta_i + n)t_d$$

In the case of conventional arithmetic:

$$T_{CON} = M \sum_{i=1}^{L} T_i$$

For example, a non-linear recurrence to compute the square root of y

$$x(i+1) = \frac{1}{2}[x(i) + \frac{y}{x(i)}]$$

requires k iterations in order to obtain n digits of precision. If implemented using conventional arithmetic units, the time would be

$$T_{CON} = k(T_{DIV} + T_{ADD})$$

and

$$T_{OLP} = [k(\delta_{DIV} + \delta_{ADD}) + n]t_d$$

in on-line arithmetic.

5. Cost Considerations

The implementation costs of conventional and on-line networks consisting N_U arithmetic units are compared with respect to the total cost of arithmetic modules and the costs of data communications in the network. The cost of a conventional network is defined as:

$$C_{CON} = C_{CU}N_U + (n\log_2 r)C_B N_K$$

where C_{CU} is the cost of a conventional arithmetic unit; C_B is the total communication cost per bit; and N_K is the number of data paths in the network.

Similarly, the cost of an on-line network C_{OL} is defined as:

$$C_{OL} = C_{OU}N_U + (\log_2 r + 1)C_B N_K$$

assuming one signed radix r digit per data path.

If the number of modules required to implement a conventional arithmetic unit with the module cost C_{CM} is at least linearly proportional to the number of digits, i.e.,

$$C_{CU} = nC_{CM}$$

we obtain that

$$\frac{C_{CU}}{C_{OU}} = 2\frac{C_{CM}}{C_{OM}}$$

since the number of modules in an on-line, non-pipelined unit is proportional to n/2. Let

$$\frac{C_{OM}}{C_{CM}} = \alpha$$

The ratio of implementation costs can now be expressed in the following form:

$$R = \frac{C_{CON}}{C_{OL}} = \frac{1 + R_K x}{\alpha/2G + x}$$

where

$$R_K = \frac{H\log_2 r}{\log_2 r + 1}$$

is the communication cost ratio and x is defined as,

$$x = \frac{(\log_2 r + 1)N_K C_B}{nFN_U C_{CM}}$$

where G, H and F are implementation-dependent parameters. We estimate [12] that for non-pipelined units G=1, H=n and F=1 while for pipelined units G=2c, H=1 and λ=c assuming a stage delay of t_d units.

The cost ratio R indicates, for example, that the sufficient condition for an on-line, non-pipelined network to be less costly than the conventional one is that the cost of the on-line module is no more than twice the cost of the conventional module.

6. Concluding Remarks

On-line arithmetic offers an alternative approach in achieving higher speed in numeric computations. On-line arithmetic is complementary to other approaches that are used to achieve concurrency in execution of algorithms: for example, it can be used in minimal-depth tree-structured networks. In particular, the use of on-line arithmetic in non-linear recurrence systems would be advantageous. The main features of on-line networks are (a) high modularity and (b) simple interconnection requirements. These properties make on-line arithmetic very attractive in reconfigurable networks. Importantly, the on-line structures are easily extendable to accomodate either more levels or higher precision. Thus it is interesting to compare the on-line arithmetic networks with the conventional ones. The results of this study indicate that by using on-line arithmetic, besides highly reduced communication requirements and modular, uniform implementation, one can expect an additional speedup factor of 2-16.

Acknowledgements

The authors are grateful to A. Gorji-Sinaki and C.S. Raghavendra for constructive comments.

References

[1] K.S. Trivedi and M.D. Ercegovac, On-line algorithms for division and multiplication, IEEE Trans. on Computers, Vol. C-26, No. 7, July 1977, 681-687.

[2] M.D. Ercegovac, A general hardware-oriented method for evaluation of functions and computations in a digital computer, IEEE Trans. on Computers, Vol. C-26, No. 7, July 1977, 667-680.

[3] M.D. Ercegovac, An on-line square rooting algorithm, Proc. Fourth IEEE Symposium on Computer Arithmetic, October 1978, 183-189.

[4] K.S. Trivedi and J.G. Rusnak, Higher radix on-line division, Proc. Fourth IEEE Symposium on Computer Arithmetic, October 1978, 164-174.

[5] M.J. Irwin, An arithmetic unit for on-line computation, (Ph.D. dissertation), Report No. 873, Department of Computer Science, University of Illinois, Champaign-Urbana, 1977.

[6] M.J. Irwin, Reconfigurable pipeline systems, Proc. 1978 Annual Conf. of the ACM, December 1978, 86-92.

[7] J.E. Robertson, A new class of digital division methods, IRE Trans. Electron. Computers, Vol. EC-7, September 1958, 218-222.

[8] D.E. Atkins, A study of methods for selection of quotient digits during digital division, (Ph.D. dissertation), Report No. 397, Department of Computer Science, University of Illinois, Champaign-Urbana, June 1970.

[9] J.R. Jump and S.R. Ahuja, Effective pipelining of digital systems, IEEE Trans. on Computers, Vol. C-27, No. 9, September 1978, 855-865.

[10] D.J. Kuck, The Structure of Computers and Computations, John Wiley & Sons, Inc., 1978.

[11] H.T. Kung, New algorithms and lower bounds for the parallel evaluation of certain rational expressions and recurrences, Journal of the ACM, Vol. 23, No. 2, April 1976, 252-261.

[12] A.L. Grnarov and M.D. Ercegovac, VLSI-Oriented Iterative Networks for Array Computations, Proc. IEEE Intl. Conference on Circuits and Computers, 1980.

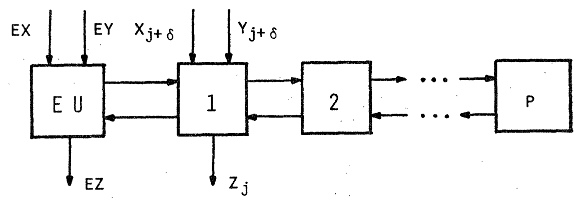
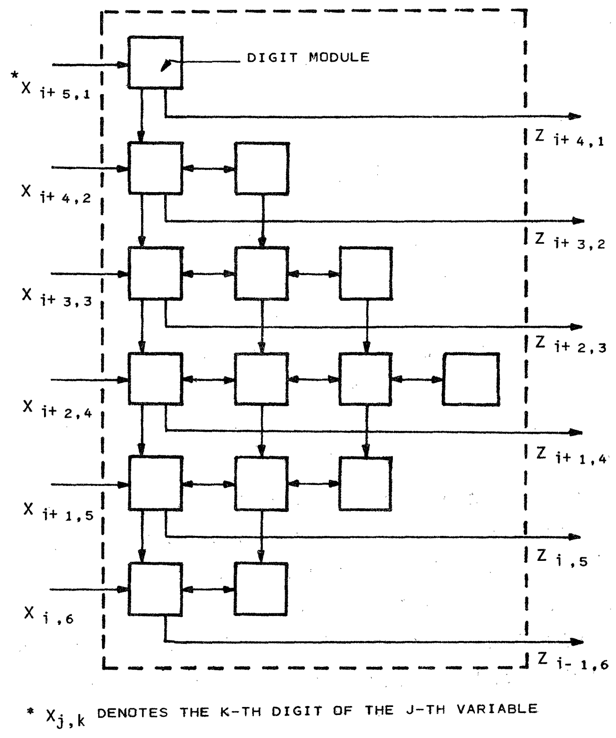


Figure 1: A Modular Organization of an On-Line Unit



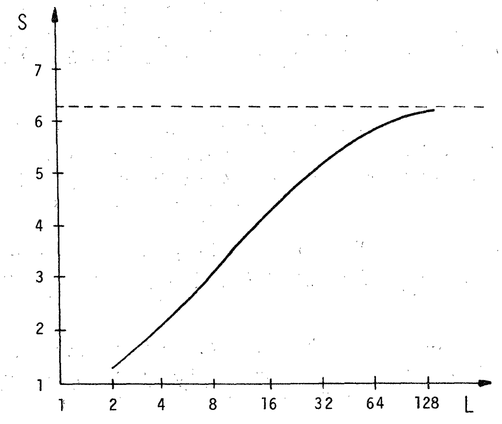


Figure 2: Speedup for n=32 and c=25/32

Figure 3: An Array Organization of Pipelined On-Line Unit (n=5, d=1, s=3, δ =1)

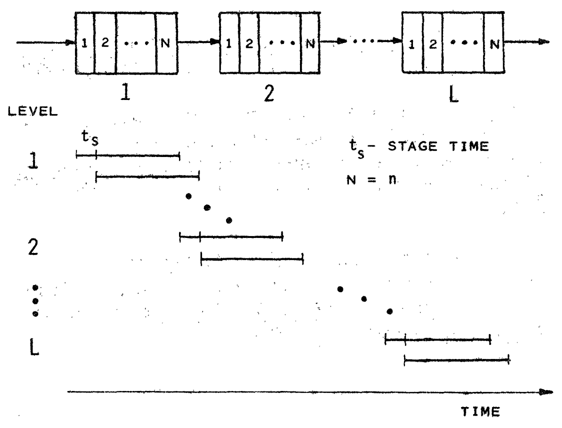


Figure 4: Conventional Pipeline

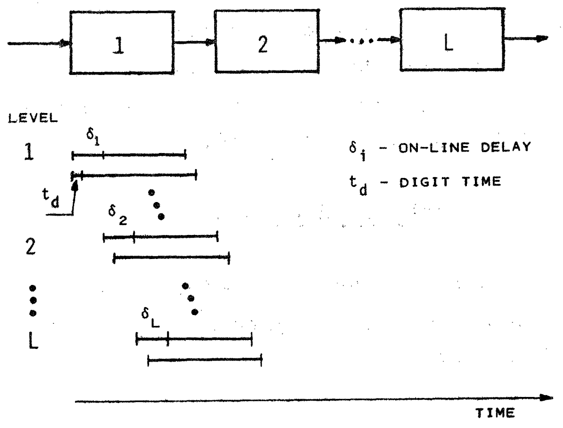


Figure 5: On-Line Pipeline

SESSION 3: INTERCONNECTIONS I

AN INTERCONNECTION NETWORK FOR PROCESSOR COMMUNICATION WITH OPTIMIZED LOCAL CONNECTIONS

Y. Chow, R. Dixon, T. Feng
Computer Science Department
Wright State University
Dayton, Ohio 45435

Abstract -- In this paper we use a simple graph model to describe the routing algorithms for a class of circuit switching networks in interprocessor communication including permutation and full processor communication networks. In full processor communication systems, processors communicate with each other in arbitrary pairs as opposed to pairs between two disjoint sets of processors in a permutation network. It is also assumed that the pool of processors is hierachically structured and the minimum connection paths for local (and/or global) connection are desired. We propose such a full processor communication network with optimized connection paths. Both size and routing complexities are shown to be O(N log N).

I. Introduction

The interconnection network is an essential part of a multiple-processor system and has been widely investigated as a means of interprocessor communications. These networks are generally classified as non-blocking, rearrangeable, or blocking in terms of their flexibility in interconnection. A special class of the interconnection networks is the multi-stage organization. This kind of organization has appeared in various literatures [CLOS 53, BENE 65, WAKS 68, OPFE 71, STON 72, FENG 74, BATC 76, SIEG 78, WU 78, NASS 79, etc]. Research problems associated with multi-stage interconnection networks include system topology, connectivity, control structure (routing), fault tolerance, and cost-effectiveness of the system. In an SIMD or an MIMD environment two major interconnection schemes that are of interest are permutation networks and partition networks. A permutation network performs specific one-to-one connections between two disjoint sets of processors while a partition network partitions a set of processors into disjoint subsets such that the processors within each subset can communicate with each other. A special case of partitioning in which a set of processors is partitioned into pairs of processors will be referred to as full processor communication throughout the paper. This kind of full processor communication can be achieved by extending some existing permutation networks. In this paper we will discuss some proposed full processor communication networks and then present an interconnection network for full processor communication with optimized local connections, i.e., a network in which the pool of processors is hierachically structured and the minimum connection paths for local (and/or global) communications are obtainable. The complexities of routing and switching elements in the network are discussed.

In Section II we review a bipartite graph routing algorithm for permutation networks. The algorithm can also be applied to the routing control in the full processor communication models. Section III includes the discussion of existing full processor communication networks and a network with localized property is proposed. The routing for such network is developed. A formal description of the routing is discussed in Section IV.

II. The Bipartite Graph Routing Algorithm

A general structure of multi-stage networks which allow complete permutation of a set of processors is shown in Figure 1. This NxN (where $N=2^n$) network is recursively defined and Pi denotes a complete permutation network of size $(2^i x 2^i)$. Each (2x2) switching element may assume one of the three states as indicated in Figure 2. This network is a special case of the general Clos network [CLOS 53]. Its structure covers networks such as base line, omega, and indrect binary n-cube networks since it has been shown that these networks are topologically equivalent [WU 78]. It is also shown by Clos [CLOS 53] that this network can realize all N! permutations of the N inputs. The argument is usually made by induction using HALL's theorem [HALL 35] although it can be illustrated easily in the bipartite graph algorithm. Connections between processors (routing) can be established by some local addressing schemes [WU 78] or by a centralized routing control [OPFE 71]. Existing routing algorithms for realizing any permutation using only two states, straight and cross, have been shown to have the complexity of O(N log N) if the algorithm is implemented in a single processor system.

The bipartite graph algorithm is demonstrated as follows. The structure of the network in Figure 1 is recursively defined with $O(\log_2 N)$ stages. Figure 3 shows an example of such network with N=8. This is essentially a 8x8 Benes network. For a given permutation (or connections), the routing is to determine the switch settings of the entire network such that desired connections can be achieved. If we set the switches iteratively from outer stages into the inner stages, we observe that after each iteration the network is divided into two independent subnetworks. This property leads us to a simple conclusion, i.e., in order to connect two processors (one from the left hand side, one from the right

CH1569-3/80/0000-0065$00.75 © 1980 IEEE

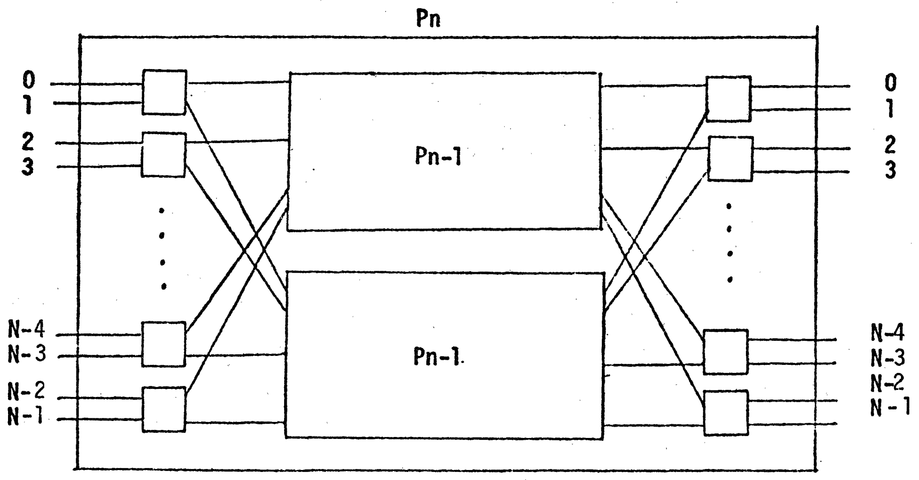


Figure 1: A Base-2 (NxN) Multistage Permutation network ($N=2^n$).

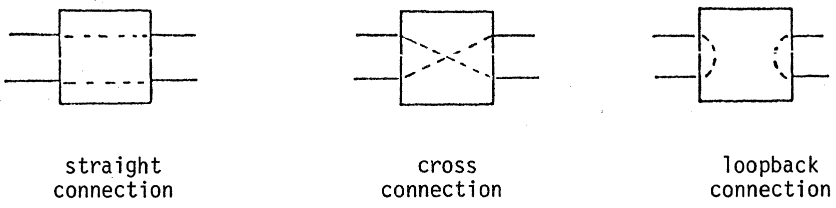


Figure 2: Three States of the (2x2) Switching Element.

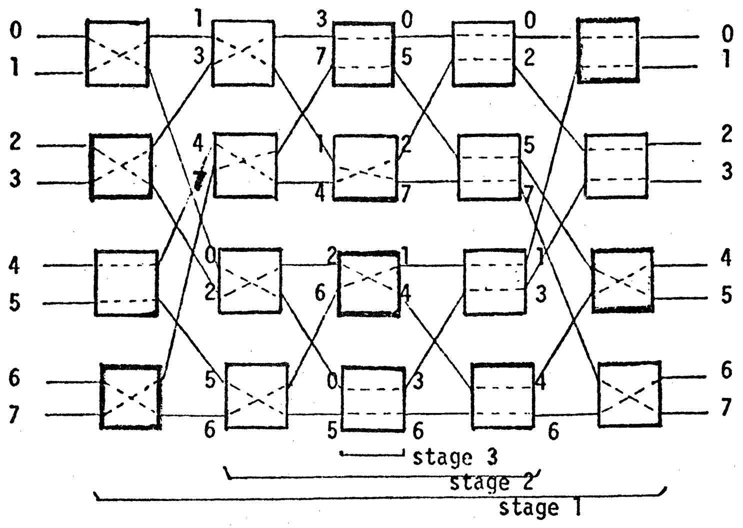


Figure 3: A 8x8 Benes Network.

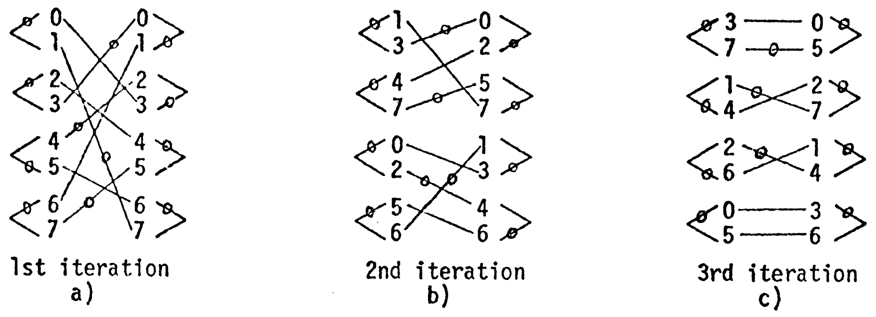


Figure 4: Coloring of Graph for Switch Setting in Figure 3 Network

hand side), they must be switched either both to the upper subnetwork or both to the lower sub-network. This is the basic idea behind the graph algorithm. The graph algorithm is illustrated by the following example of permutation

$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 3 & 7 & 4 & 0 & 2 & 6 & 1 & 5 \end{pmatrix}$. Figure 4-a is a bipartite

graph that represents this permutation. The symbols '$<$' or '$>$' denote a switch and the lines across the two set of numbers (processors) denote the desired connections. A mark o on a switch indicates that the corresponding processor will be switched down (or up) while the other processor to the same switch should be switched up (or down). The whole graph is marked such that each pair (two numbers linked by a line) are both marked or both unmarked. This process ensures that the two processors in each connected pair will always go to the same sub-permutation-network in the next stage. It is obvious that we can rephrase the marking process by saying that the paths in the graph are marked alternately. The same process is repeated as shown in Figure 4-b and Figure 4-c for the subnetworks. It takes log N iterations to complete the marking of the graphs and therefore the switch settings of the entire network. The result is shown with dashed lines in Figure 3.

There is a non-conflict marking for every permutation graph since there are an even number of paths and the marking is done alteratively. After log N iterations we will always obtain $2^{\log N-1}$(2x2) subgraphs and still maintain the desired connections. It is always possible to realize a (2x2) subgraph by a (2x2) switch. Thus all N! permutations are realizable by using the interconnection network in Figure 1. The routing algorithms using the graph model requires the traversal of the graph and is of complexity O(N). There are a total of log N stages in the network. The overall complexity for setting the switches for any permutation is therefore O(N log N). Since subgraphs are independent, parallel processors may be assigned for computing the routing. 2^{i-1} processors may be used to set the $\frac{N}{2^{i-1}}$ switches at the ith iteration. Thus a parallel algorithm would require the following computations:

$$N + \frac{N}{2} + \frac{N}{4} + \dots + \frac{N}{2^{\log N}}$$

For large N, this would have an upper bound of 2N. We reduce the time complexity from O(N log N) to O(N) if parallel processors are available.

III. The Full Processor Communication Models

In full processor communication systems, processors communicate with each other in arbitrary pairs as opposed to pairs between two disjoint sets of processors in a permutation network. Full processor communication can be achieved by including the loop-back state of the (2x2) switching elements or by using additional two-ways (straight and cross states) switches in a conventional binary switching network. Several full processor communication models are presented in this section. Subsection A describes a non-blocking network using three-state switching elements with complexity $O(N^2)$. Subsection B introduces a blocking interconnection network with complexity O(N log N). Finally in subsection C, we present a rearrangeable model with optimal connections and of complexity O(N log N).

A. A non-blocking network using three-state switching elements

By using all three states of the (2x2) switching elements as shown in Figure 2, Gecsei [GECS 77] shows a non-blocking full processor communication system with $O(N^2)$ switching elements. A typical eight processors network is shown in Figure 5 in which pairs of processors (07) (16) (25) (34) are to be connected. For N processors the possible ways of connections is $(N-1)\times(N-3)\times\dots 3\times 1$. The total number of switches required is $(N-2) + (N-4) + \dots + 2 = \frac{N^2}{4} - \frac{N}{2}$. The non-blocking

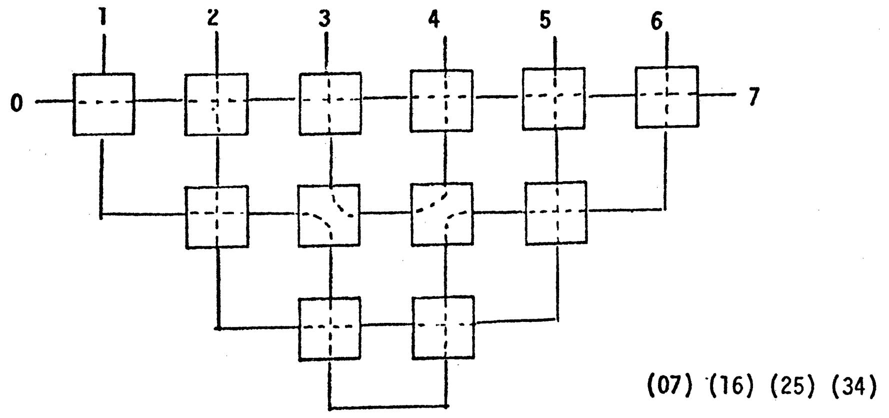


Figure 5. A Non-Blocking Interconnection Network with Three-State Switches.

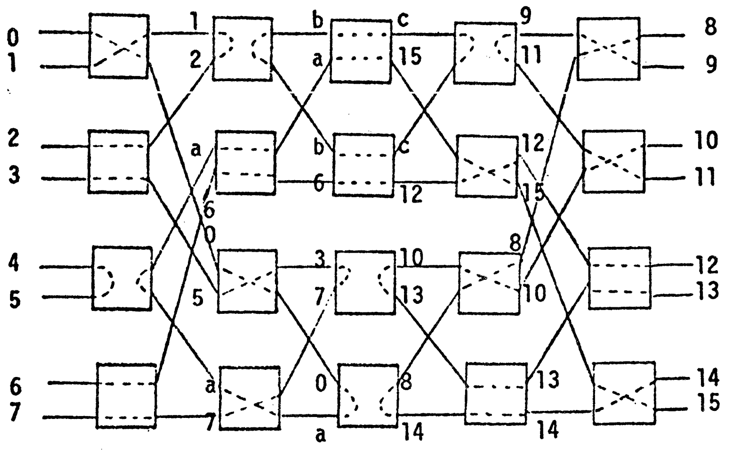


Figure 6: A Sixteen Processors Communication Network.

property of the network can be easily shown by induction.

B. A blocking network using three-state switching elements

The non-blocking network previously described becomes impractical for large N since it has a size complexity of $O(N^2)$ and an average delay of O(N). Better solution must be solicited. By incorporating the loop-back state in the (2x2) switching element, it becomes possible to connect a pair of processors in the same side of the multi-stage permutation network. The permutation network thus becomes a full communication network of 2N processors. Figure 6 is an example of 8x8 sixteen processors network for the connection of {(1 2) (4 5) (0 15) (3 7) (6 12) (8 14) (10 13) (9 11)}. Since the network is hierachically structured we can introduce the concept of local connections, e.g. connections (01) (23) (45) (67) are considered as the first level local connection, (03) (46) as the second level local connections, and (15) (36) as the third level local connections, etc. Connections between the two sides of the network are considered as long distance connections. If we use binary numbers to name the processors, then the levels can be represented in bit positions. Assume that lower level local connections are more likely to occur than the long distance connections. It is therefore desirable to have minimum delays for the local connections such that the overall performance of the routing delays can be improved. With some modifications the graph algorithm presented in Section II can be used for the routing control in the full processor communication systems. It is illustrated in the following example. Figure 7 is a connection graph similar with that of the permutation network. Again the '$<$' or '$>$' denotes a switch, the curved lines and the straight lines represent local connections and long distance connections respectively. There are two unconnected sub-graphs in the graph. Both sub-graphs have odd number of paths. An alternating marking is possible only if the number of paths is even in a sub-graph. The sub-graph $<^{4}_{5})$ is called a minimum sub-graph since it has the minimum number of paths possible in a graph. Such a graph implies an immediate loopback since

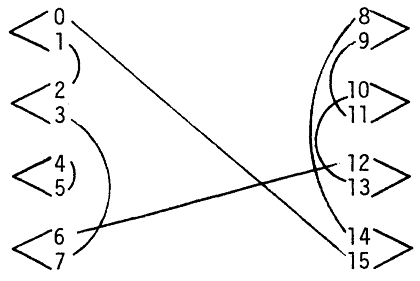


Figure 7: Connection Graph for Connection of Sixteen Processors

the alternating marking is impossible. This loopbacked switch can be utilized by other sub-graphs. A non-minimum sub-graph with even number of paths can be marked as usual. If it contains odd number of paths, some rearrangements have to be made. This rearrangement must utilize the loopback switch, if available, to make an even paths graph such that an alternating marking is possible. Figure 8 shows such an example of marking. The path between 0 and 15 is deleted

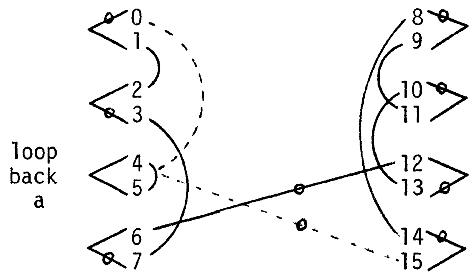


Figure 8: Rearrangement of Connections.

and connections of (0 4) and (5 15) are established. This rearrangement makes an even path subgraph and yet maintains the connection of (0 15) because the (4 5) connection is a loopback and the output of the switch is shorted. The marking is done for the first iteration (Figure 8) and the outside layer of switches is set accordingly (Figure 6). The second and third iterations of the marking process are shown in Figure 9 where a, b, c are the common connection points due to loopback switches. The final switch setting dashed lines in Figure 6) shows that all local connections have the shortest paths at the expense of the prolonged delay of the long distance connection (0 15) which has a delay of 9 switches.

It can be seen from the above graph example that a connection is realizable only if all sub-graphs have an even number of paths in each iteration or all non-minimum subgraphs with odd number of paths can be made into even paths subgraphs by combining with minimum subgraphs. Although the network has a complexity of O(N log N) and has local connection property, it is a blocking network. Figure 10 is an example that connections can not be realized. The connection {(0 8) (1 11) (2 12) (3 13) (4 14) (5 6) (7 15) (9 10)} involves two sub-graphs with odd number of paths and no minimum subgraph can be utilized. In this example the loop back of the local connection (9 10) forces the two terminals 8 and 11 to both go to the upper subnetwork or the lower subnetwork. The two terminals 0 and 1 to be connected with 8 and 11 can only go to different subnetworks. Thus one of the connections cannot be made unless there is a minimum subgraph that provides a loopback for the connection.

C. A rearrangeable network using two-state switching elements

Both non-blocking and blocking full processor communication networks presented earlier use three-state switching elements. The former network requires $O(N^2)$ switching elements while the later has a size complexity of O(N log N). We now propose an O(N log N) full processor communication network which has optimal local connections and requires only two-state switching elements. Figure 11 is a sixteen inputs modified reverse-exchange network. The sixteen outputs on the right hand side are shorted to form the connections {(0 8) (1 9) (2 10) (3 11) (4 12) (5 13) (6 14) (7 15)}. It can be seen that four switches in the lower right corner are redundant. The network now consists of two parts: a partition network on the left and a permutation network on the right. The partition network partitions the input processors such that half of the processors goes to the upper part of the permutation network and the other half is sent to the lower part of the permutation network. For any desired full processor communication, e.g. the eight sets of connections (0 15) (1 4) (2 7) (3 14) (5 9) (6 11) (8 10) (12 13) of sixteen terminals, if we can partition the terminals into two sets such that the two terminals in all the connection pairs appear in different sides of the permutation network, then it becomes possible to achieve the desired connections. We will show by using the graph model that the

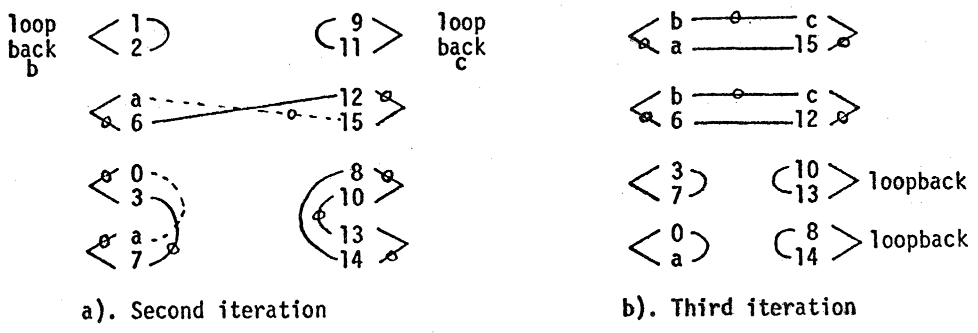


Figure 9: 2nd and 3rd Iterations of the Marking Process

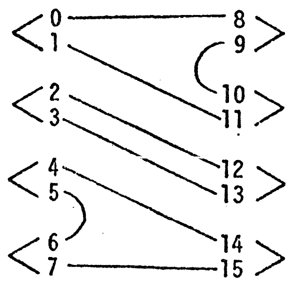


Figure 10: An Unrealizable Connection.

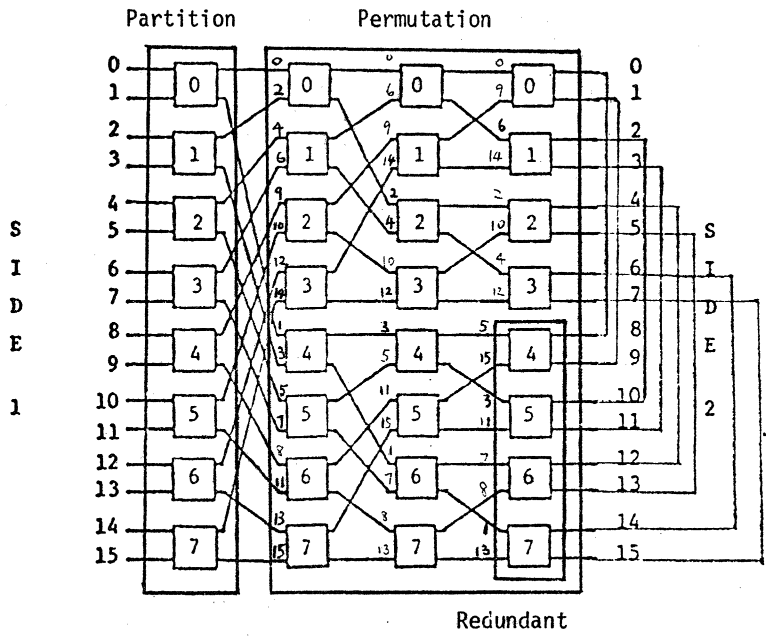


Figure 11: A Modified Reverse Exchange Network for Full Processor Communication

partition network in the box shown in Figure 11 will accomplish such a partition. The graph in Figure 12 represent the desired connections. To obtain a connection, the two terminals linked by a curved line should be dispatched to two different sides of the permutation network, e.g. pair (2 7), if terminal 2 is labeled left (l) then terminal 7 should be labeled right (r). We can traverse the graph marking the terminal l or r without considering the curved lines as paths. Since we have even number of straight paths, a partition of the terminals into two sets is always possible. The labeling of the graph in Figure 12 gives us the two sets (0 2 4 6 9 10 12 14) and (1 3 5 7 8 11 13 15). By using these two sets as both sides of the permutation network we can achieve the permutation
$\begin{pmatrix} 0 & 2 & 4 & 6 & 9 & 10 & 12 & 14 \\ 15 & 7 & 1 & 11 & 5 & 8 & 13 & 3 \end{pmatrix}$ by using the graph algorithm and therefore establish the connections. The result is shown in Figure 11.

The full communication network is similar with the one proposed by Chung and Wong [CHUN 79]. However the interconnection network is centralized in the sense that all terminals are considered as a single group and all connections have the same delay (the number of switches traversed). We are interested in the concept of local connections, i.e., processors are hierachically structured and local connections are expected to have minimum delays. We have shown that the network in Figure 11 can be used to get all possible connections. By structuring the network we can obtain an equivalent network with local properties. To illustrate the restructuring we use the same network in Figure 11. The network is first unfolded (turn the bottom half to the right hand side) in order to make the picture clearer. Then we merge the redundant switches and rearrange the switch boxes. The network is then converted into the following (Figure 13).

The twisted switch boxes in the center stage essentially serve as the purpose of straight through long distance connection or loopback local connections. Such additional switches can be used in other stages to provide an immediate loop back local connection without effecting the other routing in the whole network as indicated by dash lines in the figure. The structure of the network indicates that these modifications can be grouped in pairs as a standard form shown in Figure 14. Two-state switches S1 and S2 are additional switches for loop-back purposes. There are four inputs (a, b, c, d) to each pair of switches. To perform loop-backs we have six possible connections: (a b), (a c), (a d), (b c), (b d), (c d). Connections (a b) (c d) can be ruled out because in the graph algorithm they would have been routed to different center stages. Connections (a c) (b d) are not necessary because they come from the same subnetwork and if local connections are desired they would have been looped back in the previous stages. Thus, if a local loop back is desired, it should be either (a d) or (b c). In other words, if we use binary code for each input, a loop back is performed only if the pair of processors differ in the two least significant bits. Switch S1 and S2 in Figure 14 have the ability of looping back (a d) and (b c) in straight-state and preserving original connections when in cross-state.

The overall algorithm for full processor communication is summarized as follows:

a. partition the terminals into two disjoint sets by using the graph algorithm

b. connect the two sets of terminals by using the permutation graph algorithm iteratively

c. restructure the network and set switches for routing local connections

The complexity of the overall process is O(N log N). The process establishes all connections with shortest possible routes for both local and long distance connection with twice the number of switches as in a non-localized network. The number of switches remains O(N log N). The making of the localized full communication network is more formally described in the following section.

IV. A Formal Description of Routing in the Network

We have shown that the interconnection network in Figure 13 can be used for permutation and full processor communication. We may formally define our switching schemes as follows:
Let the processors be labeled 0 to $2^n - 1$.
For each processor, a, define its binary expansion as:

$$a = a_n a_{n-1} \dots a_2 a_1.$$

Number the stages of the switching network as 1,2,...,n-1,n,n-1,...2,1.
In all the networks below we define a connection and switching procedure for which a is switched to

location $a_n a_{n-1} \dots a_2 a_1$ at input to stage 1

location $a_n a_{n-1} \dots a_3 x_1 a_2$ at input to stage 2

location $a_n a_{n-1} \dots x_2 x_1 a_3$ at input to stage 3

.
.
.

location $a_n x_{n-2} \dots x_1 a_{n-1}$ at input to stage n-1

where x_i's are determined by the switch positions.

A. Permutation Network

For permutation the n^{th} stage is redundant. Here we may map any a to b if $a_n = \bar{b}_n$ ($\bar{0}=1$, $\bar{1}=0$).

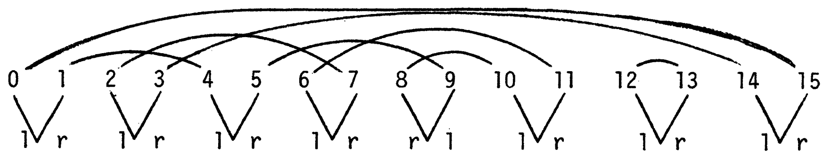


Figure 12: Graph for Partitioning 16 Terminals.

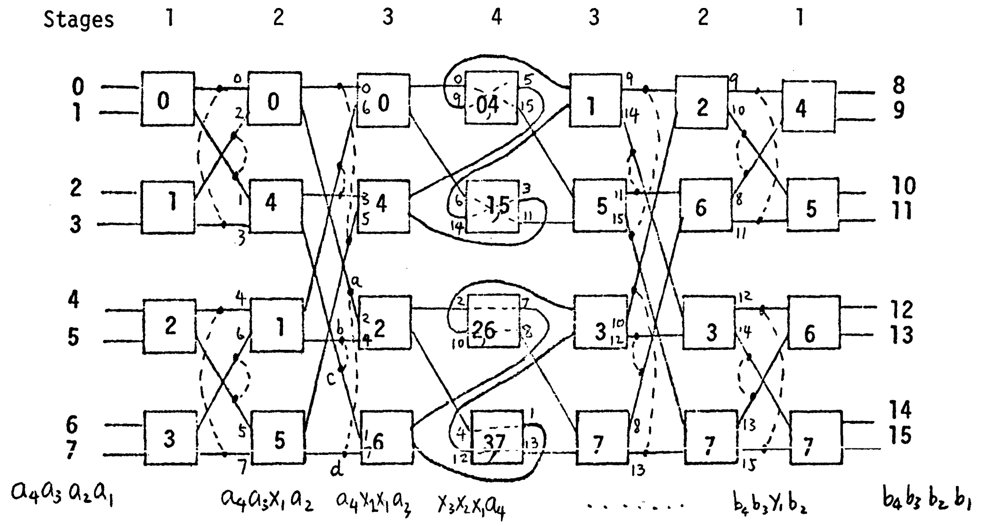


Figure 13: A Restructured Interconnection Network with Local Properties.

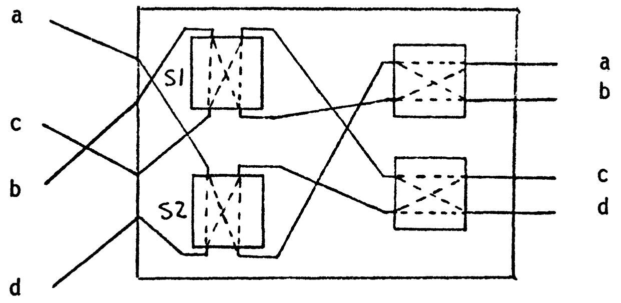


Figure 14: A Standard Circuit Which Provides Loopback and Preserves Other Connections.

The n-1th stage consists of 2^{n-2} switches each of which corresponds to the middle n-2 digits of the input, $X_{n-2} \ldots x_1$. The 4 inputs are determined by $a_n\ a_{n-1}$(i.e. the first and last bit). The switch is one of type which will map $a_n\ a_{n-1}$ to $\overline{a}_n y$, y = 0 or 1.

Thus if our routing algorithm maps a to b then $a_n = b_n$ and
if a maps to $a_n x_{n-2} \ldots x_1 a_{n-1}$
and b maps to $b_n y_{n-2} \ldots y_1 b_{n-1}$
then $x_{n-2} = y_{n-2}$, $x_{n-3} = y_{n-3}; \ldots x_1 = y_1$.

B. Full Processor Communication Without Local Connection

The inputs to the nth stage are $x_{n-1}x_{n-2}\cdots x_2x_1a_n$. The nth stage consists of 2^{n-2} switches each of which corresponds to the front n-2 digits of the input $x_{n-1}x_{n-2}\cdots x_2$. The four inputs to the center stage switches are determined by the digits x_1a_n. The switch has the mappings

$x_1a_n \rightarrow \bar{x}_1y$

where a maps to $x_nx_{n-1}\cdots x_1a_n$

and b maps to y_ny_{n-1} y_1b_n

then $x_n=y_n\ldots x_2=y_2$ and $x_1=\bar{y}_1$

C. Full Processor Communication With Optimized Local Connection

Two inputs a and b are local at stage k if $a_n=b_n\ldots a_{k+1}=b_{k+1}$ and $a_k \neq b_k$. Let a and b be local at stage k, and suppose we want to connect them. Follow the full connection algorithm to stage k. Then we have

a is at $a_n\ldots a_{k+1}x_k\cdots x_1a_k$

b is at $b_n\ldots b_{k+1}y_k\cdots y_1b_k$

Since $a_n=b_n\ldots a_{k+1}=b_{k+1}$ and since we know the routing produces $x_k=y_k\ldots x_2=y_2$, $x_1=\bar{y}_1$, we know the precise relationship of these calls already

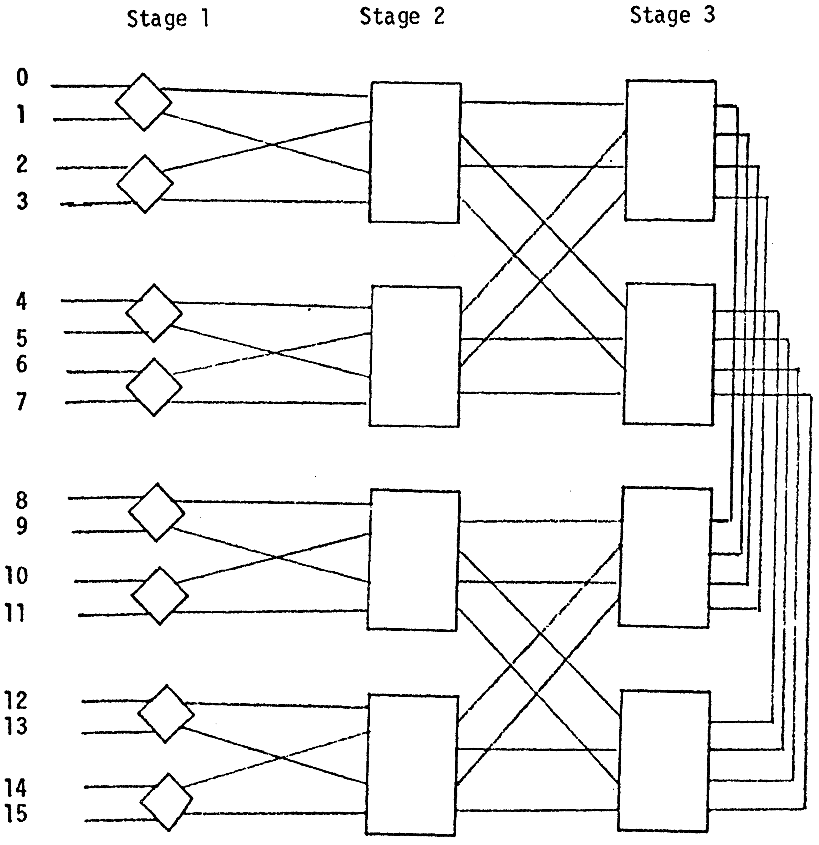


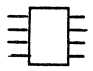
Figure 15: A Full Communication Network With Localized Connections.

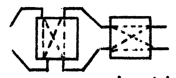
at the kth stage, i.e. $x_1=\bar{y}_1$, $a_k=\bar{b}_k$, the local connection differs in the two least significant bits.

We add to the network switches to optionally connect at each stage k, k=1, 2, ... n input $Z_nZ_{n-1}\ldots Z_1$ to $Z_nZ_{n-1}\ldots Z_1$. This exactly doubles the number of switches in the network. The additional switch type is shown in Figure 14.

We now notice that at the nth stage, if a is to be mapped to b with $a_n=b_n$ then it will have been cut across by the n-1 stage switches. Thus the mapping $x_1a_n \to \bar{x}_1\ y$ where y=0 or 1 is only necessary when $a_n=\bar{y}$. Thus no switch is necessary and we map $x_1a_n \to \bar{x}_1\bar{a}_n$ and we can eliminate the nth stage.

Finally, we give one last diagram using the switch from Figure 14, represented by the symbol

 , and to represent

this figure:  . Figure 15 is the full processor communication network with optimized local connections.

Conclusion

We have used a graph model in computing the routing for permutation network, partitioning network, and full processor communication network. Special emphasis is placed on multistage full interconnection network with hierachical structure. A rearrangeable non-blocking interconnection network with local properties is developed for full processor communication. Such network provides shortest routing for both local and long distance connections. The complexity of the routing algorithm and the number of switches used are both in the order of N log N. It is also shown that by using parallel processors, the routing computation time can be reduced to O(N). In addition to the rearrangeable non-blocking interconnection network, blocking and non-blocking models are reviewed. Interconnect networks play an important role in communication and parallel processor systems.. Further research results in the application of the techniques used in this paper to general Clos networks with full processor communications and local routing are expected.

References

[BATC 76] "The Flip Network in STARAN," K.E. Batcher, Proceedings of the 1976 International Conference on Parallel Processing, pp. 65-71, 1976.

[BENE 65] Mathematical Theory of Connecting Networks, V. Benes, New York, Academic Press, 1965.

[CHUN 79] "Asymptotically Optimal Interconnection Networks from Two-States Cells," K. M. Chung and C. K. Wong, IEEE Transaction Computer, Vol. C-28, No. 7, pp. 500-505, July 1979.

[CLOS 53] "A Study of Non-blocking Switching Network," C. Clos, Bell Systems Technical J., Vol. 32, pp. 406-424, 1953.

[FENG 74] "Data Manipulating Functions in Parallel Processors and their Implementations," T. Feng, IEEE Transaction Computer, Vol. C-23, No. 3, pp. 309-318, March 1974.

[FENG 79] "A Microprocessor-Controlled Asynchronous Circuit Switching Network," T-Y Feng, C-L Wu, D. P. Agrawal, Proceeding 6th Annual Symposium on Computer Architecture, pp. 202-215, April 1979.

[GECS 77] "Interconnection Networks from Three-State Cells," J. Gecsei, IEEE Transaction Computer, Vol. C-26, No. 8, pp. 705-711, August 1977.

[HALL 35] "On Representatives of Subsets, "P. Hall, J. London Math. Soc., 10, pp. 26-30, 1935.

[NASS 78] "An Optimal Routing Algorithm for Mesh-Connected Parallel Computers," D. Nassimi and S. Sahni, TR 78-19, University of Minnesota, 1978.

[OPFE 71] "On a Class of Rearrangeable Switching Networks," D. C. Opferman and N. T. Tsao-Wu, Bell System Journal, Vol. 50, No. 5, pp. 1579-1599, May-June 1971.

[SIEG 78] "Study of Multistage SIMD Interconnection Network," Proceeding 5th Annual Symposium on Computer Architecture, pp. 223-229, April 1978.

[STON 72] "Parallel Processing and the Perfect Shuffle," H. S. Stone, IEEE Transaction Computer, Vol. C-20, No. 4, pp. 357-366. April 1972.

[TRIP 79] "Packet Switching in Banyan Networks," A. R. Tripathi, G. J. Lipovski, Proceeding 6th Annual Symposium on Computer Architecture, pp. 160-167, April 1979.

[WAKS 68] "A Permutation Network," A. Waksman, JACM, Vol. 15, pp. 159-163, 1968.

[WU 78] Interconnection Networks in Multiple Processor Systems, Ph.D. Dissertation, Wayne State University, C.L. Wu, 1978.

USE OF THE AUGMENTED DATA MANIPULATOR MULTISTAGE NETWORK FOR SIMD MACHINES

S. Diane Smith
University of Wisconsin, Electrical and Computer Engineering Dept., Madison, WI 53706

Howard Jay Siegel, Robert J. McMillen, George B. Adams III
Purdue University, Electrical Engineering School, West Lafayette, IN 47907

Abstract -- The capabilities of the augmented data manipulator (ADM) and the inverse ADM (IADM) as permutation networks for SIMD machines are explored. Redundant control settings for commonly used permutations are examined. A method to count the number of distinct permutations performable by these networks is given. Finally, techniques for controlling these networks in SIMD mode are presented.

I. INTRODUCTION

In [13] it is shown that the multistage cube networks called the generalized cube, omega, indirect binary n-cube, and STARAN flip are equivalent and that the capabilities of the augmented data manipulator (ADM) network are a superset of those of these multistage cube networks. In this paper, the use of the ADM in an SIMD environment is studied.

An SIMD (single instruction stream-multiple data stream) machine has a control unit which broadcasts instructions to N processors. A processor along with its private memory is called a processing element or PE. All active PEs execute the same instruction at the same time, each processor on data from its own memory. Data can be transferred by the interconnection network from PE to PE. Each PE is assigned a unique address from 0 to N-1, where $N=2^n$.

An interconnection network can be described as a set of interconnection functions, where each interconnection function is a permutation (bijection) on the set of PE addresses [8]. When interconnection function f is applied, input i is connected to output f(i) for all i, $0 \leq i < N$, simultaneously. An equivalent definition is that the interconnection network takes the set of PE addresses as its input and produces as its output a permutation of these PE addresses, i.e., it maps an input address to an output address.

The Plus-Minus 2^i (PM2I) network consists of the 2n functions defined by

$PM2_{+i}(j) = j+2^i \bmod N$ and $PM2_{-i}(j) = j-2^i \bmod N$

for $0 \leq j < N$, $0 \leq i < n$ [8], where $(-x = N-x) \bmod N$.

The data manipulator network [2], Fig. 1, consists of n stages with N switching cells per stage, plus a column of network output cells. The stages are ordered from n-1 to 0, where the interconnection functions of stage i are $PM2_{+i}$, $PM2_{-i}$,

This work was supported by the Air Force Office of Scientific Research under AFOSR-78-3581. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

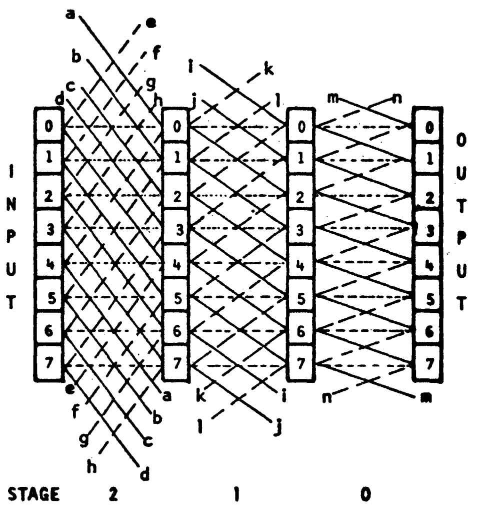


Figure 1: The data manipulator network for N=8.

and the identity (straight). There is one pair of control signals per stage. At stage i, cells whose i-th address bit is 0 respond to one control, the other cells to the other control.

The augmented data manipulator (ADM) is a data manipulator with individual cell control [9,11,13,14]. Each cell receives control signals independently of any other cell.

If the stages of the ADM are traversed in reverse order, i.e., the input stage is stage 0 ($PM2_{\pm 0}$) and the output stage is stage n-1 ($PM2_{\pm n-1}$), the resulting network is the inverse ADM (IADM) [15].

II. CAPABILITIES OF THE ADM AND IADM

Lemma 1: The ADM passes the permutation f if and only if the IADM passes the inverse permutation.
Proof: See [15]. ⬡

Theorem 1: The ADM can perform a perfect shuffle in one pass through the network.
Proof: The perfect shuffle interconnection function is $\text{shuffle}(p_{n-1} \dots p_1 p_0) = p_{n-2} \dots p_1 p_0 p_{n-1}$, $P = p_{n-1} \dots p_1 p_0$, $0 \leq P < N$. The switch settings for stage i, $n > i \geq 0$, are determined as follows, where the address of a cell P at stage i is $p_{n-1} \dots p_1 p_0$.

```
set stage n-1 to straight across;
for i = n-2 step -1 until 0 do
  if $p_{i+1} \neq p_i$
    then if $p_{i+1}=0$
```

```
        then set cell P at stage i to $PM2_{+i}$;
        else set cell P at stage i to $PM2_{-i}$;
    else set stage i to straight across;
```

For the controls calculated from the algorithm, data originally from PE $p_{n-1} \cdots p_1 p_0$ is sent to cell $p_{n-2}p_{n-3} \cdots p_i p_{n-1} p_{i-1} \cdots p_0$ at stage i. This algorithm is related to the "PM2I → shuffle" algorithm in [10] and is proved correct in [15]. ⬡

Corollary 1: The IADM can perform an inverse perfect shuffle in one pass through the network.
Proof: Follows from Lemma 1 and Theorem 1. ⬡

Theorem 2: The IADM cannot perform a perfect shuffle in one pass through the network.
Proof: Assume arithmetic is mod N. Consider P = $0^{n-2}11$, where the superscript is a repetition factor, e.g., $0^4 11$ = 000011. The difference of the addresses P and shuffle(P) is an odd number. Since no combination of $PM2_{+i}$ and $PM2_{-i}$, $0<i<n$, yields an odd number difference as the shuffle does, data from P must use $PM2_{+0}$ at stage 0. The distance between P+1 and shuffle(P+1) is even, as is the distance between P-1 and shuffle(P-1). The straight connections are used for the data from P+1 and P-1 at stage 0, creating a conflict. ⬡

Corollary 2: The ADM cannot perform an inverse perfect shuffle in one pass through the network.
Proof: Follows from Lemma 1 and Theorem 2. ⬡

The generalized cube and its equivalents [13] cannot perform the shuffle or inverse shuffle (for $N \geq 16$, $00p_{n-3} \cdots p_1 1$ and $10p_{n-3} \cdots p_1 1$ conflict at stage n-1 for the shuffle, and $1p_{n-2} \cdots p_2 01$ and $1p_{n-2} \cdots p_2 11$ conflict at stage 1 for the inverse shuffle).

Theorem 3: A bit reversal function transfers data from PE $P = p_{n-1} \cdots p_1 p_0$ to $P' = p_0 p_1 \cdots p_{n-1}$. For N>8, the IADM cannot perform a bit reversal in one pass through the network.

Proof: Let P = $0^{n-2}11$. The distance between P and its bit reversal is an odd number, so $PM2_{+0}$ must be used. The distance between P+1 and its bit reversal is an even number, as is the distance between P-1 and its bit reversal. The straight connections are used for the data from P+1 and P-1 at stage 0, creating a conflict. ⬡

Corollary 3: For N>8, the ADM cannot perform a bit reversal in one pass through the network.
Proof: Follows from Lemma 1 and Theorem 3. ⬡

For some transfers, more than one setting exists for the ADM. In addition to being of theoretical interest, the existence of redundant paths adds a certain amount of fault tolerance. Two classes of these redundant settings are shown (details in [15]).

Theorem 4: There are n-i different control settings for the ADM which realize the $Cube_i$ interconnection function, $0 \leq i \leq n-2$.

Proof: $Cube_i (p_{n-1} \cdots p_1 p_0) = p_{n-1} \cdots \bar{p}_i \cdots p_0$, $0 \leq i < n$ [8]. $Cube_i$ can be realized by setting the ADM controls such that at stage i, cells whose i-th address bit equals 0 perform $PM2_{+i}$, while those whose i-th address bit is 1 perform $PM2_{-i}$. Since, $2^i = 2^k - 2^{k-1} - \cdots - 2^i$, n>k>i, there are n-i different settings for the ADM which accomplish $Cube_i$. Data items from an arbitrary pair of inputs, P and P', P<P', cannot conflict.
Case 1: $p_i = p_i'$. Always P'-P cells apart.
Case 2: $p_i \neq p_i'$. For stage j, j>i, the data items must be in cells which differ in at least the i-th bit position (since $p_i \neq p_i'$). At stage i, the data from P will be at cell $Cube_i(P)$ and the data from P' will be at $Cube_i(P')$, which will differ in at least the i-th bit position. ⬡

The uniform shift permutations send data from PE P to P' = P+A mod N, 0<A<N, for all PEs. Let A = $a_{n-1} \cdots a_1 a_0$.

Theorem 5: The ADM has redundant control settings for all uniform shifts of A mod N, 0<A<N.
Proof: The ADM can be set as follows: at stage i, if $a_i=0$, then set the network to straight across; if $a_i=1$, then set the network to $PM2_{+i}$. Let A be expressed in signed digit notation, where $a_i' \in \{0, +1, -1\}$, the sum and difference of powers of 2 (e.g., A = 0111 = 100(-1) = 10(-1)1 = 1(-1)11). The following are all equivalent [7]:

$$a'_{n-1} \cdots a'_k 01 \ldots 110 a'_j \cdots a'_0$$
$$a'_{n-1} \cdots a'_k 10 \ldots 0(-1)0 a'_j \cdots a'_0$$
$$a'_{n-1} \cdots a'_k 10 \ldots 0(-1)10 a'_j \cdots a'_0$$
$$a'_{n-1} \cdots a'_k 1(-1)1 \ldots 10 a'_j \cdots a'_0$$

Each of these different representations of A can be used to yield control settings for the ADM network as follows: at stage i, if $a_i'=0$, then set stage i to straight across; if $a_i'=1$, to $PM2_{+i}$; if $a_i'=-1$, to $PM2_{-i}$. Since all cells in a stage are set the same way, no conflicts can occur. ⬡

Corollary 4: Theorems 4 and 5 hold for the IADM.
Proof: Follows from Lemma 1, $Cube_i^{-1} = Cube_i$, and the inverse of shift A mod N is shift N-A mod N. ⬡

One measure of a network is the number of permutations it can perform. The generalized cube network (and its equivalents [13]) can perform $2^{Nn/2}$ permutations [12]. The following theorems consider the number of permutations performable by the ADM (details in [1]).

Lemma 2: For N = 4, the ADM can perform all possible N! = 24 permutations.
Proof: By enumeration (see [15]). ⬡

A size N ADM can be partitioned into two independent subnetworks of size N/2 [11], plus stage 0. These subnets have the same structure as a size N/2 ADM. All the inputs of one subnet are even-numbered (the even subnet). The subnet with all the odd-numbered inputs is the odd subnet. The connection of the two subnets to stage 0 of the size N ADM is shown in Fig. 2. All even-numbered inputs of stage 0 are connected to the outputs of the even subnet and all odd-numbered inputs to the outputs of the odd subnet.

Let S_i, D_i specify a source/destination pair.

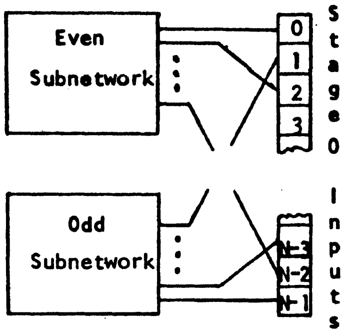


Figure 2: Partitioning the ADM network.

A connection in stage 0 that does not affect the low order bit, i.e., $(s_0)_i = (d_0)_i$, is a straight connection. A connection that changes the low order bit, $(s_0)_i \neq (d_0)_i$, is called an exchange (see Fig. 3). A regular exchange is between stage 0 inputs $P = p_{n-1}p_{n-2}\cdots p_1 0$ and P+1. An irregular exchange is between stage 0 inputs P and P-1 mod N. Any possible configuration of stage 0 that is a permutation, except the all +1 or all -1 configurations, consists of straight and exchange connections only [11] and can be expressed as an N-bit number. A bit is associated with each adjacent pair of inputs, including the wrap-around pairing of 0 and N-1. If the adjacent pair of inputs form an exchange, the bit is 1; if not, 0 (see Fig. 3).

Two kinds of adjacency for binary numbers are distinguished. When the first and last bits of the binary number (representing the wrap-around) are not considered adjacent it is linear adjacency. When the first and last bits are considered adjacent it is circular adjacency.

Lemma 3: Every configuration of stage 0, except the settings all +1 or all -1, that is a permutation, has a unique associated binary number with no circular adjacent bits that are 1.

Proof: If there are circular adjacent 1's, then an input P is in two exchanges such that $P \rightarrow P+1$ mod N and $P \rightarrow P-1$ mod N. ⬡

Lemma 4: The number of N-bit numbers with no linear adjacent 1's is $B(N)=B(N-1)+B(N-2)$; $B(2)=3$, $B(3)=5$, $N \geq 4$.

Proof: If the number ends in a 0, it must have no linear adjacent 1's in the first N-1 bits. If it ends in a 1, the bit immediately preceding must be a 0, and the first N-2 bits must have no linear adjacent 1's. ⬡

Lemma 5: For an N-input ADM network, the number of stage 0 configurations that yield a permutation of stage 0 inputs to outputs is $\alpha(N) = B(N) - B(N-4) + 2$; $N \geq 8$.

Proof: By Lemma 3, $\alpha(N)$ is the number of N-bit numbers with no circular adjacent 1's, plus all +1 and all -1. B(N) exceeds the number with no circular adjacent 1's by the number with no linear adjacent 1's which do have circular adjacent 1's. These numbers are of the form $10a_1a_2\cdots a_{N-4}01$ where $a_1\cdots a_{N-4}$ has no linear adjacent 1's. ⬡

Lemma 6: Consider the stage 0 permutations except the all irregular exchanges, all +1, and all -1. Any two of these permutations differ in the source subnetwork for at least one output.

Proof: Consider two distinct permutations of the given set. There must be at least one output D_i which is mapped differently. If output D_i is connected to a straight stage 0 connection, $(d_0)_i = (s_0)_i$. If it is connected to an exchange at stage 0, $(d_0)_i \neq (s_0)_i$, and it receives its data from a different source subnet. ⬡

Theorem 6: A lower bound on the number of distinct permutations performable by the ADM, P(N), is $P(N) \geq P(N/2)^2 \cdot [\alpha(N)-3]$; $P(4)=24$; $N \geq 8$.

Proof: Each subnet can perform P(N/2) permutations. Let stage 0 be restricted to any permutation other than all +1, all -1, or all irregular exchanges; there are $\alpha(N)-3$ such configurations. By Lemma 6, any change in the stage 0 setting will cause at least one output to be mapped from a different subnet, changing the overall permutation. P(4) is from Lemma 2. ⬡

Theorem 7: An upper bound on the number of distinct permutations performable by the ADM is

$$P(N) < P(N/2)^2 \cdot \alpha(N);\ P(4)=24;\ N \geq 8 .$$

Proof: Assuming that the composition of any input permutation with any stage 0 permutation yields a unique overall permutation gives the above result. The < inequality is because the assumption is false (there are redundant settings). ⬡

III. NETWORK CONTROL

Routing tags are used to distribute control of the network among the N PEs. A full routing tag $= f_{2n-1}f_{2n-2}\cdots f_1f_0$ at each input can specify any arbitrary path. In stage i, if $f_{2i}=0$, the straight link is used; if $f_{2i}=1$ and $f_{2i+1}=0$, the $+2^i$ link is used; otherwise the -2^i link is used. If all the sign bits in a full tag are the same, form an n+1 bit routing tag by computing the signed magnitude difference between destination D and source S: $T = t_nt_{n-1}\cdots t_1t_0 = D-S$, where $t_n=0$ indicates positive and $t_n=1$ negative, and $t_{n-1}\cdots t_1t_0$ equals the absolute value of D-S [5]. At stage i if $t_i=0$, the straight connection is used; if $t_n=0$ and $t_i=1$, the $+2^i$ link is used; otherwise the -2^i link is used. If all N tags for a permutation are calculated in this way, then the permutation is routed using natural routing tags. An individual route consisting of only straight or $+2^i$-type connections is positive dominant; an individual route consisting of only straight or -2^i-type connections is negative dominant [5].

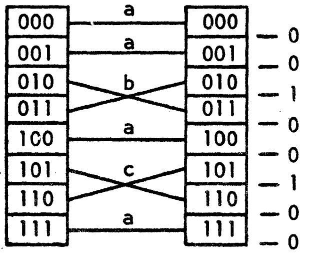


Figure 3:
a) Straight connections
b) Regular exchange
c) Irregular exchange
Also shown, the associated binary number (N = 8).

Two tags are equivalent if they route a message from the same source to the same destination.

Theorem 8: Let A' denote the two's complement of A and $T \neq 0$ $(S \neq D)$. Then T' is equivalent to T.

Proof: See [5]. ⬡

A permutation is routed using positive dominant routing tags if those tags that are negative dominant in the set of natural routing tags are converted to positive dominant using Theorem 8.

Lenfant has defined five families of frequently used permutations [4]. Theorems 9 to 12 show that two of the families are passable by both the ADM and IADM using positive dominant tags. The proofs are very briefly sketched and the details are in [6]. Let $(\hat{a}_2,\hat{a}_1)$ be the bitwise representation of an address P, $\hat{a}_2$ the j high order bits, and $\hat{a}_1$ the n-j low order bits. $\langle T\rangle_r$ denotes T mod 2^r.

Lemma 7: The location of a message in stage i of the IADM is cell $\langle S + \langle T\rangle_i\rangle_n$, where T is the magnitude portion of its positive dominant tag.

Proof: At stage i, bits 0 to i-1 have been examined, so the message has been displaced by $\langle T\rangle_i$. ⬡

Theorem 9: The class of permutations $\lambda^{(n)}_{j,k}$, which maps X to jX+k mod N (j odd), is passable by the IADM using positive dominant tags.

Proof: Lemma 7 is used to show no conflicts can occur. ⬡

Theorem 10: The class of permutations $\lambda^{(n)}_{j,k}$, which maps X to jX+k mod N (j odd), is passable by the ADM in one pass using positive dominant tags.

Proof: Lemma 1, Theorem 9, and properties of the ring of integers mod N [3] are used to show $\lambda^{-1}=\lambda$ and the class is passable. ⬡

Theorem 11: The class of permutations $\delta^{(n)}_{j,k}$, which maps $(\hat{a}_2,\hat{a}_1)$ to $(\hat{a}_2,\lambda^{(n-j)}_{1,k}(\hat{a}_1))$ $(j \leq n)$, is passable by the IADM using positive dominant tags.

Proof: It is shown that if $\hat{a}_1 < 2^{n-j}-k$, the tag is $00\ldots0k_{n-j-1}\ldots k_1k_0$; otherwise it is $11\ldots1k_{n-j-1}\ldots k_1k_0$. This is used to demonstrate no conflicts can occur. ⬡

Theorem 12: The class $\delta^{(n)}_{j,k}$ $(j \leq n)$ is passable by the ADM using positive dominant tags.

Proof: Lemma 1 and Theorem 10 are used to show $\delta^{-1}=\delta$ and the class is passable. ⬡

Positive dominant routing tags cannot be used to route all passable permutations without conflict (e.g. perfect shuffle).

Theorem 13: The perfect shuffle is passable by the ADM network using natural routing tags.

Proof: If $p_{n-1}=1$, T = (shuffle(P)-P)≤ 0, i.e. T is negative dominant. In Theorem 1, if $p_{n-1}=1$, the bit pair $p_{i+1}p_i$ will always be of the form 10 or 11. The algorithm specifies settings of -2^i and straight respectively, representable by a negative dominant tag. The case for $p_{n-1}=0$ is similar. ⬡

Corollary 5: The inverse shuffle permutation is passable by the IADM using natural routing tags.

Proof: Follows from Lemma 1 and Theorem 13. ⬡

The tags used in Theorems 9 to 13 require only n+1 bits and are easy to compute. If a passable permutation is needed, but cannot be represented with natural or positive dominant tags, full routing tags can be precomputed.

IV. CONCLUSIONS

The use of the ADM and IADM networks for SIMD processing have been explored. Analyses such as these are necessary in order to evaluate the cost-effectiveness of the ADM (and IADM) as SIMD interconnection networks.

ACKNOWLEDGEMENTS: Significant contributions to the counting the number of stage 0 permutations were made by C. H. Smith and M. J. O'Donnell.

V. REFERENCES

1 Adams, Siegel, Tech Report in preparation.
2 Feng, "Data manipulating functions in parallel processors and their implementations," IEEE Trans. Comp., 3/74, pp. 309-318.
3 Goldstein, Abstract Algebra: A First Course, Prentice-Hall, NJ, 1973.
4 Lenfant, "Parallel permutations of data: A Benes network control algorithm for frequently used permutations," IEEE Trans. Comp., 7/78, pp. 637-647.
5 McMillen, Siegel, "MIMD machine communication using the augmented data manipulator network," Symp. Comp. Arch., 6/80, pp. 51-58.
6 McMillen, Siegel, Interconnection Networks and Operating System Considerations for PASM - A Reconfigurable Multimicroprocessor System, Elec. Eng. School, Purdue, TR-EE 80-15, 6/80.
7 Reitwiesner, "Binary Arithmetic," in Advances in Computers, 1, Academic Press, NY, 1960.
8 Siegel, "Analysis techniques for SIMD machine interconnection networks and the effects of processor address masks," IEEE Trans. Comp., 2/77, pp. 153-161.
9 Siegel, "Interconnection networks for SIMD machines," Computer, 6/79, pp. 57-65.
10 Siegel, "A model of SIMD machines and a comparison of various interconnection networks," IEEE Trans. Comp., 12/79, pp. 907-917.
11 Siegel, "The theory underlying the partitioning of permutation networks," IEEE Trans. Comput., 9/80.
12 Siegel, McMillen, Mueller, "A survey of interconnection methods for reconfigurable parallel processing system," Natl. Comp. Conf., 6/79, pp. 529-542.
13 Siegel, Smith, "Study of multistage SIMD interconnection networks," Symp. Comp. Arch., 4/78, pp. 223-229.
14 Smith, Siegel, "Recirculating, pipelined, and multistage SIMD interconnection networks," Intl. Conf. Parallel Proc., 8/78, pp. 206-214.
15 Smith, Siegel, Design and Analysis of Interconnection Networks for Partitionable Parallel Processing Systems, Elec. Eng. School, Purdue, TR-EE 79-39, 8/79.

Design and Validation of a Connection Network for Many-processor Multiprocessor Systems[a]

George H. Barnes
Burroughs Corporation
Advanced Development Organization
Paoli, Pa., 19301

Summary

A connection network is described for connecting any or all of a large number of processors on one side to a large number of memory modules on the other. Each processor independently requests connection through the network. Response time is to be commensurate with the access time of memory, and hence no time can be allowed for global control of the network. The connection is made at combinatorial logic speed, and the connection held for accessing one word only. In the specific case studied, a network to be embedded in the Flow Model Processor of the Numerical Aerodynamic Simulator, there were 512 processors and 521 memory modules, with an assumed memory access time of 240 ns. [1,2].

The selected network (the "baseline" network of [3]) is isomorphic to the Omega network of [4]. Figure 1 shows an example of this type of network together with an example of how the individual bits of the requested memory module number control the connection being made through each two-by-two node. To avoid the hazards of designing with arbiters and synchronizers, the connection network is synchronized by a clock, whose cycle time exceeds the roundtrip delay through the net, but may be substantially shorter than the memory access time. The bidirectional path through the network is latched up with the acknowledge bit from the memory module while addresses, memory commands, and data are transmitted. A path width of 11 bits was chosen. This width is wide enough to allow the module number and a strobe through the network in parallel and provides sufficient bandwidth for the balance of the system.

The entire collection of processors will run no faster than the slowest processor due to points of synchronization within the programs being executed. An important constraint on the network is that it treat processor requests fairly since a slow processor will slow the whole system; in the applications studied, all processors had equal amounts of computation to do. Thus on the average no node in the network may favor one input port over another. For redundancy and additional bandwidth the CN is assumed to be duplexed.

The performance of the Connection Network was evaluated by simulation. The simulators collected data on the effect of blockage in the network on processor throughput, particularly the effect on the last processor to finish. The simulator generated its own test cases. Table 1 shows the result when the simulator is presented with a single access from each processor, and then runs to completion. The test cases included the three data access patterns that dominated the aerodynamic flow programs furnished by NASA, as well as the case where the 512 processors request access to memory modules which have been selected at random. Figure 2 shows the result for one case in which each processor has a number of random memory access requests. The three curves in Figure 2 are R (the number of processors making a request on this memory cycle), M (the number of different memory modules represented in these requests (some processors request connection to the same memory module, and thus conflict with each other)), and Z (the number of processors which the connection network succeeds in connecting to some memory module). Z/M is the fraction of successes versus maximum possible number of successes. In all these simulations, the network was duplexed.

Simulations reported by Harris and Zichterman [5], and reproduced here by permission, are shown in Fig. 3 and 4. In this case, the processor queues were filled by requests (and the associated timings) generated by a simulator of the FMP processor. In this way the test case had realistic timings. Fig. 3 shows six accesses during this first iteration of a particular segment of code. Fig. 4 shows the fourth iteration of these same six access patterns. The spread of access times represents the processors getting slightly out of synchronism with each other as some get delayed slightly.

The network, with NlogN complexity, has been validated for application to a many-processor multiprocessor with success not only for the access patterns exhibited in the targeted areodynamic flow code applications, but also for random patterns of accessing.

(a) The information in this paper was previously submitted to NASA Ames as the final report of contract NAS2-9897.

CH1569-3/80/0000-0079$00.75 © 1980 IEEE

References

[1] "Final Report, Numerical Aerodynamic Simulation Facility Feasibility Study", Burroughs Corporation, March 1979, prepared for NASA Ames under Contract NAS2-9897.

[2] G. H. Barnes and S. F. Lundstrom, "A Controllable MIMD Architecture", this conference.

[3] C. Wu and T. Feng, "Routing Techniques for a Class of Multistage Interconnection Networks", Proceedings of the 1978 International Conference on Parallel Processing IEEE Computer Society, 1978.

[4] D. H. Lawrie, "Access and Alignment of Data in an Array Processor", IEEE Transactions on Computers, C-24 (1975) pp. 1145-1155.

[5] K. R. Harris and J. L. Zichterman, "Evaluation of the Numerical Aerodynamic Simulation Facility, Final Report", Computer Sciences Corporation, June 1979, prepared for NASA Ames under contract NAS2-9359.

Table I
Simple Cases

A. Aerodynamic Cases

Pattern:	1st	3d					2d			
p =	1	100		30		50	(1)		(2)	
Cycle	1	1	2	1	2	1	1	2	1	2
R(req.)	512	512	72	512	8	512	512	69	512	4
M(mem.)	512	512	72	512	8	512	447	69	508	4
Z(succ.)	512	440	72	504	8	512	443	69	508	4

B. Random Case

Cycle	1	2	3	4	5	6
R(req.)	512	212	63	16	4	1
M(mem.)	327	158	48	12	3	1
Z(succ.)	300	149	47	12	3	1

Note: First and third patterns are vectors with stride p. Second pattern is the simultaneous access of several vectors. Case (1) has p=1, vector length = 31, and stride between vectors, q, of 31^2 modulo 521. Case (2) has p = 1, length = 100, and q = 5000 modulo 521.

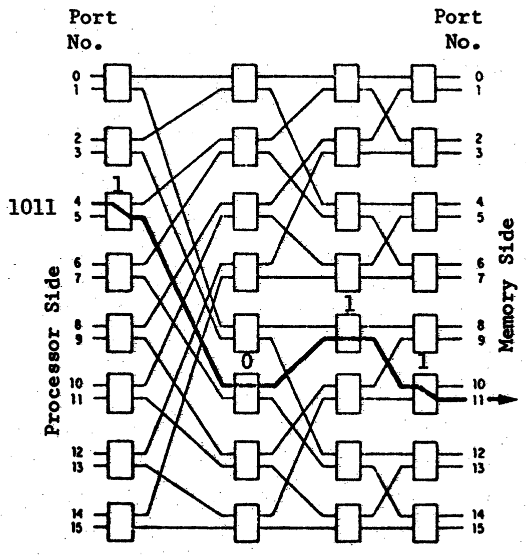


Fig. 1. Network with Example of Control

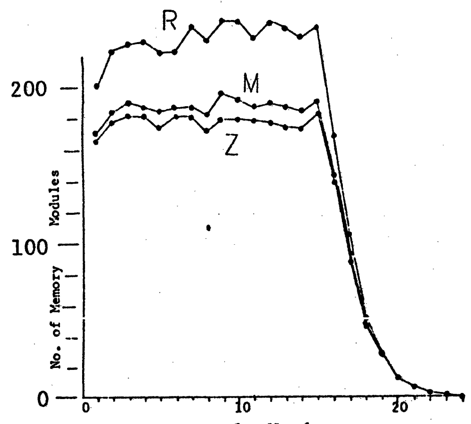


Fig. 2. Sequence of Random Requests

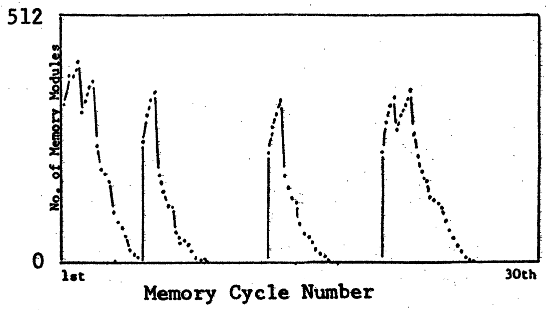


Fig. 3. Test Case from FMP Simulator, Start

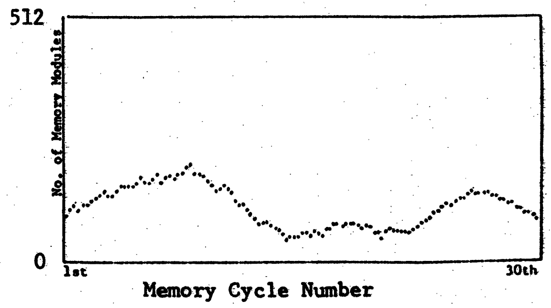


Fig. 4. Test Case from FMP Simulator, cont.

THE PRIME MEMORY SYSTEM FOR ARRAY ACCESS*

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

and

C. Vora**
Burroughs Corporation
Paoli, Pennsylvania 19301

Abstract

In this paper we describe a memory system designed for parallel array access, and first used in the Burroughs Scientific Processor. The system is based on the use of a prime number of memories to allow conflict free access, and a powerful combination of indexing hardware and data alignment switches. The use of a prime number of memories causes certain difficulties in addressing hardware, and particular emphasis is placed on the memory indexing equations and their implementation.

1. Introduction

The problem discussed in this paper is the design of a memory system that can access, in parallel, the required sections of an array, e.g., a row, column, diagonal, etc. A number of these memory systems have been discussed in the literature. In [Batc77], Batcher discusses a scheme for allowing access to words, bit slices, or "byte" slices of a two-dimensional bit array. Feng described another scheme for assessing various "slices" of data in [Feng74]. Other work described in [Ston71], Swan74], [Lang76], [LaSt76], [Orcu76], [Sieg77], [Lawr75], and [Shap75] has treated the problem from a variety of viewpoints. However, all of these designs have restrictions either on the kinds of "slices" available without memory access conflicts or in the data alignment capabilities.

In [BuKu71], Budnik and Kuck observed that if the number of memory modules is a prime number, then access to any "linear" array slices can be achieved without conflict (provided that the memory ordering of the desired array elements is relatively prime to the number of array elements). This observation turns out to be quite useful. However, the problem of addressing this type of memory turns out to be difficult due to the need to do integer divisions and modulo operations in the addressing hardware. In this paper we will discuss these problems in more detail, and will present a feasible implementation of the prime memory system.

Since many of the ideas in this paper have been incorporated in the design of the Burroughs Scientific Processor (BSP), we will describe some of the details of the memory, alignment, and indexing hardware of this machine. The BSP is a high performance computer designed to be especially effective on vector processing applications, without significantly impairing its performance on scalar computations. As can be seen in Figure 1, the BSP consists of sixteen processing units, seventeen memories, two alignment networks, and a

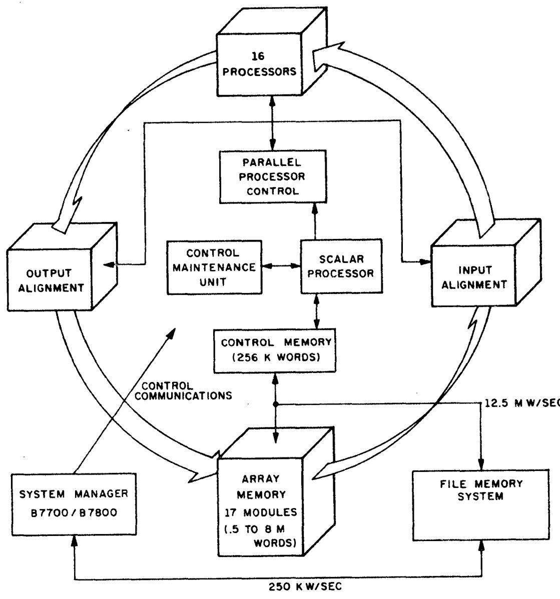


Figure 1. Block Diagram of the Burroughs Scientific Processor

*This work was supported by Burroughs Corporation, Paoli, PA, and in part by the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801.

**Current address is:
Data General Corporation
15 Turnpike Road
Westboro, MA 01581

central control and scalar processing unit. The control unit includes a fully functional scalar processing unit which can be overlapped with vector operations, and additional memory for scalar data and program storage. (See [KuSt79] for further details.) Special hardware is included in the control unit to perform vector addressing and alignment control, and these operations can be overlapped with vector and scalar processing. We refer to the alignment, indexing, and memory systems collectively as the AIM system. We will discuss this system in more detail in Section 3.

The alignment networks shown in Figure 1 are in reality crossbar switches controlled by source tags. (That is, each output port of the network can supply a "tag" which specifies the number of the input from which it needs data.) While in general, crossbar switches are too expensive for large arrays of processors, due to the relatively small number of processors in the BSP it was determined that crossbar switches were the most cost-effective form of switch capable of performing all the desired alignments.

In particular, the functions such as compress, expand, merge, require a random aligning pattern which only the crossbar switch could perform efficiently in the allocated time. Other forms of switches were investigated, e.g., the Swanson network [Swan74], Omega switch [Lawr75], Barrel Shift network, etc., but these switches do not perform all the functions needed in the allocated time, and they become cost-effective only with a large number of ports.

2. The Storage Scheme and Associated Equations

By a storage scheme, we mean the set of rules which determine the module number and address within that module where a given array element is stored. For the present, we will restrict our attention to two-dimensional arrays. However, generalization of these storage schemes is trivial for higher dimensioned arrays.

Figure 2 shows an 8×8 array stored in 5 memory modules using one storage scheme. Notice that any 5 consecutive elements of a row, column, diagonal, etc., all lie in separate modules, and thus can be accessed in parallel, i.e., without conflict. For example, the second through sixth elements of the first row are stored in module numbers 3, 1, 4, 2, 0, and at addresses 2, 4, 6, 8, 10, respectively.

We begin with some definitions. Let M be the number of memory modules and P be the number of processors, where we assume P < M and M is prime. There are two storage equations, f(i,j), and g(i,j) which determine the module number and address, respectively, of element (i,j) of the array. In our case, we have the following equations:

$$f(i,j) = [j * I + i + \text{base}] \bmod M \qquad (1)$$

$$g(i,j) = [j * I + i + \text{base}]/P \qquad (2)$$

where we assume the array is dimensioned (I,J), "base" is the base address of the array, and P is the greatest power of two less than M. Notice

	Memory Module Number				
Address	0	1	2	3	4
0	00	10	20	30	x
1	50	60	70	x	40
2	21	31	x	01	11
3	71	x	41	51	61
4	x	02	12	22	32
5	42	52	62	72	x
6	13	23	33	x	03
7	63	73	x	43	53
8	34	x	04	14	24
9	x	44	54	64	74
10	05	15	25	35	x
11	55	65	75	x	45
12	26	36	x	06	16
13	76	x	46	56	66
14	x	07	17	27	37
15	47	57	67	77	x

Figure 2. Example of an 8×8 Array Stored in 5 Memory Modules

that these equations require a MOD M operation where M is a prime number. They also require an integer divide by P operation. However, P is a power of two which makes this divide easily implementable. This simplification is made possible by the "holes" shown in Figure 2.

Clearly, the number of holes in each row of the memory is equal to M - P in gneral. For example, if M = 37 and P = 32, then 5/37th of the memory is wasted. These holes could be filled with other data, e.g., scalar data, but a cleaner solution is available at the expense of an increase in the complexity of the indexing equations (see [LaVo79]).

Next we define a linear N-vector, or simply an N-vector, to be an N element set of the elements of the array formed by linear subscript equations:

$$V(a,b,c,e) = \{A(i,j):\ i = ax + b,\ j = cx + e,\ 0 \leq x < N\} \qquad (3)$$

where again we assume the array is dimensioned A(I,J). Thus, if a = b = 0 and c = e = 1, then the N-vector (N = 5) is the second through sixth elements of the first row of A: A(0,1), A(0,2), ..., A(0,5). If a = c = 2 and b = e = 0, then the N-vector (N = 4) is every other element of the main diagonal of A: A(0,0), A(2,2), ..., A(6,6). Notice that the elements of the N-vector are ordered with index x.

Next we define the index equations for the N-vector V. We define $\alpha(x)$ to be the address, in module $\mu(x)$, of the x-th element of the N-vector. Thus combining equations (1) through (3) above, we get:

$$\begin{aligned}\mu(x) &= f(ax + b,\ cx + e)\\ &= [(cx + e) * I + (ax + b) + \text{base}] \bmod M\\ &= [dx + B] \bmod M \qquad (4)\end{aligned}$$

where $d = a + cI$ and $B = b + eI + \text{base}$. We define d to be the order of the N-vector, and B to be the initial address. Next we get:

$$\begin{aligned}\alpha(x) &= f(ax + b,\ cx + e)\\ &= [(cx + e) * I + (ax + b) +\\ &\quad \text{base}]/P \qquad (5)\end{aligned}$$

It is easy to show that if d is relatively prime to the number of memory modules, then access to the N-vector can be made without memory conflict. (See [BuKu71] and [Lawr75] for a proof.)

Since it is most convenient to be able to generate the address $\alpha(x)$ in memory $\mu(x)$, we solve for x in terms of μ and get:

$$x(\mu) = [(\mu - B)d'] \bmod M \qquad (6)$$

where d' is the multiplicative inverse of d modulo M. Substituting this into equation (5), we get:

$$\begin{aligned}\alpha(\mu) &= \{(a + Ic)[(\mu - B)d' \bmod M] + b +\\ &\quad eI + \text{base}\}/P\\ &= \{d[(\mu - B)d' \bmod M] + B\}/P \qquad (7)\end{aligned}$$

For example, consider the 5-vector V(0,0,1,1,), i.e., the second through sixth elements of the first row of A(8×8). We have B = 8 and d = 8, thus

$$\mu(x) = [(x + 8) * 8 + 0] \bmod 5,$$

$$\alpha(x) = [(x + 8) * 8 + 0]/4,$$

and since d' = 2 (i.e., 2 * 8 = 1 mod 5), we get:

$$\alpha(\mu) = \{8[2(\mu - B) \bmod M] + 8\}/4.$$

Thus, $\mu(x) = (3, 1, 4, 2, 0)$,

$$\alpha(x) = (2, 4, 6, 8, 10),$$

and

$$\alpha(\mu) = (10, 4, 8, 2, 6).$$

Notice that the proper addresses in memories 0, 1, ..., 4, are 10, 4, 8, 2, 6, respectively. We use the $\mu(x)$ equation in the x-th processor to determine the module number of the memory containing the x-th element of the desired N-vector. At the same time, addressing hardware in memory μ uses the $\alpha(\mu)$ equation to determine the necessary address of the desired element. We use $\alpha(\mu)$ instead of $\alpha(x)$ because this eliminates the need to route the addresses from the processors through the switch.

This process is reasonably straightforward, except that it is not obvious that the hardware can do the necessary calculations efficiently. In Section 3, we will describe how we partition the equations into parts that can be done separately by special hardware in the CU, AU, and memory addressing box.

3. Indexing Hardware

Vector instructions in the BSP are designed to allow processing on vectors of arbitrary length. The control unit automatically sequences vector operations as a series of superword operations where a superword consists of 16 or less vector elements. For example, a vector instruction which specifies a vector of length 53 would be sequenced as three superwords of 16 elements, followed by a superword of 5 elements.

Associated with every array is an array descriptor (AD), shown in Figure 3(a). The two values in the AD describe the base address and total volume (words) of the array, and are used for addressing and bounds checks on the array. Every vector instruction refers to at least one and as many as six vector operands. Each vector operand is referenced through a vector set descriptor (VSD), shown in Figure 3(b). The VSD actually describes a set of vectors from a given array. B is the address of the first element of the first vector in the set. This vector is ordered with distance d, and contains L elements. The first element of the second vector in the set is the (signed) distance D from the first element of the first vector. There are K vectors in the set. Thus the VSD describes a two-dimensional set of data.

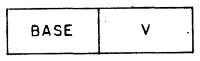
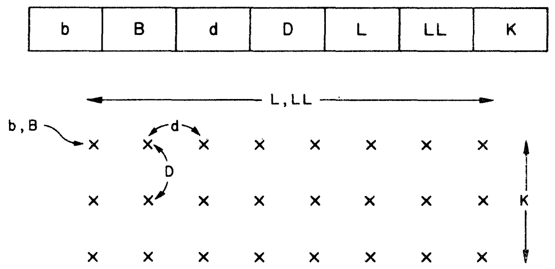


Figure 3(a). Array Descriptor



b: Memory address of the first element of a superword
B: Memory address of the first element of a vector
d: Vector element displacement
D: Vector displacement
L: Remaining (unprocessed) length of a vector
LL: Initial length of a vector
K: Total number of vectors in the vector set

Figure 3(b) Vector Set Descriptor (VSD)

For example, the VSD (B = 1, d = 8, L = 8, D = 2, K = 4) describes the odd numbered rows of the array A(8,8) shown in Figure 2. Similarly, VSD (B = 0, d = 1, L = 8, D = 16, K = 4) describes even numbered columns, and VSD (B = 0, d = 0, L = 8, D = 1, K = 8) describes a two-dimensional set of data, X(i,j), where X(i,j) = A(i,1), $0 < i, j < 8$, and A(i,j) is the array shown in Figure 2. The above parameters are not all stored together. The first step in preparing a vector instruction is to compute the above parameters, together with other values needed for addressing and alignment. This is greatly facilitated by special-purpose indexing hardware.

The purpose of the indexing hardware is to generate alignment tags and memory addresses for vector access. Consider first the input alignment network. To access a superword, processor p must generate an input alignment tag, IAT, which specifies the memory module number of the p-th element of the superword, i.e., $\mu(p)$. At the same time, the address of the p-th element, $\alpha(p)$, is generated in memory $\mu(p)$. Notice that each processor could generate the required address using equation (5), and then route this address to the proper memory through the output alignment network. However, by using equation (7), we avoid the extra routing operation.

The output alignment network works similarly. Memory $\mu(p)$ is to receive the p-th element of a superword, and thus generates an output alignment tag, OAT, whose value is computed from equation (6) above. Each memory also computes the required address, $\alpha(\mu)$, for storing the output.

The alignment, indexing, and memory systems are responsible for a number of other functions. We will discuss these functions in a later section. For now, we will restrict our attention to accessing linear N-vectors.

3.1 Linear N-Vector Access

Let us assume for the moment that we are interested in access to a single superword, with initial base address B, and with order d. If the superword is to be fetched from the memory, then for each memory μ, we must generate an address (see equations (4) through (6))

$$\alpha(\mu) = \{B + p(\mu) \cdot d\}/P \qquad (8)$$

where $$p(\mu) = (\mu - B)\, d' \bmod M \qquad (9)$$

and for each processor p, we must generate an IAT

$$\mu(p) = (B + d * p) \bmod M \qquad (10)$$

However, if the superword is to be stored in the memory, then for each memory μ, we must generate an address given by equations (8) and (9) and for each memory μ we must also generate an OAT

$$p(\mu) = [(\mu - B)\, d'] \bmod M \qquad (11)$$

Thus M-addresses and P-IAT's or M-OAT's are required to access a superword. In the next section we will show how the generation of these values can be simplified.

3.1.1 Recursive Generation Technique

Consider the equation (10). Substituting (p + k) and (p - k) for p, we get

$$\begin{aligned}\mu(p \pm k) &= [B + d * (p \pm k)] \bmod M \\ &= [\mu(p \pm k \mp 1) \pm d] \bmod M \qquad (12)\end{aligned}$$

Equation (12) implies that $\mu(p \pm k)$ can be generated from $\mu(p)$ with modulo M addition/subtraction operations instead of a multiply followed by a modulo M addition. Extending the notion, from any $\mu(p)$ all tags can be generated recursively with appropriate modulo M additions or subtractions. In practice, primary $\mu(p)$ for several values of p are generated using equation (10), and secondary $\mu(p)$ for the remaining values of p are generated using equation (12). The number of primary $\mu(p)$ versus the number of secondary $\mu(p)$ calculated can be determined by a simple hardware versus time tradeoff.

The same technique can be applied to generate output alignment tags and memory addresses. The equation for the OAT's is:

$$p(\mu \pm k) = [p(\mu \pm k \mp 1) \pm d'] \bmod M \qquad (13)$$

For memory addresses, the equation is:

$$\alpha(\mu + k) = (B + \{[p(\mu \pm k \mp 1) \pm d'] \bmod M\} \mp d)/P \qquad (14)$$

3.1.2 BSP Implementations

For the BSP, P = 16 and M = 17. The base address, B, is a 23-bit value. Element displacement, d, is a 23-bit signed quantity. For timing and hardware considerations, 4 initial memory addresses, 4 IAT's and 4 OAT's are generated by using multiplications and modulo and normal additions. Other addresses and tags are generated by using binary adders. To use the binary adders, the equations described in the previous section were further simplified as follows. Let δ = d mod M, and notice that $\mu(p) < M$. For IAT's, we get

$$\mu(p + k) = \mu(p) + k\delta - cM \qquad (15)$$

where $cM \le \mu(p) + k\delta < (c + 1)\, M$

For example, assume M = 17. We might generate primary $\mu(p)$ for p = 1, 4, 7, 10, 13, 16 from equation (10). Secondary $\mu(p)$ for the remaining values of p would be generated as follows from equation (15).

$$\mu(p + 1) = \mu(p) + \delta \text{ corrected by } -17 \text{ if } \mu(p) + \delta \ge 17$$

$$\mu(p - 1) = \mu(p) - \delta \text{ corrected by } +17 \text{ if } \mu(p) + \delta < 0$$

Equations for OAT's are the same as above except δ is replaced by d'. For memory address generation, the equations are as follows. Let

$$A(\mu) = B + p(\mu) \cdot d \quad \text{so that } \alpha(\mu) = A(\mu)/P.$$

Then

$$\begin{aligned} A(\mu + k) &= B + d \cdot p(\mu + k) \\ &= B + d([d'(\mu + k - B] \bmod M) \\ &= B + d([p(\mu) + dk'] \bmod M) \\ &= A(\mu) + kdd' - dcM \end{aligned}$$

where $cM \leq p(\mu) + kd' < (c + 1)M$ (16)

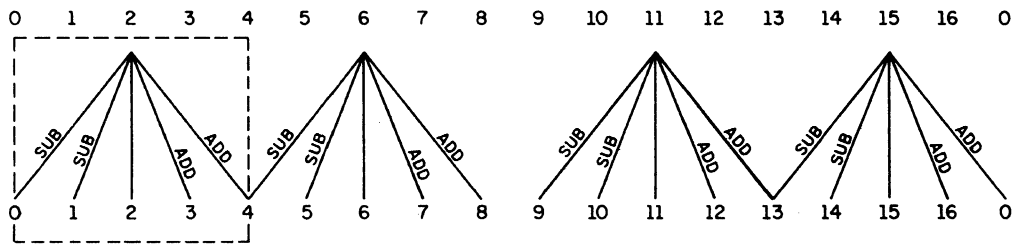


Figure 4. Primary and Secondary Address Generation

Address generation in the BSP is performed as follows. Primary $A(\mu)$ are generated for $\mu = 2$, 6, 11, and 15 as shown in Figure 4. Then secondary values are generated for $K = \pm 1, \pm 2$. Notice that $A(\mu)$ for $\mu = 4$ and 13 are each generated twice. This redundancy is used to check the hardware integrity by comparing duplicated values. (In addition, modulo 3 checks are performed on all additions to further verify hardware integrity.)

A primary $A(\mu)$ generator is shown in Figure 5. B mod M and d' are each 5-bit quantities (since M = 17) and are supplied by the Central Index Unit (to be discussed in the next section). The quantity $d'(\mu - B) \bmod M$ is supplied by a 1024×5 bit ROM. (The ROM contents differ for each primary μ.)

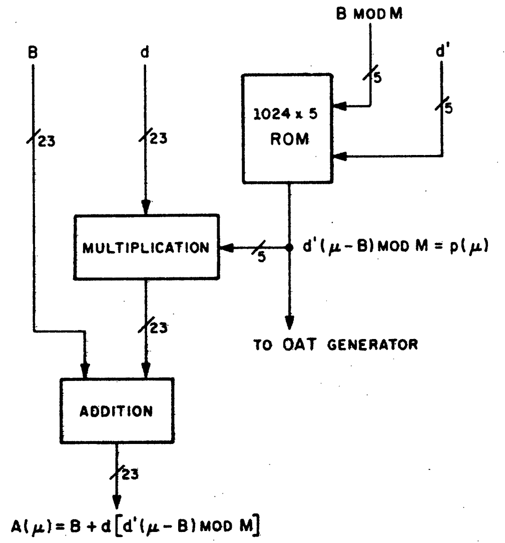


Figure 5. A Primary Address Generator

A secondary address generator for $A(\mu + 2)$ is shown in Figure 6. Notice that in equation (16) a test is required to determine the quantity added to (or subtracted from) $A(\mu)$. This test depends on the quantity $p(\mu) + 2d' = d'(\mu - B) \bmod M + 2d'$ (see equation (16)). $d'(\mu - B) \bmod M$ (available from the primary address generator) and d' are each 5-bit quantities and are used as address inputs to a 1024×5 ROM. The output of the ROM determines the test result and is used to multiply the necessary additive factor for the final adder. The other secondary address generators for $A(\mu + 1)$, $A(\mu - 1)$, and $A(\mu - 2)$ are similar to the one shown in Figure 6. However, the $A(\mu \pm 1)$ generators only need 2-way multiplexers and a one-bit wide decision ROM. (Through further simplification, these decision ROM's can be reduced to 512 words, so that the total decision ROM for four secondary generators is just 6×512 bits.) The primary and its four associated secondary address generators are all grouped together physically.

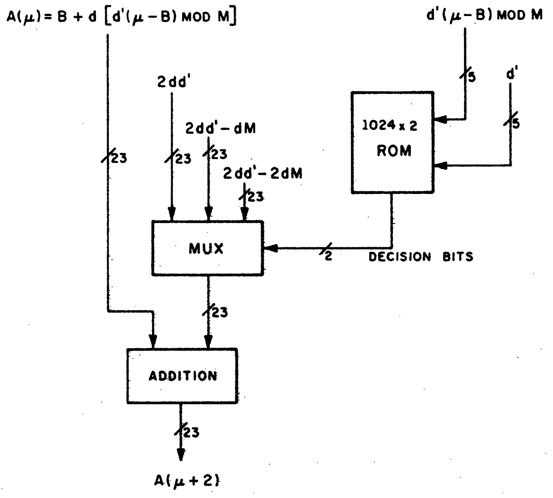


Figure 6. Secondary Address Generation for $A(\mu + 2)$

Generation of IAT's and OAT's is essentially the same or simpler than address generation. Only the values and number of bits change. One group of hardware, described above, generates the addresses, and a similar group of hardware generates both the IAT's and OAT's. Both groups of hardware form part of the Central Index Unit that will be described next.

3.1.3 The Central Index Unit

One of the components of the control unit is the Central Index Unit (CIU). The purpose of the CIU is to a) perform automatic indexing of multiple superwords; b) generate input and output alignment tags; and c) generate 4 initial memory addresses and indexing constants. The CIU can be divided into 4 major sections: 1. Descriptor Store Unit; 2. Descriptor Processing Unit; 3. IAT and OAT generators; and 4. Memory Address and Indexing Constant Generators. Figure 7 shows the organization of the above sections. The IAT, OAT and address generators were described in the previous section. The descriptor store unit stores up to 16 vector set descriptors (VSD). A simplified descriptor's contents are shown in Figure 3(b).

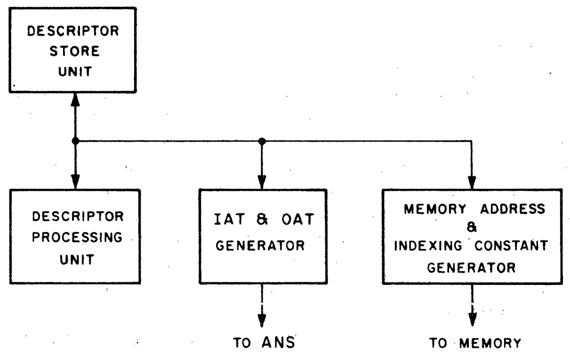

DESCRIPTOR STORE UNIT
DESCRIPTOR PROCESSING UNIT
IAT & OAT GENERATOR
MEMORY ADDRESS & INDEXING CONSTANT GENERATOR
TO ANS
TO MEMORY

Figure 7. Central Index Unit

A superword access requires an Indexing Event in the CIU. During this event the descriptor is updated by the Descriptor Processing Unit to reflect the access. The processing depends upon the kind of descriptor as well as the data values within the descriptor. For example, suppose we have a two-dimensional vector set operand (e.g., $K > 1$). The processing will be as follows:

If the length L is longer than a superword (N), then the descriptor values are updated as follows. These updates are performed after each superword access is initiated.

$$b \leftarrow b + d * N$$
$$B \leftarrow B$$
$$L \leftarrow L - N$$
$$K \leftarrow K$$

However, if the length L of the last access was equal or less than a superword (N), then the next superword should come from the next vector in the vector set. The appropriate update equations are as follows.

$$b \leftarrow B + D$$
$$B \leftarrow B + D$$
$$L \leftarrow LL$$
$$K \leftarrow K - 1$$

These actions cause the length to be reset to the initial length (LL), and increment the base address (B) to the base of the next vector in the set.

3.2 Other Functions of the Alignment, Indexing, and Memory System

As we mentioned above, the AIM system is also responsible for a number of other functions. In order to facilitate the smooth flow of data through the vector processing elements, forms of data other than linear N-vectors must be handled more or less automatically. These functions will be discussed next.

3.2.1 Automatic Padding of Short Superwords

As mentioned earlier, not all superwords in a vector operation are a full 16 words. Internally in the BSP System, the Arithmetic Elements (AE) recognize a "NULL" operand. The array memory also recognizes the NULL operand and inhibits a store when a NULL operand is encountered. The control unit automatically causes the alignment networks to pad short superwords by selecting NULL operands during input and output alignment events.

3.2.2 Vector Element Conflict

In the memory storage scheme, if d mod M = 0, all the elements of the linear vector lie in the same memory module. This is referred to as a vector element conflict condition. In this case, the access to the memory has to be sequential. In the BSP System, this condition is handled by forcing superword size equal to 1. Thus the BSP System automatically adapts to this case without any software or other interruption.

3.2.3 Inner and Outer Loop Optimization

Consider the following FORTRAN program setment.

```
      DO  10  I = 1, 14
      DO  10  J = 1, 4
10    A(I,J) + B(I,J)
```

This program can be performed in a single BSP vector operation consisting of 14 superwords each of length 4. However, it is faster to execute the above program segment with inner and outer loops interchanged, using 4 superwords of size 14. The BSP optimizes these cases by using hardware detection of the fastest loop order from the parameters L and K of a VSD. Of course, not all loops can be interchanged, and a software check is made to allow the above optimization.

Space prevents us from describing all the other functions performed by the alignment and memory system. These functions include, among others, handling scalar data in vector operations, data compression and expansion, and mode vector operations.

4. Conclusion

In this paper we have shown one design for a conflict-free array access memory. This design is based on the use of a prime number of memories. Crucial to this design is the simplification of the indexing equations which allow most of the mod M operations and much of the other index calculations to be done with ROM's and other simple hardware. These simplifications were discussed in Section 3, along with a brief discussion of some of the necessary indexing hardware. Further details can be found in [LaVo77].

The design of this memory system fits nicely in the context of the Burroughs Scientific Processor ([Stok77], [KuSt79]). The vector machine instructions on this computer can encompass two levels of loop nesting, and the indexing hardware carries out the necessary addressing and alignment calculations automatically, once the initial vector set descriptors have been set up. One of the major problems with large vector computers has been that indexing overhead and memory access conflicts have a significant effect on overall vector performance. By using the prime memory system and indexing hardware described in this paper, the BSP is able to execute vector instructions efficiently.

Acknowledgment

A number of people have contributed to this work. The overall design of BSP is due primarily to David Kuck and Richard Stokes. Special thanks go to Robert Cantarella, who made many significant contributions to the BSP design and especially to the alignment, indexing and memory system, and to Om Gupta and Ram Gupta.

References

[ArHS76] E. Artzy, J. A. Hinds and H. J. Saal, "A Fast Division Technique for Constant Divisors," Comm. of the ACM, Vol. 19, No. 2, pp. 98-101, Feb. 1976.

[Batc77] K. E. Batcher, "The Multi-dimensional Access Memory in STARAN," IEEE Trans. on Computers, Vol. C-26, No. 2, pp. 174-177, Feb. 1977.

[BuKu71] P. Budnik and D. J. Kuck, "The Organization and Use of Parallel Memories," IEEE Trans. on Computers, Vol. C-20, No. 12, pp. 1566-1569, Dec. 1971.

[Feng74] T-y. Feng, "Data Manipulation Functions in Parallel Processors and Their Implementations," IEEE Trans. on Computers, Vol. C-23, No. 3, pp. 309-318, Mar. 1974.

[GaRu78] D. D. Gajski and L. R. Rubinfield, "Design of Arithmetic Elements for Burroughs Scientific Processor," Proc. 4th Symp. on Computer Arithmetic, pp. 245-256, 1978; also, Proc. 1978 LASL Workshop on Vector and Parallel Processors, 1978.

[KuSt79] D. J. Kuck and R. Stokes, "The Burroughs Scientific Processor (BSP)," submitted for publication, 1979.

[Lang76] T. Lang, "Interconnections Between Processors and Memory Modules Using the Shuffle-Exchange Network," IEEE Trans. on Computers, Vol. C-25, No. 5, pp. 496-503, May 1976.

[LaSt76] T. Lang and H. S. Stone, "A Shuffle Exchange Network with Simplified Control," IEEE Trans. on Computers, Vol. C-25, No. 1, pp. 55-65, Jan. 1976.

[LaVo77] D. H. Lawrie and C. R. Vora, "Multi-dimensional Parallel Access Computer Memory System," U. S. Patent No. 4,051,551, Sept. 27, 1977.

[LaVo79] D. H. Lawrie and C. R. Vora, "The Prime Memory System for Array Access," submitted for publication, IEEE Trans. on Computers, Mar. 1979.

[Lawr75] 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.

[Orcu76] S. E. Orcutt, "Implementation of Permutation Functions in ILLIAC IV-Type Computers," IEEE Trans. on Computers, Vol. C-25, No. 9, pp. 929-936, Sept. 1976.

[Shap75] H. D. Shapiro, "Theoretical Limitations on the Use of Parallel Memories," Ph.D. thesis, University of Illinois at Urbana-Champaign, Dept. of Computer Science Rpt. No. 75-776, Dec. 1975.

[Sieg77] H. J. Siegel, "Controlling the Active/Inactive Status of SIMD Machine Processors," Proc. 1977 Int'l. Conf. on Parallel Processing, p. 183, Aug. 1977.

[Stok77] R. A. Stokes, "Burroughs Scientific Processor," Invited paper, in High Speed Computer and Algorithm Organization, pp. 85-89, Academic Press, Inc., 1977.

[Ston71] H. S. Stone, "Parallel Processing with the Perfect Shuffle," IEEE Trans. on Computers, Vol. C-20, No. 2, pp. 153-161, Feb. 1971.

[Swan74] R. C. Swanson, "Interconnections for Parallel Memories to Unscramble p-Ordered Vectors," IEEE Trans. on Computers, Vol. C-23, No. 11, pp. 1105-1115, Nov. 1974.

SESSION 4: PERFORMANCE

EMPIRICAL RESULTS ON THE SPEED, EFFICIENCY, REDUNDANCY AND QUALITY OF PARALLEL COMPUTATIONS*

Ruby Bei-Loh Lee
Computer Systems Laboratory
Stanford University
Stanford, California 94305.

Abstract -- The purpose of this paper is to present empirical results on the performance of parallel computations, with respect to various performance criteria, under different assumptions of the underlying computer architecture. The performance criteria used are the Parallel Index, the Speedup, the Utilization, the Efficiency, the Redundancy, the Compression and a definition of the Quality of the resultant computation. The underlying architectures assumed are parallel processor organizations of both the SIMD and MIMD varieties, with limited and unlimited degrees of physical parallelism.

1 Introduction

Computer architectures incorporating multiple processors which execute in parallel are being designed to speed up the execution-time, for better cost-performance, greater reliability and modularity. The trends appear to be towards special purpose, scientific supercomputers on the one hand, and towards general purpose multiple microprocessor systems with high performance to cost ratios, on the other hand. Although the decreasing cost and size of processors makes it feasible to consider using a large number of processors in a computer organization even at reduced efficiency of each component processor, it is important to estimate the effective speed actually attainable, over a representative set of computations. The sample considered in this paper may be described as existing general technical computations drawn from military, commercial and academic environments. It is emphasized that we are not interested in the maximum or minimum performance for any individual computation, but in the average performance over all the computations.

A computer organization with p parallel processors will rarely attain its maximum parallel execution bandwidth of p operations per time-unit, or a speedup in execution-time of p times that of the uniprocessor organization, due to both logical and physical constraints on the parallel execution of operations. Logical constraints on parallelism include intrinsic data-dependencies, control dependencies and operator precedences in the program, which force a sequential chain of execution amongst the dependent operations, and hence limit the number of operations which may be executed in parallel [9]. Physical constraints on parallelism include the maximum number of processors available in the architecture, the control restrictions on the different types of operations which may be executed simultaneously, and the delays due to the communication and competition amongst the interacting components in the computer organization. The empirical results in this paper account for the logical constraints and the first two physical constraints mentioned. For tractability, the experiments assume that the rest of the system, like effective memory-processor and processor-processor bandwidths, are balanced with respect to the execution-bandwidth of the parallel processors. This may even be considered an advantage since the results are then independent of the specific machine implementation. The empirical performance results in this paper should be interpreted as the best performance results expected with current techniques for parallelism exposure in Fortran programs [5-8, 2], assuming no delays due to the cooperation and competition amongst the components of the parallel processor organization. The results would be degraded if communication delays within the system are considered, but at the same time, the results would probably be improved by the explicit specification of parallel programs or by even better algorithms for the automatic conversion of serial programs to parallel computations.

Some of the questions we ask are: What is the performance of a computer organization with a limited number of parallel processors in the architecture? What is the performance in the idealized case where the number of processors is essentially unlimited? Are there severe performance degradations when only one type of operation may be executed simultaneously in one time-unit by the active processors?

2 Model and Definitions

A parallel computation is a sequence of steps, where each step consists of i operations which may be executed simultaneously, by i parallel processors. A step with i simultaneous operations is said to have degree of parallelism i, $1 \leq i \leq P$, where P is the maximum degree of parallelism in any step of the computation. The logical parallelism or minimax degree of parallelism, P', is the smallest maximum number of processors required by the computation in order to achieve its minimum execution time, Tmin.

*This work was supported in part by the Office of Naval Research under Contract N00014-79-C-0516.

A parallel processor organization is a computer organization with multiple processors, each of which is capable of executing one operation in one time-unit. Each processor is also capable of executing the whole repetoire of operations. An SIMD (Single Instruction Multiple Data) organization is a parallel processor organization where only one type of operation may be executed by the active processors in any one time-unit. An MIMD (Multiple Instruction Multiple Data) organization is a parallel processor organization where more than one type of operation may be executed by different processors in the same time-unit [3].

A parallel processor organization with p processors available in the architecture is said to have limited physical parallelism of degree p, and denoted a p-limited architecture. A parallel processor organization which always has as many processors, P', as required by the computation in order to achieve its minimum execution-time is said to have unlimited physical parallelism, and denoted an unlimited architecture. A p-limited computation is a parallel computation executing on a p-limited architecture, and an unlimited computation is a computation executing on an unlimited architecture.

TOP-form. The TOP-form is a canonic form of parallel computations defined as the following 3-tuple:

$$(T(P),\ O(P),\ P)$$

where T(P) is the execution-time of the computation in steps, O(P) is the computation-size in number of operations executed, and P is the maximum degree of parallelism in the computation. $P=P'$, the logical parallelism, for unlimited computations, and $P=\min(P',p)$ for p-limited computations.

The TOP-form captures the fundamental difference in the dimensions of the parallel computation when compared to a serial computation. In a serial computation, since each operation takes one time-unit for execution, the execution-time and computation-size have the same value, $T(1)=O(1)$. But in a parallel computation where $P>1$, the execution-time and computation-size necessarily have different values, $T(P)<O(P)$, forming two distinct dimensions of a parallel computation. The maximum degree of parallelism forms a third variable dimension in a parallel computation.

Equivalence, Optimality and Acceptability. Computations are said to be equivalent if given the same inputs, they always produce the same outputs. The internal algorithms and intermediate results in equivalent computations need not be the same.

An optimal serial computation is defined as a serial computation with the minimum computation-size, Omin. An optimal parallel computation is defined as a minimum-time minimax-parallel computation, which is a computation that achieves the minimum execution time, Tmin, using the minimax degree of parallelism, P'. Further discusssion on these definitions of optimality are available in [9].

A serial computation is said to be acceptable for comparison with a parallel computation if $O(1)\leq O(P)$. Otherwise, the O(P) operations of the parallel computation may be executed one at a time to obtain a shorter serial execution time and computation size. A parallel computation is said to be acceptable for comparison with a serial computation if $T(P)\leq T(1)$. Otherwise, the $O(1)=T(1)$ operations of the serial computation may be executed using one processor to get a shorter parallel execution time. Hence, we propose the following:

Principle of Acceptable Parallel-Serial Comparisons. A performance comparison of a parallel computation with an equivalent serial computation is said to be acceptable iff

$T(P) \leq T(1)$ and $O(1) \leq O(P)$, or equivalently

$T(P) \leq O(1) \leq O(P)$, if each operation takes one time-unit for execution.

This principle of acceptable parallel-serial comparisons is necessary to ensure that any measured performance improvements are due solely to parallel versus serial processing, rather than due to other factors. For example, the Speedup in execution time may be greater than p, the number of processors available in the architecture, when a parallel computation is compared with an unacceptable (nonoptimal) equivalent serial computation. Part of the performance improvement in this case is due to the optimization of a relatively inefficient serial computation. Similarly, the Speedup in execution time may be less than one, if the parallel computation entering into the comparison is unacceptable.

Performance Measures. The TOP-form of a parallel computation and its equivalent serial size form the smallest set of parameters for the evaluation of all the performance criteria considered in this paper. Basically, the performance criteria fall into four categories: the speed of execution given by the Parallel Index and Speedup measures, the utilization of the processor-time resource given by the Utilization and Efficiency measures, the Compression (or conversely, the Redundancy) in the size of the computation, and the resultant Quality of processing.

The Parallel Index and Speedup measure the average and effective speed, respectively, of the parallel computation in operations executed per time-unit:

$PI(P) = O(P)/T(P)$,
$S(P,1) = O(1)/T(P) = T(1)/T(P)$.

The Speedup is defined with respect to the computation-size of an equivalent serial computation, and takes into account the extra operations introduced into a parallel computation to reduce its execution time. The Speedup may also be regarded as the ratio of the execution-time of the serial computation, to that of the parallel computation, making it equivalent to the definition found in [5].

The Parallel Index and Speedup may also be regarded as measures of the average and effective parallel execution bandwidths of the underlying parallel processor organization, during the execution of the given computation.

The Utilization and Efficiency measure the cost-effectiveness of the computation in the sense that they weigh the speed improvement with the number of processors required. They measure the performance of the parallel computation with respect to its use of the processor-time resource:

$$U(P) = O(P)/[P.T(P)], \quad E(P,1) = O(1)/[P.T(P)].$$

In figure 1, the Utilization is the proportion of the rectangle PxT(P) covered by busy processor-steps, i.e., those time-units where a processor is busy executing an operation. The Efficiency may be regarded as the ratio of the serial processor-time requirement over the parallel processor-time requirement, since T(1)=O(1).

The Redundancy measure is the ratio of the parallel computation-size, O(P), to the serial computation-size, O(1), of an equivalent serial computation. The Compression measure is the inverse ratio:

$$R(P,1) = O(P)/O(1), \quad \text{and} \quad C(P,1) = O(1)/O(P).$$

One significance of the Redundancy measure is that it relates the relative speed and efficiency measures, S and E, to the absolute speed and efficiency measures, PI and U:

$$S = C.PI = PI/R \text{ and } E = C.U = U/R$$

The Speedup, Efficiency and Compression measures compare serial to parallel execution-time requirements, processor-time requirements and computation-size requirements, respectively (see table 1). In an optimal serial computation, the execution-time, processor-time and computation-size requirement are each equal to Omin. The Quality measure is defined as an overall performance measure comparing serial to parallel computations with respect to these three requirements:

$$Q(P,1) = S.E.C = S.E/R = \frac{Omin^3}{T(P)^2.O(P).P}$$

Hence, the Quality measure is a more stringent measure of the performance improvement of parallel versus serial processing than the Speedup measure. One use of the Quality measure is to decide whether parallel processing is preferable to serial processing for a given program. For example, a computer installation may decide that parallel processing is desirable for a given program if the quality of processing increases by at least fifty percent ($Q \geq 1.5$).

In all acceptable parallel-serial comparisons, the following relationships hold:

$$1 \leq S(P,1) \leq PI(P) \leq P$$
$$1/P \leq E(P,1) \leq U(P) \leq 1$$
$$1 \leq R(P,1) \leq 1/E(P,1) \leq P$$
$$1/P \leq E(P,1) \leq C(P,1) \leq 1$$
$$Q(P,1) \leq S(P,1) \leq PI(P) \leq P$$

PI is an upper bound for S, and U is an upper bound for E, with equality iff O(P)=O(1) so that R=C=1. The standard of comparison, an optimal serial computation, has PI, S, U, E, R, C and Q all equal to unity.

The last relationship above shows four successively refined measures of the performance improvement of parallel versus serial processing. First, P=min(p,P') indicates the maximum processor bandwidth, or the maximum speed of the parallel computation. Then, the Parallel Index indicates the average processor bandwidth, or average speed, of the computation. Third, the Speedup indicates the effective processor bandwidth, or effective speed, of the computation. Finally, the Quality is a single performance measure that takes into account mainly the speed improvement, but also the efficiency and the redundancy of parallel versus serial processing.

TABLE 1: OPTIMIZATION OF PERFORMANCE MEASURES AND TOP-FORM PARAMETERS

Performance Measure		TOP-form parameter
(1) maximizing Speedup	=	minimizing T(P)
(2) maximizing Efficiency	=	minimizing PxT(P)
(3) maximizing Compression (minimizing Redundancy)	=	minimizing O(P)
(4) maximizing Quality	=	all of the above: minimizing $T^2.O.P$ (minimizing each component of TOP-form with emphasis on time)

Measures of Central Tendency. To characterize the performance of a set of computations rather than an individual computation on a given parallel organization, measures of the central tendency of the data are desired. In table 2, the sample mean of the performance measures and TOP-form parameters are given, and in table 3, the median values are given. In table 4, another measure of central tendency is introduced, called the aggregate performance measures [9]. The aggregate performance measures are performance measures defined for the aggregate computation, which is the computation consisting of every step of every computation in the set of computations. In other words, the aggregate computation is the end-to-end concatenation in time of all the computations in the set. In parallel-serial comparisons, the aggregate performance measure is a ratio of the sum of the requirements of all the serial computations in the sample, divided by the sum of the corresponding requirements of all the equivalent parallel computations. For example, the aggregate speed measures for a set of computations are defined as:

$$S^a = \Sigma \ T(1) \ / \ \Sigma \ T(P) = \overline{T(1)} \ / \ \overline{T(P)}$$

$$PI^a = \Sigma \ O(P) \ / \ \Sigma \ T(P) = \overline{O(P)} \ / \ \overline{T(P)}$$

PI^a and S^a may be regarded as the average and effective parallel execution bandwidths, respectively, for a set of computations, in a single program environment, i.e., when the execution of the next computation in the set does not start till the execution of the current computation has ended. Since not all the processors are utilized by a computation during all steps of its execution, the introduction of multiprogramming could reduce the overall execution-time of all the computations in the set, though it cannot reduce further the execution time of any individual computation. Hence, PI^a and S^a may be interpreted as lower bounds for the average and effective parallel execution bandwidths in a multiprogramming environment.

Whereas the mean performance measures give equal weight to each computation in the set, the aggregate performance measures tend to weigh each computation by the relative magnitudes of its computation-size and execution-time. It seems reasonable that the overall performance should be more affected by a longer computation than by a shorter one. In general, the aggregate performance measures indicate the performance of all computations considered as a whole, whereas the mean performance measures indicate the expected performance for any one computation in the set of computations.

3 The Experiments

The raw data is obtained from runs of the Illinois Analyser version 2 [7], which transforms ordinary serial programs into equivalent parallel computations. The Analyser incorporates sophisticated algorithms for recognising serial program constructs and converting these to parallel program constructs for fast and efficient parallel execution. Version 2 (1978) of the Analyser differs from version 1 (1973) [5] mainly in the improved handling of linear recurrences [2] found in the serial program.

Existing Fortran programs (ANSI standard) were obtained from various locations, like the Air Force Weapons Laboratory, Burroughs Corporation, the collected algorithms published by the ACM, a well-known scientific library of programs called EISPACK, some of the old programs from the Illinois Analyser Version 1 (prior 1973), and other miscellaneous sources. These programs are run through the Illinois Analyser version 2, which produces as output, the dependency graph of each program. This is then entered as input to simulators, which restructure the computation when necessary, to execute on either an unlimited MIMD organization, an unlimited SIMD organization, or a p-limited SIMD organization where $p=2^i$, for $i=1,2,\ldots,14$. Hence, 16 different parallel processor organizations are compared with the uniprocessor organization.

Various data on the nature of the serial program and its parallel equivalent are collected, from which we abstract sixteen sets of raw TOP-forms and the raw serial computation size, O(1), for each computation, to use as inputs to our analysis programs. First, a standardization procedure is performed, to ensure that only acceptable parallel-serial comparisons of performance are produced. Essentially, the standardization consists of estimating the optimal serial computation size and the optimal parallel TOP-form for each computation-architecture combination [9]. The performance measures are then calculated from these standardized TOP-forms. The rows labelled "SIMDB" and "MIMDB" in tables 2,3 and 4 refer to the unlimited SIMD and MIMD cases, respectively. The row labelled "M/S ratio" gives the statistics for the ratio of values in the unlimited MIMD over the unlimited SIMD cases, to compare the effect of the added control restriction of SIMD parallel architectures. All the statistics in the tables are calculated for the entire sample of 355 computations.

4 The Results

TOP-form parameters: When the same sample of 355 computations is executed with varying degrees of limited physical parallelism, both the mean and median execution-times decrease, and both the mean and median computation-sizes increase, as the number of processors increases. In each case, the sample mean is about two orders of magnitude larger than the sample median, indicating a distribution that is skewed to the right. The mean and median execution-times in an unlimited MIMD environment are about 60% of the corresponding execution-times in an unlimited SIMD environment.

The median values of the maximum number of processors required, P', indicate that if up to 64 parallel processors are available in the architecture, more than half the computations executed will utilize all the processors available during execution. For this sample of

TABLE 2: MEAN PERFORMANCE MEASURES AND TOP-FORM PARAMETERS

P	PI	S	U	E	R	Q	T	O	P'
1	1.00	1.00	1.000	1.000	1.00	1.00	47524.1	47524.1	1.0
2	1.46	1.36	0.732	0.682	1.08	0.95	28926.9	49427.4	1.7
4	2.53	2.18	0.633	0.546	1.20	1.36	14817.8	49745.2	3.3
8	4.31	3.49	0.539	0.436	1.35	2.02	8538.1	50819.3	6.5
16	7.19	5.57	0.449	0.348	1.52	3.07	5372.4	51359.2	12.4
32	11.73	8.73	0.366	0.273	1.72	4.69	3664.6	54437.8	23.0
64	19.04	13.47	0.297	0.210	2.51	7.14	2700.8	56381.2	42.0
128	28.72	20.10	0.224	0.157	3.62	10.45	2224.1	67413.5	72.2
256	44.85	29.48	0.175	0.115	5.87	15.07	2019.9	91305.2	119.1
512	68.83	44.73	0.134	0.087	6.59	23.22	1883.5	96505.7	199.7
1024	98.07	63.84	0.096	0.062	8.31	31.00	1803.6	120378.9	314.7
2048	136.27	89.12	0.067	0.044	8.50	40.60	1768.4	120494.6	480.5
4096	172.87	116.74	0.042	0.029	8.90	44.95	1754.6	121472.6	693.7
8192	220.70	136.51	0.027	0.017	11.91	37.75	1747.4	140873.1	1134.4
16384	282.62	180.96	0.017	0.011	16.64	51.11	1741.7	156339.9	1748.9
SIMDB	1792.65	187.68	0.309	0.261	145.20	118.53	1620.5	294950.9	235922.1
MIMDB	2524.83	364.01	0.248	0.199	145.23	168.12	1058.0	295011.7	236718.0
M/S	2.07	2.01	0.975	0.974	1.03	2.14	0.6	1.0	4.1

TABLE 3: MEDIAN PERFORMANCE MEASURES AND TOP-FORM PARAMETERS

P	PI	S	U	E	R	Q	T	O	P'
1	1.00	1.00	1.000	1.000	1.00	1.00	607.0	607.0	1
2	1.49	1.24	0.745	0.622	1.00	0.65	553.5	673.0	2
4	2.74	2.01	0.686	0.502	1.00	0.83	335.0	866.5	4
8	4.50	3.04	0.562	0.380	1.02	0.78	254.0	991.0	8
16	6.64	4.24	0.415	0.265	1.03	0.72	162.5	1005.5	16
32	8.57	4.80	0.268	0.150	1.03	0.49	132.5	1062.0	32
64	10.45	5.60	0.163	0.087	1.06	0.31	110.0	1217.0	64
128	11.50	5.79	0.090	0.045	1.07	0.19	87.0	1210.0	94
256	12.77	5.81	0.050	0.023	1.08	0.11	78.0	1198.5	100
512	13.80	6.11	0.027	0.012	1.09	0.06	69.0	1262.5	100
1024	14.06	6.11	0.014	0.006	1.09	0.03	66.5	1372.5	100
2048	13.96	6.27	0.007	0.003	1.08	0.01	65.0	1372.5	100
4096	13.96	6.36	0.003	0.002	1.08	0.01	63.0	1372.5	100
8192	14.36	6.36	0.002	0.001	1.09	0.00	61.0	1402.5	100
16384	14.36	6.76	0.001	0.000	1.09	0.00	61.0	1416.5	100
SIMDB	16.12	7.57	0.205	0.153	1.12	0.99	60.5	1451.0	100
MIMDB	29.16	12.65	0.164	0.117	1.13	0.95	32.0	1451.0	208
M/S	1.58	1.58	1.000	1.000	1.00	1.33	0.6	1.0	2

FIG. 2: PARALLEL INDEX vs. p

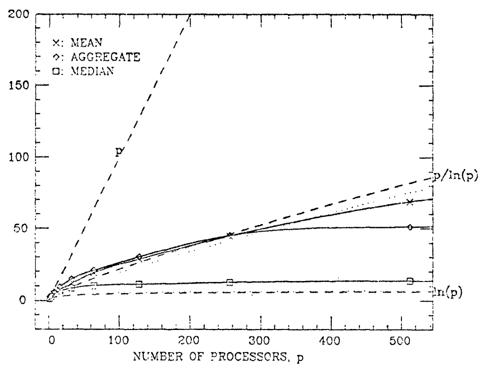


TABLE 4: AGGREGATE PERFORMANCE MEASURES

P	PI	S	U	E	R	Q
1	1.00	1.00	1.000	1.000	1.00	1.00
2	1.71	1.64	0.854	0.821	1.04	1.59
4	3.36	3.21	0.839	0.802	1.05	3.41
8	5.95	5.57	0.744	0.696	1.07	6.06
16	9.56	8.85	0.598	0.553	1.08	9.77
32	14.85	12.97	0.464	0.405	1.15	13.33
64	20.88	17.60	0.326	0.275	1.19	18.89
128	30.31	21.37	0.237	0.167	1.42	20.96
256	45.20	23.53	0.177	0.092	1.92	10.79
512	51.24	25.23	0.100	0.049	2.03	5.61
1024	66.74	26.35	0.065	0.026	2.53	2.79
2048	68.14	26.87	0.033	0.013	2.53	1.42
4096	69.23	27.09	0.017	0.007	2.56	0.71
8192	80.62	27.20	0.010	0.003	2.96	0.34
16384	89.76	27.29	0.005	0.002	3.29	0.17
SIMDB	182.01	29.33	0.004	0.001	6.21	1.19
MIMDB	278.83	44.92	0.005	0.001	6.21	1.47
M/S	1.53	1.53	1.198	1.197	1.00	1.24

FIG. 1: A PARALLEL COMPUTATION

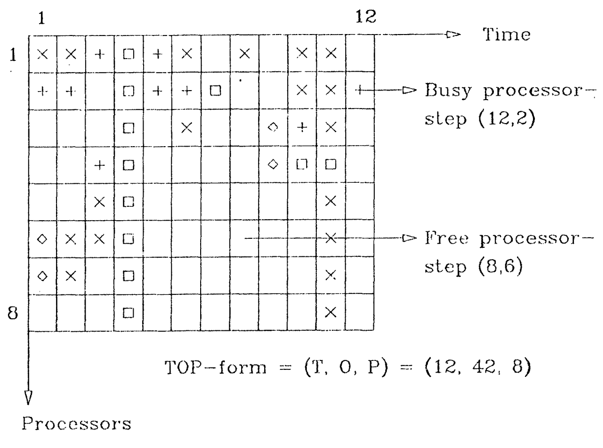

1
12
Time
1
8
Busy processor-step (12,2)
Free processor-step (8,6)
TOP-form = (T, O, P) = (12, 42, 8)
Processors

FIG. 3: SPEEDUP vs. p

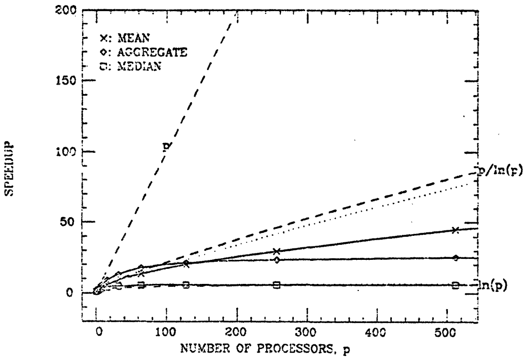


FIG. 7: MIMD UTILIZATION & EFFICIENCY vs. P′

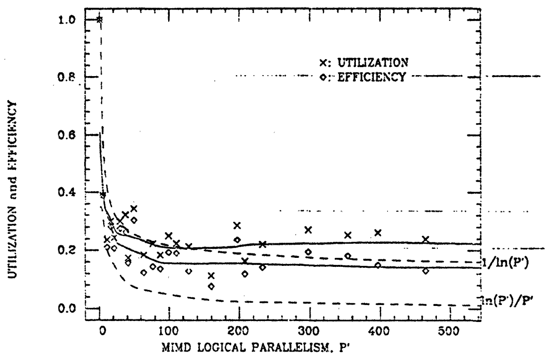


FIG. 4: MIMD SPEED VS. P′

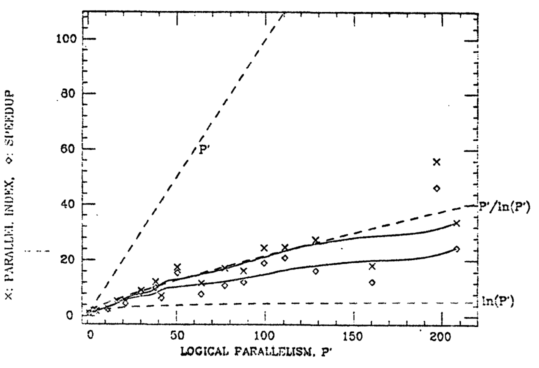


FIG. 8: REDUNDANCY vs. p

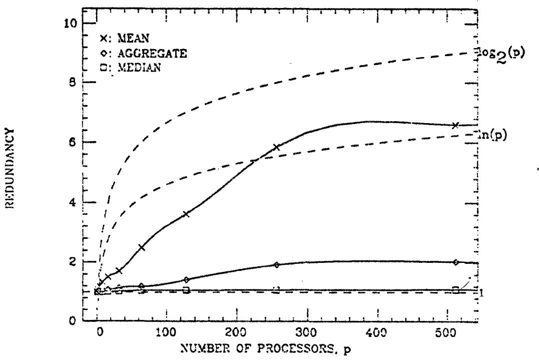


FIG. 9: MIMD AND SIMD REDUNDANCY vs. P′

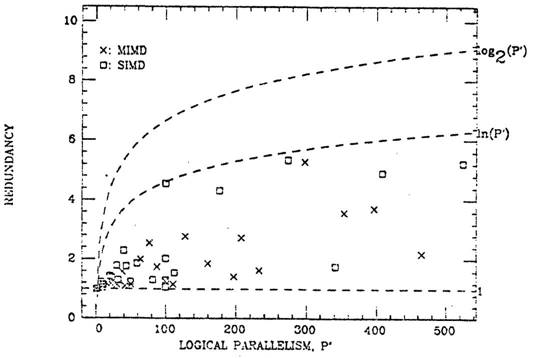


FIG. 5: UTILIZATION vs. p

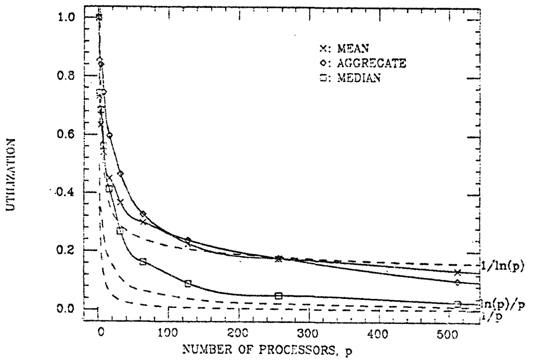


FIG. 6: EFFICIENCY vs. p

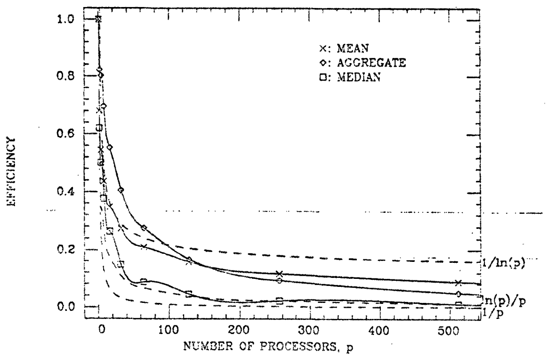
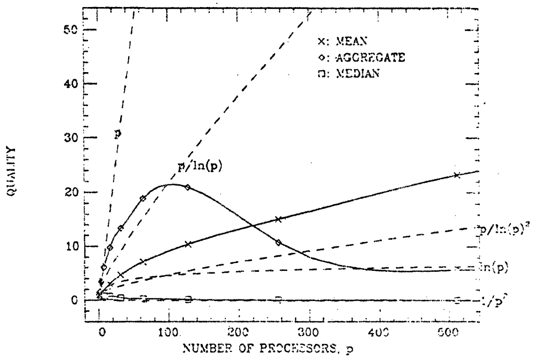


FIG. 10: QUALITY vs. p



computations executing under an SIMD environment, half the computations require not more than 100 parallel processors. Also, half the computations use twice as many processors when an MIMD organization is assumed than when an SIMD option is assumed.

Parallel Index and Speedup. In [9], probabilistic hypotheses on the parallelism distribution in computations were proposed, yielding simple characterizations of the average speed, PI. Necessary and sufficient conditions were given for PI to have a value on the order of $P/\ln(P)$, and for PI to have an upper bound of $P/\ln(P)$. The natural logarithm function of P, $\ln(P)$, is used as an approximation to the Pth. Harmonic number, $H(P)$. The predicted values of PI, which are upper bounds for S in all acceptable parallel-serial comparisons, agree well with empirical observations.

In figure 2, the average and aggregate values of PI are well approximated by the $p/\ln(p)$ curve, for $p \leq 300$. For larger p, the average and aggregate values of PI are less than $p/\ln(p)$. Similarly, in figure 3, the average and aggregate values of S are well approximated by $p/\ln(p)$, for $p \leq 100$, and less than $p/\ln(p)$ for larger p. The $\ln(p)$ curve forms a lower bound in each case.

The mean, aggregate and median PI values tend to run approximately parallel to the corresponding Speedup values. Hence, the trend of these measures of central tendency (mean, median, aggregate) of the Speedup values are well predicted by the corresponding trend of the PI values, and vice versa.

Figure 4 plots the Parallel Index and Speedup values, averaged over every ten consecutive points, assuming an MIMD organization with unlimited physical parallelism. The solid lines are the smoothed curves automatically generated for the crosses (PI) and diamonds (S) by the plotting package [1], using a smoothing algorithm involving running means, running medians, quadratic interpolation and "hanning" [11]. There is excellent agreement between the observed average PI values and the $P'/\ln(P')$ curve, for this range of P'. The Speedup values tend to lie below the $P'/\ln(P')$ curve.

Similar plots for the unlimited SIMD computations indicate that there are no significant differences in the trends of the observed PI and S values when compared with the unlimited MIMD case.

Binomial tests [4, 9] with a significance level of 5% were performed which indicate that the majority (more than 50%) of computations encountered have Parallel Indices and Speedups less than $P'/\ln(P')$, in an SIMD or MIMD environment with unlimited physical parallelism. In fact, for unlimited MIMD computations,

$$\mathrm{Prob}\{S(P',1) < P'/\ln(P')\} > 0.75$$

In an SIMD environment with p-limited physical parallelism, the majority of computations have Parallel Indices and Speedups less than $p/H(p)$ for p sufficiently large ($p \geq 64$ processors for PI, $p \geq 16$ processors for S). More than 80% of the computations have Speedups less than $p/H(p)$, for $p \geq 256$ processors.

Utilization and Efficiency. For any individual computation:

$$PI(P) = U(P).P, \quad \text{and} \quad S(P,1) = E(P,1).P,$$

where P may be interpreted as the physical parallelism, p, in a p-limited parallel architecture, or as the logical parallelism, P', in an unlimited parallel architecture.

This relationship between the Parallel Index and Utilization measures, and between the Speedup and Efficiency measures also holds for the corresponding pairs of mean, median and aggregate values for a set of p-limited computations. For example,

$$\overline{PI(p)} = \overline{U(p,1)} \;.\; p, \quad \text{and} \quad \overline{S(p,1)} = \overline{E(p,1)} \;.\; p.$$

It is sometimes hypothesized that the Speedup is a linear function of p, of the form, k.p, for some constant $k \leq 1$. However, figures 5 and 6 clearly show that the mean, aggregate and median values of U and E are not constants independent of p. Hence, it is impossible for the corresponding values of PI and S to be linear functions of p. In fact, the mean, median and aggregate values of U and E are decreasing convex functions of p, implying that the corresponding values of PI and S are increasing concave functions of p. Note that the $p/\ln(p)$ characterization of PI and S is an increasing concave function of p.

In the case of unlimited physical parallelism, the smoothed curves of U(P') and E(P',1) are well approximated by the $1/\ln(P')$ curve (figure 7). However, if the first data point is ignored, it is not clear that U(P') and E(P',1) are necessarily decreasing convex functions of P'. In fact, they could be described as very slightly decreasing linear functions of P', which may even be considered constant functions, independent of P'. The first data point plotted has P', U(P') and E(P',1) all identically equal to 1. The ten computations represented by this data point are those where the optimal parallel computation is in fact a serial computation, since the minimax degree of parallelism, P', is equal to 1. Except for this first data point, all the other averaged U and E values lie between 0.1 and 0.4. These empirical observations suggest the following

Hypothesis on the Conservation of Processor-Time. The ratio of the processor-time requirement of an optimal serial computation to that of an equivalent optimal parallel computation is:

$$E(P',1) = \frac{Qmin}{P' x Tmin} = k, \quad \text{where } k \leq 0.5.$$

This hypotheses on the conservation of processor-time does NOT imply that when more processors are available to execute a given computation, then the execution-time will decrease accordingly, so that the processor-time requirement stays constant. Rather, it implies that the optimal parallel processor-time requirement is more than twice the optimal serial processor-time requirement, and the ratio of the two quantities appears to be fairly constant over many different computations.

There are no major differences in the utilization of the processor-time resource, when the computations are structured for an SIMD organization rather than an MIMD organization, with unlimited physical parallelism.

Redundancy. In the empirical results, the median redundancy is less than 1.15 for all MIMD and SIMD computations, with unlimited and limited degrees of physical parallelism. Hence, half the computations achieve a parallel execution-time less than the serial execution-time, with the introduction of less than 15% of redundant operations compared with the serial computation size. Although the mean redundancy is less robust than the median redundancy to extreme values, it is less than O(log(P)) (figures 8 and 9).

A variable X is said to be positively associated or in agreement, with another variable Y if large values of X tend to occur with large values of Y, and small values of X tend to occur with small values of Y. Similarly, X is said to be negatively associated, or in disagreement, with Y if large values of X occur with small values of Y, and small values of X occur with large values of Y. The Spearman rank correlation coefficient, R, may be used to test the degree and direction of association between any pair of variables. The magnitude of R, $0 \leq |R| \leq 1$ gives the degree of association, and the sign of R gives the type (agreement or disagreement) of association.

From the tests of association based on the Spearman correlation coefficient in both unlimited SIMD and MIMD cases, the redudancy measure is found to be negatively associated with the Efficiency and Quality, positively associated with P' and the Parallel Index, and not associated with the Speedup. It has also been observed [10] that larger redundancies are associated with larger probabilities of numerical instability in the parallel computation, as compared with the serial equivalent. Hence, parallel computations with large redundancies should be avoided, since these tend to be associated with inefficient computations with low qualities and higher probabilities of numerical instability. Also, since the Redundancy measure is found to be independent of the Speedup measure, redundant operations should be introduced into a parallel computation only if this increases the Speedup (effective speed), and not just the Parallel Index (average speed), of the resultant parallel computation.

Quality. In the tests of association based on the Spearman correlation coefficient, the Quality measure is found to be independent of the logical parallelism, P', in both SIMD and MIMD computations, assuming unlimited physical parallelism. This is a desirable result for the chosen definition of the Quality measure, since the quality of processing should not be biased towards computations with either smaller or larger degrees of logical parallelism.

The empirical median quality decreases as the physical parallelism increases beyond p=4, and is less than one, in all cases. Hence, more than half the computations in each p-limited and unlimited SIMD and MIMD case have higher qualities when executed as a serial computation than when executed as a parallel computation.

Unlike the relationship between the mean quality and mean speedup, the definition of the aggregate quality does not constrain it to have an upper bound given by the aggregate speedup measure. In the sample of computations examined, the aggregate Quality increased in value, at approximately the same rate as the aggregate Speedup, up to p around 100 (figure 11). As p increased beyond this "saturation point", the aggregate Speedup began to level off and the aggregate Quality declined. This behaviour is representative of most individual and aggregate computations, and hence the Quality measure may be used to chose the optimal number of processors to use in executing a given computation, or set of computations.

Figure 12 shows the frequency distribution of the values of p at which the highest quality is attained for each computation in the sample. Almost half (46%) of the computations examined attain their highest quality value of one, at p=1 (serial computations). The next largest frequency occurs at p=16 and p=32, where about eight percent, each, of the computations attain their highest quality values. The cumulative relative frequency curve indicates that about ninety percent of the computations attain their highest quality values for p<256. If this sample is representative of computations in general, then the parallel processor organization need not have more than 256 processors, in order that ninety percent of the computations executing on it may attain their highest quality potential. It is an interesting coincidence that the ILLIAC IV, an SIMD machine, was originally designed to have a maximum of 256 parallel processors.

5 Conclusions

The characterization of the performance measures varies according to the maximum degree of parallelism, P. Hence, we define the following approximate ranges:

Let P=O(1) denote $1 \leq P \leq 5$, and $P=O(10^i)$ denote $5.10^{i-1} \leq P \leq 5.10^i$, for i=1,2,...

For p-limited architectures, the best characterization for the mean or aggregate values of the speed measures are (figures 13a,b,c):

Approx. range of p	Speed: PI, S
p = O(1)	O(p)
p = O(10) or O(100)	O(p/ln(p))
p = O(1000)	$O(\ln(p)) \leq PI, S \leq O(p/\ln(p))$
p = O(10000) or more	O(ln(p))

In the case of unlimited physical parallelism, PI and S are best characterized as O(P'/ln(P')), for all P'=O(1000) or less.

These empirical observations support the speed characterization of parallel computations given in [9]:

"For general technical computations, the measures of central tendency such as the mean, median and aggregate values of the Parallel Index and the Speedup, all lie between $k_1.\ln(P)$ and $k_2.P/\ln(P)$, where $0<k_1\leq 1$ and $k_2 \geq 1$. Furthermore, the majority of computations will also have individual PI and S values between these lower and upper bounds. P may be interpreted as either the logical parallelism P', in an environment with unlimited physical parallelism, or as the physical parallelism, p, for sufficiently large p, in an environment with limited physical parallelism. So,

$$\max(1, k_1.\ln(P)) \leq S(P,1) \leq PI(P) \leq \min(k_2.P/\ln(P), P)"$$

Suppose that k_1=0.5 and k_2=2. Then, for $P \geq 8$, ln(P)/2 is greater than 1, and 2.P/ln(P) is less than P. Hence, except for the smallest P values, the O(ln(P)) and O(P/ln(P)) bounds form increasingly tighter bounds for S and PI, as P increases, when compared with the absolute limits of 1 and P.

The empirical speed characterization may be used as a rough guide to the minimum number of parallel processors needed to attain a certain average or effective parallel execution bandwidth. For example, if an average parallel execution bandwidth of ten operations per time-unit is desired, then at least 36 parallel processors should be used, since p/ln(p) = 36/ln(36) = 10.05. This predicted speed of O(p/ln(p)) operations per time-unit for p=O(10) processors should be regarded as the expected speed potential, since in practice, interactions between processors, memories and other components of the computer organization will cause further performance degradations. Conversely, given a parallel processor organization with limited physical parallelism of degree p, the appropriate speed characterization given above may be used to estimate its expected speed potential.

For p-limited architectures, the best characterization for the efficiency measures, U and E, are obtained by dividing the corresponding values of PI and S by p. For example, if p=O(10) or O(100), U and E are characterized by O(1/ln(p)). Hence, if an efficiency of at least 25 percent is desired, then less than 60 parallel processors should be used, since 1/ln(p) = 1/ln(59) = 0.25, but 1/ln(60) = 0.24. For a smaller efficiency, more parallel processors may be used.

For p-limited architectures, the empirical mean and aggregate Utilization and Efficiency measures substantiate the observation that the corresponding mean and aggregate PI and S measures are increasing concave functions of p, like p/ln(p), and not increasing linear functions of p.

For unlimited architectures, the averaged Utilization and Efficiency measures defined with respect to the logical parallelism, P', suggested a hypothesis on the conservation of processor-time.

Parallel computations with large Redundancy measures should be avoided since these are associated with inefficient computations with low qualities and higher probabilities of numerical instability. Also, redundant operations should be introduced into parallel computations only if this decreases the parallel execution-time when compared with known equivalent serial execution-times.

The Quality measure may be used to choose the optimal number of processors to use in executing a given computation or set of computations.

The mean, median and aggregate values of the M/S ratios for the Speedup and Quality measures imply that the performance improvement of MIMD versus SIMD organizations is less than two and a half times. This same performance improvement may also be obtained by increasing the number of processors available in the architecture. For example, to upgrade the performance of a given p-limited SIMD architecture, the incremental cost of conversion to the less restrictive MIMD organization should be compared with the incremental cost of adding more parallel processors to the SIMD architecture.

Acknowledgements

I am indebted to Professor David Kuck, Robert Kuhn, Michael Wolfe, Bruce Leasure, Pen-Chung Yew and other colleagues at the University of Illinois, for permission to use the raw data generated by the Illinois Analyser version 2. The invaluable database built up at

Illinois represents probably the largest and most significant experimental effort on the speedup of existing serial programs by parallel processing. This paper represents only one possible study of a small subset of the data available in the Illinois database.

I also wish to thank Professor Michael Flynn of Stanford University for his advice and help.

References

[1] Chaffee, R.,B., "Top Drawer", CGTM No. 178, SLAC Computation Group, Stanford, California, revised August 1978.

[2] Chen,S.C., and Kuck,D., "Time and Parallel Processor Bounds for Linear Recurrence Systems", IEEE Transactions on Computers, c-24 no. 7, July 1975, pp.701-717.

[3] Flynn,M., "Some Computer Organizations and Their Effectiveness", IEEE Transactions on Computers, c-21, Sept 1972, pp.948-962.

[4] Gibbons, J. D., "Nonparametric Statistical Inference", McGraw Hill, 1971.

[5] Kuck,D., Budnik,P., Chen,S.C., Davis,E., Han,J., Kraska,P., Lawrie,D., Muraoka,Y., Strebendt,R., and Towle,R., "Measurements of Parallelism in Ordinary Fortran Programs", 1973 Sagamore Computer Conference on Parallel Processing.

[6] Kuck, D.J., "A Survey of Parallel Machine Organization and Programming", Computing Surveys, vol. 9 no. 1, March 1977, pp. 29-58.

[7] Kuck,D., Kuhn,R., Wolfe,M., Yew,P.,and Leasure,B., "A Database of Parallel Programs Automatically Generated from Ordinary Fortran Programs", private communication, October 1978.

[8] Leasure, B., "Compiling Serial Languages for Parallel Machines", Report No. 805, Department of Computer Science, University of Illinois at Champaign-Urbana, November 1976.

[9] Lee, R.B., "Performance Characterization of Parallel Processor Organizations", Ph.D. thesis, Stanford University, May 1980, distributed by University Microfilms International, 300 North Zeeb Road, Ann Arbor, Michigan 48106.

[10] Sameh,A., "Numerical Parallel Algorithms - A Survey", Proceedings of the Symposium on High Speed Computer and Algorithm Organization, April, 1977, (invited), Academic Press pub. 1977, (D. Kuck, D. Lawrie, A. Sameh, eds.).

[11] Tukey, J., "Exploratory Data Analysis", Addison-Wesley Publishing House, 1977.

FIG. 11: AGGREGATE SPEEDUP & QUALITY vs. p

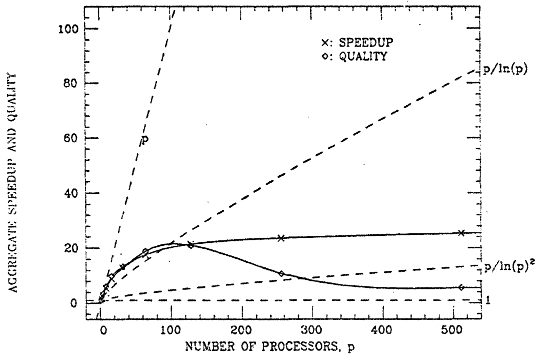


FIG. 12: PROCESSORS USED FOR HIGHEST QUALITY

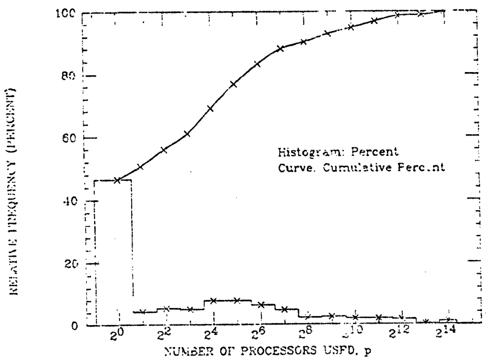
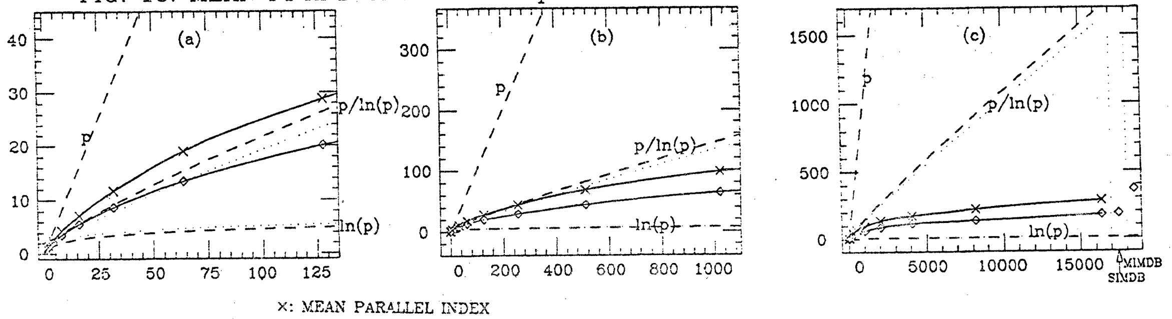


FIG. 13: MEAN PI AND SPEEDUP vs. p



×: MEAN PARALLEL INDEX
◇: MEAN SPEEDUP

PERFORMANCE EVALUATION OF PIPELINE ARCHITECTURES

Jamshed H. Mirza
Division of Computer Science
Polytechnic Institute of New York
Brooklyn, New York 11201

Summary

This paper is a report on the investigation of a generalized method for evaluation of pipeline processors under more realistic conditions than has been previously considered.

In Section I, our aim is to consider several alternatives for a performance measure for pipeline architectures, and come up with an index that clearly reflects how well the pipeline has been organized from a purely architectural point of view. The chosen index should be independent of all issues unrelated with the architectural sophistication of the design. Such a performance measure would be useful during the initial design stages for analysing a design to see if it is likely to meet the stated requirements. It can be used for studying and comparing several alternatives for a design, and as an aid to making relevant architectural decisions based upon that study. It can be used for studying and evaluating several structurally different pipeline architectures and to determine which, if any, is inherently superior under a given job environment.

In section II, a Markov Chain model is proposed for pipeline processors, and a method is suggested for determining the performance factor. This is followed by an illustrative example and some results of a preliminary analysis of the Texas Intsturments Advanced Scientific Computer (TI-ASC).

Section I: Our aim in this section is to inspect several alternative performance indices for pipeline architectures, and select one that best satisfies the following two conditions: (a) it should clearly reflect the sophistication of the design, viewed from a purely architectural point of view, and (b) it should be unaffected by aspects that are not relevant to the basic architecture. To satisfy these conditions, we need a performance factor that shows the increase in throughput rate attained by the pipelined architecture, as compared to an unpipelined architecture supporting strictly sequential, non-overlapped execution.

The throughput rate of a pipeline is directly affected by the number of segments in the pipe, the job characteristics, the pipe structure, and the pipe configurability. By jobs we mean units of computation that are separately initiated. In the usual instruction processing pipe, a job refers to a machine instruction. The jobs processed may be logically dependent or independent of each other. The pipe structure may be linear (when each segment receives control from only one segment and transfers control to only one other segment), or planar (when the previous restriction does not hold). The pipe may also be configurable (when the segment-interconnection structure is capable of taking different forms at different times), or non-configurable (when the interconnection structure is always constant).

The number of segments in the pipe will determine the extent of possible overlapped execution, and therefore the possible increase in throughput rate. Job characteristics will decide whether logical dependencies are possible. Dependent jobs imply more complex control requirements and lower utilization and throughput rates. Pipe structure, on the other hand, will decide whether job collisions are possible. While planar pipes allow the sharing of common segments among two or more functional pipes, they also introduce the possibility of job collisions (two or more jobs attempting to use a particular segment at the same time). Detection and avoidance of collisions also result in complex control requirements and reduced throughput rate. Configurable pipes also allow the sharing of segments while reducing the collision problems. However, they entail a reconfiguration overhead; a job requireing a configuration different from the one currently existing is held up until pipe is completely flushed. Thus the efficiency of a pipeline architecture tends to increase with the number of segments into which it is divided, while it is adversely affected by logical dependencies, collisions and reconfiguration overheads.

Previous attempts at evaluating pipeline architectures can be found in [1, 2, 3]. However, they all consider linear, non-configurable pipes and only [3] considers logical dependencies between jobs.

There are several alternatives for a performance measure for pipeline architectures that ought to be considered. Manufacturers have used the segment clocking rate to show the absolute raw processing rate of a pipelined machine. However that need not reflect the architectural sophistication. The number of segment in the pipe is an appealing parameter to be used as an index of the pipelining used. However, it fails to consider delays due to dependencies, collisions and reconfiguration. Moreover, one could increase the number of segments by introducing unnecessary non-compute segments which would result in no real gain in throughput rate; it may in fact deteriorate because dependent instructions may now have to wait even longer for the dependencies to be resolved. Utilization (average number of active segments at any time) would reflect the extent of performance deterioration because of various delays. However, it would not show the extent of segmenting employed. One could raise the apparent utilization by reducing the number of segments in the pipe although it would tend to reduce the throughput rate. Average instruction-initiation rate, used as a performance index gives a measure of the processing rate of the system if the various delays are considered. However it also fails to take into consideration the number of segments in the pipe.

CH1569-3/80/0000-0101$00.75 © 1980 IEEE

None of the above alternatives shows itself to be the unified quantity we are looking for-one that takes into account both the number of segments in the pipe as well as the delays due to dependencies, collisions and reconfigurations. The performance factor finally chosen is in fact a combination of two of the alternatives considered. It is given by:

$$PF = m_{avg}/d_{avg}$$

Here, m_{avg} is the number of segments in the different functional pipes weighted by their probability of being traversed. d_{avg} is the average job-initiation rate; it takes into account the relative frequency of occurence of the different instructions and the delays due to the various reasons. Note that often non-compute segments are inserted in the pipe in order to balance out the flow of jobs through the different functional pipes so as to avoid or reduce delays due to logical dependencies and collisions. Since these segments perform no computational step, they should not be considered in the determination of m_{avg}. However, since they help to reduce delays, these con-compute segments should be considered in the evaluation of d_{avg}.

The ratio of m_{avg} to d_{avg} effectively gives the average number of segments in the pipe that are active at any time (i.e., actually processing an instruction). This provides a measure of the speed-up realized over strictly sequential, non-overlapped execution. Factors that affect the absolute throughput rate, but have no bearing upon the pipeline characteristics of the system have no effect upon the performance factor.

Section II: In this section an analytic method, based on a Markov Chain model, is presented for analysis and performance evaluation of a pipeline architecture. The proposed method is general, and is applicable to multifunctional pipeline systems with $N \geqslant 1$ functional pipes. The pipes may be linear or planar, non-configurable or configurable, and the jobs processed may be mutually dependent or independent.

Let (S , P) be a multifunctional pipeline system containing N functional pipes.

$S = \{ S_1, S_2, S_3, \ldots , S_m \}$ is the set of m physically distinct segments in the system. Some of these segments may be functionally identical if, for overall efficiency reasons, certain relatively over-utilized segment-types are replicated. Also, some of the segments may be non-compute.

Each segment S_j is specified by a 3-tuple:

$S_j := (F(S_j), U(S_j), C(S_j))$ where $F(S_j)$ is the set of operations performed by S_j, and $U(S_j)$ and $C(S_j)$ identify the set of source ("used") and destination ("changed") elements referenced by S_j.

$P = \{P_1, P_2, P_3, \ldots , P_N\}$ is the set of N different functional pipes. They define a different path through a subset of the segments. Thus each functional pipe is defined by an ordered sequence

$$P_i := < S_{i,1} , S_{i.2}, \ldots , S_{i,m_i} > ,$$

where $S_{i,j} \in S$, and is the segment that a job traversing functional pipe P_i would be in during the jth active cycle after initiation.

Let $\alpha = < \alpha_1, \alpha_2, \alpha_3, \ldots \alpha_N >$, the expected job-profile be also known, where α_i is the probability that a job entering the system will traverse functional pipe P_i. We insist that

$$\sum_{i=1}^{N} \alpha_i = 1$$

If this is not the case (allowing for the possibility that at certain points in time no job arrives at the pipe for initiation), we intorduce a fictitious "null" pipe P_{N+1} which uses no segments and for which $\alpha_{N=1} = 1 - \sum \alpha_i$.

The proposed method of analysis assumes that all segments have identical and constant processing time so that the segments are clocked synchronously. This is a reasonable assumption that simplifies the analysis without limiting its applicability.

We also assume that a "Delay-Before-Initiation" (DBI) strategy is used for job initiation. According to such a strategy, all delays necessary for proper execution of a job are inserted before the job is actually initiated. When a job arrives at the pipe, it is delayed just sufficiently so that once it is initiated, it will not have to be held up at any segment within the pipe in order to resolve dependencies or avoid collisions with jobs that had entered the system earlier.

The DBI strategy is unlike the "Delay-After-Initiation" (DAI) strategy. In the case of the DAI strategy, all necessary delays are not inserted at just one point right at the beginning. Instead, jobs suffer short delays at several different stages in its path as required to remove the immediate threat of unresolved dependency or collision. For planar pipes DAI strategy in general results in fewer overall delays than the DBI strategy, but is much more difficult to analyse and to implement. Our evaluation method will therefore yield a wort-case of linear pipes, however, both strategies result in the same amount of delays, and therefore the assumption about the job-initiation strategy is not significant.

In a practical environment, the job arrivals are random and have no particular regularity. Moreover, logical dependencies will require that dependent sequences of instructions to be initiated in the order they arrive. Consequently, a first-come-first-served greedy scheduling strategy is the practical choice and is assumed here. We also assume that the processor is an SISD so that at most one instruction is initiated at each cycle. If we do allow for the possibility of more than one instruction being initiated at the same time, an extra degree of complexity would be added to the evaluation process.

Evaluation of m_{avg}: This is the weighted average of the number of segments in the N functional pipes and is given by

$$m_{avg} = \sum_{i=1}^{N} \alpha_i (m_i - m_i^{nc})$$

where

m_i = length of functional pipe P_i in number of segments

m_i^{nc} = number of non-compute segments in P_i

α_i = probability that a job will traverse pipe P_i

Evaluation of d_{avg}: This is the expected job-initiation rate. In the ideal case $d_{avg} = 1$. Under more realistic conditions, when delays due to various reasons exist, $d_{avg} > 1$, thus reducing the throughput rate. Therefore we need a representation for the state of the pipeline system that contains sufficient information to allow us to estimate this delay.

Associated with each functional pipe P_i is a Nxp binary matrix D_i called the Delay Matrix. Here N is the number of functional pipes, and

$p = \underset{i}{MAX} \{m_i\}$.

$D_i(j,k) = 1$ implies that scheduling a job P_j k time units after a job P_i has been initiated is to be prevented as it will violate logical dependency rules, cause a collision, or because a reconfiguration is required.

$D_i(j,k) = 0$ implies P_j may be initiated k times after P_i has been initiated.

Note that by a job P_i is meant a job that traverses pipe P_i. The Delay Matrix is an extention of the collision vector proposed by Davidson et.al. [4,5]. Since we are assuming a DBI strategy, information regarding logical dependencies and reconfiguration overheads can also be determined and incorporated into the Delay Matrices [6].

The state of the pipelined system with N functional pipes is then defined by Nxp binary matrix q such that

$q(j,k) = 1$ if and only if scheduling a job P_j k time units from the present will cause P_j to collide with some job within the pipe, or will allow some logical dependency of P_j on an earlier job within the pipe to go unresolved, or if P_j requires a reconfiguration.

Using such a Nxp binary matrix as the state of the system, we develop a Markov Chain model for the pipeline processor. The Markov Chain is derived by associating with the system an internal state every time a job is initiated. These states are called the initiation states. For each of the N job-types that could be waiting to be initiated, the current state provides information about the delay that would be incurred, and the next state to which the system transfers after the job is initiated. Note that at the very beginning, before any job has been initiated, the state of the system is $q_0 = \{ 0 \}$ (a matrix with all elements zero).

Definition: $q_0 = \{0\}$ is an initiation state.

If q is an initiation state, then so also are all states

$Shl(q,k_i) \bigcup D_i$ for all $1 \leq i \leq N$

where $q(i,k_i) = 0$

and $q(i, j) = 1$ for all $j < k_i$

Shl(q,k) is the Nxp binary matrix obtained by shifting each row of q, k positions to the left. Logically OR-ing the delay matrix D_i correspondint to the job P_i is initiated.

The behavior of a pipelie system (S , P) can now be completely described by the 6-tuple

$(Q , P , D , \sigma , \lambda , q_0)$,

where

Q : is the set of initiation states of the system

P : $\{ P_i, P_2, \ldots , P_N \}$ is the set of N different job-types (functional pipes)

D : $\{ D_1, D_2, \ldots , D_N \}$ is the set of N Delay Matrices associated with the N job-types

σ : $Q \times P \rightarrow Q$ is the next state function; given the present stat and the job-type, the function identifies the initiation state the system will enter when the job is initiated

λ : $Q \times P \rightarrow \{ 1,2, \ldots , p \}$ is the delay function; given the present state and the job-type, the function identifies the delay that is incurred before the job is allowed to be initiated.

q_0 : is the initial state of the system; $q_0 = \{0\}$

The functions σ and λ are defined by the following relations: Let the present initiation state be q, and the job waiting to be initiated by P_i.

If $q(i,k) = 0$

and $q(i,j) = 1$ for all $j < k$,

then $\sigma(q,P_i) = Shl(q,k) \bigcup D_i$ and $\lambda(q,P_i) = k$

With the help of the next-state function σ, and the expected job-profile α, we can find the state transition matrix $T = \{t_{i,j}\}$. T can be shown to be a stochastic matrix. Algorithms to determine Q , σ , λ, and T are given in [6].

The following theorems are proved regarding the system:

(a) The set of states Q contains one and only one ergodic set $\hat{Q} \subseteq Q$

(b) If any $D_i = \{0\}$ then $\hat{Q} = Q$

A fast algorithm is suggested in [6] for determining the transition states and the ergodic states.

Let $\hat{Q} = \{ \hat{q}_1, \hat{q}_2, \hat{q}_3, \ldots , \hat{q}_n \}$ be the ergodic set and $\hat{T}$ be the be the corresponding nxn transition matrix (obtain- from T by deleting rows and columns corresponding to the transition states). At steady-state, the system will be in the ergodic set; let

$\hat{\pi} = < \hat{\pi}_1, \hat{\pi}_2, \hat{\pi}_3, \ldots, \hat{\pi}_n >$ be the steady-state state-probability vector which can be determined by solving the set of equations:

$$\hat{\pi} \times \hat{T} = \hat{\pi}$$

and
$$\sum_{i=i}^{n} \hat{\pi}_i = 1$$

Let $\hat{\lambda}$ be the nxN delay function matrix corresponding to the n ergodic states; it is obtained from the delay function λ .

Then d_{avg} can be shown to be:

$$d_{avg} = \pi \times \lambda \times (\alpha)^{\text{transpose}}$$

A simple extention of the evaluation method, allows pipelined vector processors to be handled [6].

The following is a simple example to illustrate the performance evaluation method developed here. The hypothetical pipeline processor being considered here is planar, non-configurable and multifunctional with two functional pipes.

Processor: $= (S , P)$

$$S = \{S_1, S_2, S_3, \cdots , S_9 \}$$

$$P = \{ P_1, P_2, \}$$

$$P_1 = < S_1S_2S_3S_3S_4S_5S_9 >$$

$$P_2 = < S_1S_6S_7S_6S_2S_8S_9 >$$

Based on this information and the complete specification of the 3-tuples corresponding to each segment S_i (not shown here), the Delay Matrices are found to be :

$$D_1 = \begin{pmatrix} 1100000 \\ 0000000 \end{pmatrix} \qquad D_2 = \begin{pmatrix} 0010000 \\ 0100000 \end{pmatrix}$$

The state-diagram showing the behavior of the system is shown in the figure.

$$T = \begin{array}{c|cccccc} & q_0 & q_1 & q_2 & q_3 & q_4 & q_5 \\ \hline q_0 & & .3 & .7 & & & \\ q_1 & & .3 & & .7 & & \\ q_2 & & & & & .3 & .7 \\ q_3 & & .3 & & & & .7 \\ q_4 & & .3 & .7 & & & \\ q_5 & & & .7 & & .3 & \end{array}$$

The Ergodic set is $\hat{Q} = \{q_1,q_2,q_3,q_4,q_5\}$, and the steady-state probability vector is

$$\hat{\pi} = \langle \hat{\pi}_1 \quad \hat{\pi}_2 \quad \hat{\pi}_3 \quad \hat{\pi}_4 \quad \hat{\pi}_5 \rangle$$
$$= \langle 0.114 \quad 0.332 \quad 0.080 \quad 0.186 \quad 0.288 \rangle$$

$d_{avg} = 1.7374$ $\qquad$ $m_{avg} = 7.0$

PF $= 4.029$

Thus, for the expected job-profile α, the effective processing power of the pipelined processor is about 4 times that of a corresponding non-pipelined machine.

Table I shows some of the results of an analysis of the Arithmetic-Logic Unit of the TI-ASC. This unit is configurable and requires the complete pipe to be flushed everytime a new configuration is to be set up. The expected performance in a scalar processing environment, in a vector processing environment of several different expected vector lengths, and the ideal throughput potential are shown. These are compared with the expected performance if the ALU was in fact not made configurable and hence not burdened by the reconfiguration overheads. As can be seen, the improvement in a scalar-processing environment is quite significant, but is insignificant in a vector environment with even moderately large vector lengths. A more detailed study and relevant discussion will be found in [6].

TABLE I : ANALYSIS OF THE TI-ASC ALU
(10 Functional Pipes considered)

Processing Environment	Configurable	Non-Configurable	Improvement*
Scalar	1.190476	2.869866	2.410688
Vector(L=10)**	2.986179	3.995209	1.337900
Vector(L=100)	4.362761	4.530451	1.037284
Vector(L=1000)	4.614627	4.633850	1.004166
Vector(ideal)***	4.645000	4.645000	1.000000

* PF(non-configurable)/PF(configurable)
** L is the expected vector length
*** L is considered infinitely large

References

[1] T.C.Chen,"Parallelism,Pipelining and Computer Efficiency", Computer Design (Jan. 1971).

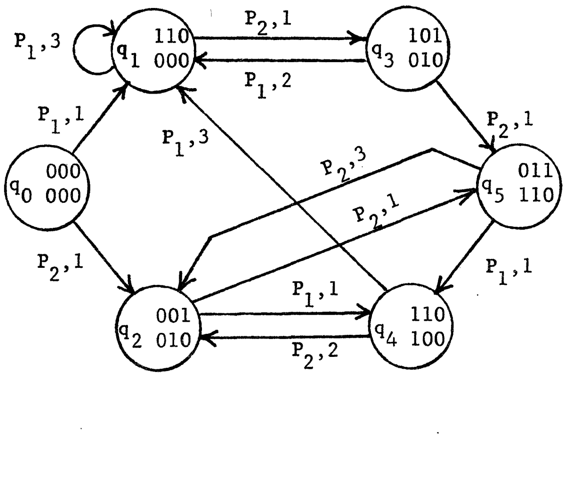
[2] C.V.Ramamoorthy,H.F.Li, "Pipeline Architecture", ACM Computing Survey (Mar. 1977).

[3] D.P.Bovet,M.Vanneschi,"Models and Evaluation of Pipeline Systems",Computer Architecture & Networks,North Holland Publ.Co. (1974).

[4] E.S.Davidson,"Scheduling for Pipeline Processors",Hawii Intn.Conf.System Sciences(1974)

[5] E.S.Davidson et al,"Effective Control for Pipeline Computers",Proc.COMPCON(Spring 1975)

[6] J.H.Mirza,"Generalized Method for Analysis of Pipeline Architectures",Polytechnic Institute of New York, POLY-CS-80-001 (Sept. 1980).



A CRAY-1 SIMULATION USING PASCAL-PLUS

R.H. Perrott and C. King
Department of Computer Science
The Queen's University of Belfast
BT7 1NN N. Ireland

Summary

This paper reports on an experiment in parallel program construction, namely, the simulation of the computation section of the Cray-1 computer [1] using the language Pascal-Plus [2]. Pascal-Plus is an extended version of Pascal which provides the user with parallel features.

As a result of this project we now have a working model of the Cray-1. The model accepts programs written in CAL, the Cray-1 assembly language, and produces a summary of the usage of the functional units, the memory accesses, the amount of scalar and vector computation, the run time of the program, the MFLOP and MIP rates. In addition the user can request an instruction by instruction trace of a program.

Previous simulations of the Cray-1, such as that at the University of Michigan [4], have been constructed using a sequential programming language like Fortran. We found that the parallel features, program modularisation and data abstraction facilities of Pascal-Plus were well suited for the simulation of the concurrent activities of the Cray-1.

The language used for the simulation was Pascal-Plus, full details of which may be found in [5]. However the salient features which were used in the construction of our model are described below.

Pascal-Plus is an extended version of Pascal which was specifically designed to support parallel processes and to enable discrete event simulation. The language extensions are the envelope structure which is an aid to program modularisation and data abstraction, the process, monitor and condition structures which provide a means of representing parallel processes and controlling their subsequent interaction, and, a simulation monitor, which provides pseudo-time control facilities for parallel programs.

The envelope is used to define a data structure and all the operations that can be performed on that data structure; the operations are represented by means of procedures or functions. In addition, there is a control structure which brackets or envelopes the execution of any block which creates an instance of the data structure. In this way the user can ensure that certain actions can be performed before and after the execution of the block in which the instance of the data structure is declared.

In our model the envelope was used to represent the collection and the output of statistics for each program executed by the simulator. In this way the design, development and construction of the statistic collection module could be isolated from the development of the rest of the model.

The overall structure of this envelope was as follows:-

```
envelope statistics ;
   declaration of local data procedures and
      functions ;
begin (* body of the envelope *)
   initialisation of the data ;
   *** ; (* inner statement *)
   finalisation
end ; (* statistics *)
```

Instances of this envelope, as many as the programmer requires, can be declared as follows:- instance timing : statistics ; The block in which 'timing' is declared is then executed in place of the inner statement which is represented by '***' in the body of the envelope.

Only the procedures and functions of the envelope which are prefixed by an asterisk '*' i.e., starred can be called by the statements comprising the body of the block in which 'timing' is declared; starred data identifiers can also be accessed but in read only mode. All other identifiers and procedures are therefore protected.

The envelope was found to be a useful abstraction mechanism in this simulation experiment. The block which declares and uses the facilities of the 'statistics' envelope requires no knowledge of its representation or its initialisation or finalisation phases. Hence it could be constructed separately and even modified at a later stage provided none of the starred identifiers were changed.

Processes are used to identify any independent actions which may take place in parallel, for example, the execution of the functional units. A process can be defined and then instances of it declared, similar to the way in which envelopes are defined. The inner statement of the block in which an instance of the process is declared represents the execution of the body of the process. Once activated the processes proceed conceptually in parallel until they terminate whereupon the finalisation statement (if any) is executed.

A monitor [6] consists of the data which several processes wish to share and the procedures

CH1569-3/80/0000-0105$00.75 © 1980 IEEE

which can manipulate this data; the data can only be accessed and updated by a single process at a time. Thus a monitor provides a means of controlling communication and interaction among the processes by guaranteeing exclusive access to the data.

If a process enters a monitor to update a shared variable it may have to be suspended pending the action of another process; this is achieved by means of condition queues. Hence within a monitor for each condition that must hold before a process can continue, a queue is required. Processes wait on a queue until signalled by another process to continue.

The user can declare these queues as follows instance unitqueue : condition ; To suspend itself on a condition queue a process performs a wait operation as unitqueue.wait. To release a process from a queue another process performs a signal operation, indicating to the signalled process that the reason it was delayed no longer holds as unitqueue.signal. Thus processes and monitors are the basic structuring tools for programs involving parallelism.

Each functional unit of the Cray-1 was represented as a process. All the functional units have a similar structure in that they oscillate between periods of activity and inactivity. When they are inactive they wait on a condition queue until requested by an instruction to perform their function. Because of the similarity in their structure an array of condition queues and processes was declared.

The structure of a functional unit process is such that after creation it will wait on a condition queue, when it is signalled by another process it enters an infinite loop. The loop consists of periods of activity and then waiting on its condition queue again.

One instance of this process for each of the functional units is declared; a parameter is used to distinguish between them. Each functional unit process is initiated whenever the inner statement of the block in which it is declared is encountered. The order of creation is the same as the order of declaration of the processes.

A simulation monitor is included in Pascal-Plus in order to provide facilities which help with discrete event simulation. The main feature is an ordered queue known as the time queue on which processes suspend themselves for a period of pseudo-time using a procedure 'hold'. The queue is organised so that processes with early wake up times are at the front of the queue; the wake up time for a process is the sum of the current time plus the parameter of the 'hold' procedure.

Thus for a functional unit process the period of activity is represented as a call to this monitor procedure. For example, simulation. hold (holdunittime) where the parameter 'holdunittime' represents the period of time for which this particular functional unit is meant to be executing.

Simulation time only advances when all processes are suspended either on a condition queue or on the time queue. Only then is time advanced to the wake up time of the first process on the time queue. All processes waiting for this value are then reactivated.

The simulation terminates when all the processes in the model are waiting on condition queues.

This section describes the structure of the model and some of the problems we encountered during the design phase.

Our original plan was to model the computation section of the Cray-1 as a series of dynamic and static resources; the former being the functional units and the latter being the memory and the various registers. In this way each dynamic resource could be represented by a process which would lie dormant on a queue until it is presented with operands and asked to perform its function.

The static resources were to be assigned to various monitors in which they would be protected from the unpredictable effects of parallel processes. Whenever a functional unit required a particular register it would make a request to the appropriate monitor. If the request could not be satisfied the process or functional unit would have to wait on a condition queue until it became available.

However two situations complicated this design decision and caused the model to be restructured:- a functional unit could become free before the registers that it was using; our scheme implied that the acquiring and releasing were performed by the functional unit process, and, the technique of chaining, where the result operand of one functional unit is fed to another, caused a similar type of problem about the releasing of registers.

To surmount these difficulties the classification of dynamic and static resources was changed. The registers were described by processes so that they could be released before the functional unit which was using them.

The structure of these register processes was similar to that of the functional units, and defined and declared accordingly.

The memory was also regarded as a functional unit and treated as a process for timing considerations. Our model did not take account of the complex timing mechanism relating to the issue of instructions whenever bank conflicts occur.

We found the program and data structures of Pascal-Plus well suited for the representation and manipulation of parallel events. The major benefit of Pascal-Plus in comparison to a sequential programming language is the ease with which concurrent operations can be specified and synchronised.

SESSION 5: RESOURCE CONTROL AND ALLOCATION

HARDWIRED RESOURCE ALLOCATORS FOR RECONFIGURABLE ARCHITECTURES

Bharat Deep Rathi
Anand R. Tripathi
G. Jack Lipovski
Department of Electrical Engineering
University of Texas at Austin
Austin, Texas 78712

Abstract

This paper describes hardwired resource allocators for TRAC-like reconfigurable architectures. These allocators facilitate searching for available resources in the system and allocation of a subset of these to a given request. Various algorithms can be implemented for the search and the allocation of the resources. Tree-structured allocators look particularly attractive with the cost-delay product being of the order of $M^*(\log M)^2$ for a system with M resources of the same type. The paper also describes how this scheme can be extended to allocate multiple type of resources in the system.

1.0 Introduction

Conversion of software functions into hardwired modules looks attractive because of the promise of improved execution speed. Due to the recent advances in semiconductor technology the trade-offs involved in cost-speed functions have favored increased speed at a small increase in hardware cost. This trend in decreasing hardware costs has encouraged system designers to incorporate many of the software functions, specially those related to the operating systems, into hardware modules [8], [13]. For example, some architectures provided hardwired functions for manipulating capabilities [3]. The Symbol-2R [8] architecture had a hardwired supervisor to support a time-shared environment, and had features for direct execution of high-level languages. In some of the IBM 360 series machines table-look-up and address translation functions for paging systems were made faster using associative memories. CASSM [11] and similar kind of architectures [6] proved that many of the conventional software functions in data-base applications could be easily transplanted into hardware to enhance the overall system performance. PASM [9] architecture uses separate microcontrollers to control partitioned processor arrays in SIMD mode. These controllers can selectively mask some of the processing elements (PEs) in the array by using a mask vector which specifies the addresses of the PEs to be masked. More than one PE can be specified by using don't-care values in the address tuples. The scheme provides an intelligent mechanism to specify and decode the addresses once it is known which PEs are to be selected for execution.

This research is supported by NSF Grant MCS77-15968.

In this paper we show that the resource allocation for architectures such as TRAC [10] can be hardwired. The scheme presented in this paper is useful in deciding which of the available resources be allocated to a given request for partition. Such decisions are dependent on the algorithm implemented in the hardwired resource allocator. In fact, the kind of resource allocator we present here can be used in any architecture having large number of identical modules as assignable units. If the system has more than one type of assignable modules (e.g. in addition to memory modules it may have disks, tape-drives, or printers), then this approach can be easily extended to cover such cases. But it is especially useful in allocating resources that need to be set up quickly, such as components of a switch or of shared memory. The structure presented here is a step towards decentralization of control. The hardware cost of this kind of resource allocator is of the order of $M^*\log(M)$ when there are M assignable modules in the system. Tree-structured allocators are particularly attractive because delay is of the order of log M.

We have used TRAC as a model to present our thesis that hardwired schedulers can be effectively used to implement a range of algorithms for resource allocation on reconfigurable machines. Performance (effectiveness) of scheduling algorithms for architectures based on networks, such as banyans, can be highly dependent on the mix of the jobs to be scheduled on the system. This paper neither proposes nor claims effectiveness of any scheduling algorithm for reconfigurable architectures like TRAC. This kind of study of scheduling algorithms for banyan network based architectures is being done elsewhere [2].

One of the basic philosophies in scheduling large, modular multi-processor architectures is that the software scheduler should maintain only minimal amount of information on the global system-state. The scheduler should transmit parameters to the hardware which allocates the resources. The hardware should allocate resources to avoid blockage, faults, etc., and respond with a success or failure signal to the scheduler. If done with care, this philosophy can permit distributed control of the switch which enables parts to be controlled independently, removes the centralized controller and most importantly removes communication paths (pins) between the central controller and the switch. Using the resource allocators presented here, we find that there is little need to maintain even the list of available resources in the system. However, maintaining such information can be very useful so the scheduler will not attempt to allocate resources when there are not enough currently available in the system.

In this paper we present two strategies to search for available resources in the system, and two algorithms to select a subset of available resources. The selection algorithms when used for TRAC like architecture require additional hardware. In contrast, the search strategies can be implemented using the logic of the banyan switch as used in TRAC. The constitution of the rest of the paper is as follows: the next section presents the problem description; section 3 describes algorithms for search and selection; section 4 describes algorithms for search and selection of resources; section 5 presents the functional design of hardware structures and the required control logic; and finally section 6 presents some ideas on multi-type resource allocators.

2.0 Problem Description

In the architectures which we will be primarily concerned with in this paper a set of resources - processors, memories, I/O devices - is connected by a switching network which is used to partition these resources into independent processing structures. Goke [4] showed that banyan networks are suitable for this purpose, and an architecture based on banyan networks was proposed in [5]. We will be using this architecture to demonstrate our ideas on hardwired resource allocation. In the following paragraph the logic for setting up such partitions is briefly described. We show that if a certain partition on the switch is requested, it is either granted and an acknowledge signal is returned to the scheduler, or a failure is signalled in case of a blockage.

In [5] the partitions consist of data-trees and instruction trees. A data-tree connects a set of memory modules to a processor called the root of that data-tree. Instruction-tree connects a set of processors which are roots of data-trees to facilitate SIMD mode of operation. In [4] the basic logic of setting up such partitions is described, a more detailed design of such a switch, as used in TRAC, can be found in [7]. In TRAC a four-level banyan switch with spread=2 and fanout=3 is used; the switch has 81 base and 16 apex nodes. The memory-modules and the I/O devices are connected to the base nodes, and the processors are connected to the apex nodes. One of the base nodes can be used as a port into which the software scheduler feeds commands and addresses into the hardware resource allocator.

In order to set-up some partition on the switch, the scheduler has to first decide which of the available memory modules are to be chosen as candidates. A bus connects the port (feeding commands to the switch) to all the resources, so they can be addressed, like I/O devices in a microcomputer. The selection procedure then selects and marks a subset of these memory modules sequentially using the control bus. Marking is done by addressing the resources, as I/O devices are addressed in microcomputer, and setting a flip-flop. To connect a set of memory modules (as data-tree leaves) to a processor, the marked modules send a request signal up towards the processor nodes. This request signal, at every node in the switch, proceeds upward to all links immediately above it, through all the levels to the apex processors. If the request signal finds a busy or faulty node at any level, then from that node upwards a denial signal is sent. The denial signal, like the request signal proceeds upward to all links connected immediately above it, through all levels to the processors. All those processors which do not receive a denial signal are candidates to be the root for the desired tree. There exist unblocked paths from these processors to each of the requested memory modules. One of these processors is selected as the root of the data-tree. Then from this processor a grant signal is sent towards the memory nodes. This grant signal proceeds toward the memory, going out on each link below each node that it gets to. Any link getting both a request and a grant signal is part of the tree. Such a link switches itself to a conducting state to form paths between the processor and the requested memory modules. The set of links, though physically a tree, act like a wire-OR or a tristate bus connecting the resources.

Because of the blocking nature of the switch it is possible that none of the processors can be connected to all the requested memory modules. Such a partition is said to be blocked and a negative acknowledge is signalled to the scheduler. Generally, a process requires a set of memories and I/O devices. Some resources have data in them required by the process. These are care resources, but other resources need only be selected from a pool of similar resources, such as empty memory modules. If the request for some or all of the memory modules is of don't-care type (i.e. any set of memory modules can be selected from the available modules), then the request for setting up the data-tree is retried with another set of modules. Testing all don't cares to generate sets of specific resources to form a data tree is an NP-complete problem, and a serious limitation in the scheduling mechanism used in the TRAC. In this paper we study how this selection can be done by hardwired allocator algorithms. These algorithms are defined in the next section. Our interest in the resource allocation problem was mainly instigated because of its existence in the TRAC system.

3.0 Resource Search and Selection Algorithms

Resource allocation has three phases: search for qualified resources, selection and validation of a subset for allocation, and finally granting of resources to the request. It is important here to understand what we mean by the term "qualified resources". The set of qualified resources is a subset of the available resources, and various criteria can be used to designate an available resource as qualified. These criteria determine what strategy must be used to find the qualified resources in the system. The second phase is to select a subset of the qualified resources and validate them if the desired partition can be set-up. In the last phase, on successful validation, these resources are marked unavailable for use in further tree allocation and the partition is set up. The search and selection algorithms might make use of some properties of the interconnection network. In case the selected set of modules cannot be connected, the selection phase retries another subset of the qualified modules.

3.1 Search Srategies for Qualified Resources

In this section we describe two strategies to search for qualified resources in the system.

Strategy 1: The simplest way to define qualified resources is to designate every available resource as qualified. Therefore, the selection algorithm considers all the available resources when selecting a subset for validation. A tag bit can be associated with each module to indicate whether it is available or busy; when the resource is busy this bit is reset, otherwise it is set. Failed modules can be excluded from the set of qualified resources by resetting this bit. Using a tag bit on each module to indicate its availability, there is no need to have a search phase, because all the available modules are tagged, which also signifies that they are the qualified modules.

Strategy 2: This strategy shows how qualified resources can be defined on the basis of some properties of the interconnection network. In the proposed strategy, a signal is sent down from an unused processor towards the memories. The signal propagates downward only if it does not encounter any busy node on its path. All the base nodes which receive the signal are designated qualified, and are connectible to this processor. If the number of qualified modules is greater than or equal to the number requested for the data-tree, then the allocator algorithm allocates a subset of these and requests the data-tree formation.

If the number of qualified modules is less than the desired number, then this procedure is repeated with the next available processor. If all the available processors have been tried, that means that the requested data tree cannot be loaded at the current state of the switch.

The advantage of these strategies is that there is no need to maintain a list of available resources with the scheduler. Whenever needed, the scheduler can generate this information in a few memory-cycles.

3.2 Resource Selection Algorithms

We will be considering two selection algorithms: first-fit and group-fit.

First-fit Algorithm: In this algorithm all the qualified modules are serially assigned a number, the order in which this number is assigned can be a function of the network topology. In banyan networks this numbering can be based on the addressing scheme [12] which implicitly captures the concept of the distance function [4]. For a data-tree request of "rr" modules the scheduler allocates the first "rr" number of qualified modules. The switch controller then attempts to form the data-tree, in case of a failure the first module in the selected set is replaced by the (rr+1)'th qualified module. The data-tree formation is retried with this new set of modules. This process is continued either for a fixed number of attempts, or until the request with the last available module is tried. This can be viewed as a moving selection window of width "rr", which is moved one step from left to right for every attempt until the requested partition can be set-up.

In the design of the resource allocators presented here, we show how this selection method can be used with the search strategies 1 and 2. This selection algorithm, when used for TRAC-like architectures along with the search strategy 2, does not need retries because any partition with the qualified resources is guaranteed to be unblocked.

Group-fit algorithm: In the systems which have resources organized in groups on the basis of physical proximity, it may be desirable to select all the resources for a request from the same group. These groups may possibly be divided into smaller subgroups. For example, in TRAC which has a fan-out of 3, the smallest group size for the base nodes is 3, and it increases as powers of 3, e.g. next larger sizes of groups are 9, 27, and 81. A group of size 9 contains three subgroups of size 3; similarly a group of size 27 contains three subgroups of size 9. Nodes belonging to the same group are "closer" (in terms of the number of links in the smallest sub-tree which can connect them in the banyan network) to one another as compared to any node in a different group.

A group-fit algorithm is encoded in a simulator for TRAC [2]. For a data-tree request it tries those memory modules, from the list of available modules, which belong to the same group. (Note that in TRAC the resource modules to be assigned by the allocator are connected to base nodes of the banyan network.) The number of such modules is f^l, where l is the number of levels in the switch. The algorithm is defined below for a banyan network with spread=s and fanout=f. The algorithm presented below is for the search strategy 2. In this case the setting up of the data-tree is guaranteed if the required number (or more) of resources qualify during the search phase.

```
{rr = number of resources requested}
set m such that $f^m$<= rr < $f^{m+1}$
if rr <= number of available resources then
begin
  for n:= m to 1 do
  begin
      if rr modules are available from
           base node addresses
         in the range $(1+(i-1)*f^n)..i*f^n$
      then allocate rr modules and request
            data-tree formation
 end;
```

If the number of base-nodes is M and the number of processors is P, then the complexity of the above algorithm is O(P*log M) for strategy 2.

4.0 Functional Design of the Resource Allocator

The designs which we propose here support search and selection phases of resource allocation. Validation and granting would normally be supported by the control logic for partitioning and reconfiguration on the interconnection structure. The functional design of these hardware allocators is presented here.

When designing hardware, a tradeoff in cost and speed is always encountered. To reduce the cost of a design, it is desirable to take advantage of low IC replication costs. This requires a design constraint on the number of IC pins. While to gain speed, structures with minimal delay are required. In the hardwired resource allocator designs, this tradeoff is guided by the operating environment.

We propose two types of allocator designs: first is the Tree Structured Allocator, and the second is the Linear Structured Allocator. The first allocator optimizes on speed and provides two types of algorithms. One is the First-Fit algorithm, where the resources are allocated on the first available basis. The second is the Group-Fit algorithm; here we assume that the system network structure divides the resources into groups. This grouping can be done intentionally, or can be a side effect of the interconnection network. The Linear Structured Allocator is aimed at optimizing hardware cost. The delay of the allocator is proportional to the number of qualified resource modules. Only the First-fit algorithm can be supported by this kind of allocator design.

For the discussion of the allocators given below, we assume that the resources are of the same type. Extending the algorithm to handle multiple types of resources is not difficult, and solutions for this are given in a later section of the paper.

4.1 Tree Structured Allocators

4.1.1 Functional Overview of the design To achieve high speeds in a cost effective way, we chose a tree interconnection structure for the resource allocator. This interconnection structure is separate from the interconnection structure used for reconfiguration and multiprocessing. The resources are attached at the leaves, or at the leaf and internal nodes of this tree structure. The tree structure is used to calculate the number of qualified resources to the left of each node/leaf and also the total number of qualified resources in the system. This structure also assigns a serial number to the qualified resources. This is done in time proportional to log(M), where "M" is the total number of the resource modules connected to the tree.

Each node of the tree looks like an adder [1]. If we assume a binary tree, then each node has 3 pairs of directed links incident on it (Fig. 1). Each pair has its left link going down,

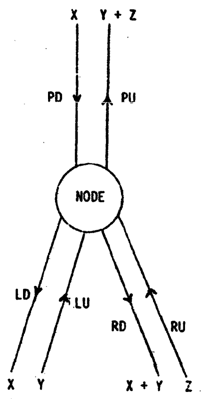


Figure 1 Functional view of the Tree-adder node

while the right link goes up. If a number X is placed on the link PD, and numbers Y and Z are placed on the links LU and RU respectively, the resulting sum of these numbers is shown at the end of links PU, LD and RD in Fig. 1. The PU link of the root node outputs the total number of qualified resources in the system.

To explain the working of our algorithm in hardware, we assume that the resources are placed arbitrarily as the leaf nodes. The algorithm requires each resource to indicate if it is qualified. This is done by appropriately setting a flip-flop QR in each qualified resource. Then QR true indicates the resource is qualified, while QR false indicates it is busy. It is also necessary to store the value "rr", the number of resources requested by the task. A suitable bus will be assumed for this transfer to take place from the controller to the resource modules.

Since only the qualified resources should be considered for allocation, we need to sum only the values of all QRs. Thus the numbers Y and Z (in Fig. 1) are the QR values of the respective resources connected at those locations, while X is set to zero to indicate that there exist no resources to the left of the left most leaf. This is best explained by studying Fig. 2. Here a system of 8 resources

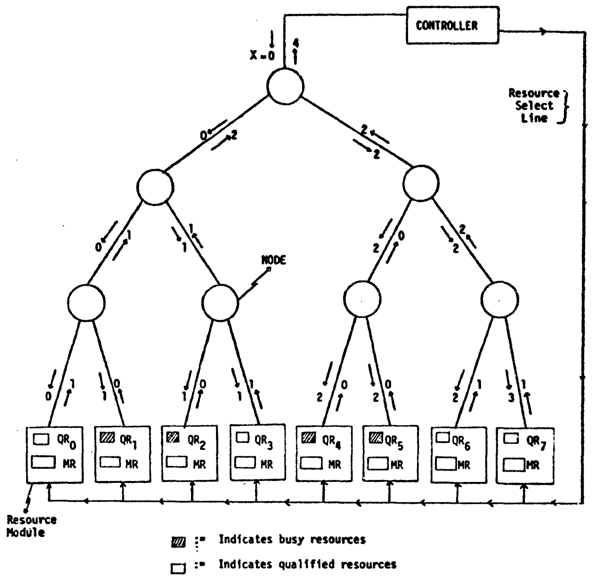


Figure 2 Tree resource allocator example

is connected as explained above. Each directed pair of links of Fig. 1 is shown as a single link between nodes, while the arrows indicate the directed links. The value at their head gives the respective sum at that link.

The total number of qualified resources in Fig. 2 is obtained at the root and is sent to the controller. If "rr" for a task is less than or equal to this total then the controller knows the task can be scheduled. The controller then asserts the resource select line to assign the resources to the processor. Each resource that is qualified and has a sum less than "rr" on its LD (or RD) link, automatically selects itself. The sum on the LD (or RD) link would be the serial number assigned to the resource. After the selection, the next phase of validation might be necessary if search strategy-1 is being used.

To implement the moving window of search strategy-1, to shift the window to the right we decrement the value X being fed into the PD line (figure 1). The controller does this when required, and stops this process on the last qualified module being tried in the window.

The example given above presents one type of architecture for tree structured resource allocators. We give below three different types of tree-structured architectures, two for the First-Fit algorithm, and one for the Group-fit algorithm. These are implemented assuming search strategy-2 is used for selecting the resource modules. The cost of such resource allocators is of the order of M*log(M) when there are M resource modules. The linear term specifies the cost of the tree nodes, while the log(M) term gives the cost of the links in the tree structure.

4.1.2 <u>First-fit Algorithm -- Binary Tree With Resources At The Leaves</u> The resulting tree structure obtained on connecting the resources is shown in Fig. 2. Each leaf node is the resource module itself. A block diagram of the internal nodes (including the root) is given in Fig. 3. The full adders of a node can be

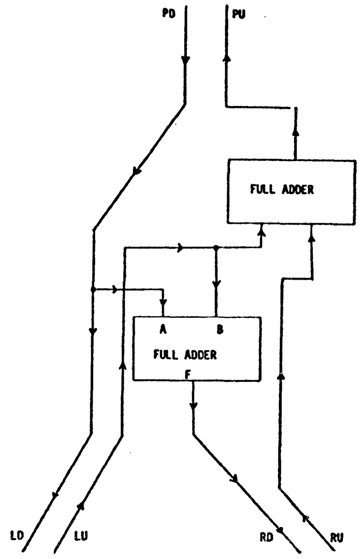


Figure 3 An internal node of a Binary tree allocator with the resources attached to the leaves

SN74283 [14]. A separate bus to pass "rr" from the controller to the resources is not required. Instead we feed "rr" into the root nodes PD link and then set the B port to 0 to get the F = A function. Looking at Fig. 3, we see our tree would be functionally equivalent to a bus.

The width of each link is $w = \lceil \log_2(M) \rceil$ and the number of wires per node required is 6w. This is a large number if the node is to be implemented as an IC module. But we notice that the number of pins can be reduced to 6, if we serialize the data in the links of the tree . This would be at the cost of a reduction in the execution speed of the algorithm. But this cost may be overshadowed by the cost of implementing 2 to 3 nodes on the same IC module.

Assuming a parallel data transmission tree structure, the following execution times are obtained. Here ND is the delay imposed by the node, while "M" is the total number of resources connected to the tree. The time taken to compute the total number of resources is "Ta", while the time taken to serialize is "Ts".

$$Ta = \lceil \log_2 M \rceil * ND$$

$$Ts = (2 * \lceil \log_2 M \rceil - 1) * ND \quad \text{for } M \geqslant 2$$

This architecture can be used when minimal design changes to the existing resource module are wanted, or when the resource modules are placed far apart. Here the consecutive leaf nodes can be the resources that are physically close to each other. This would reduce the length of the link (wire) connecting them to the tree.

Binary Tree With A Resource At Each Node In this implementation we place the tree node in the resource module itself. This way each leaf and internal node, including the root of the resulting tree is a resource module. This approach reduces the number of nodes and levels in the resulting tree. The node design will have to be modified as shown in Fig. 4.

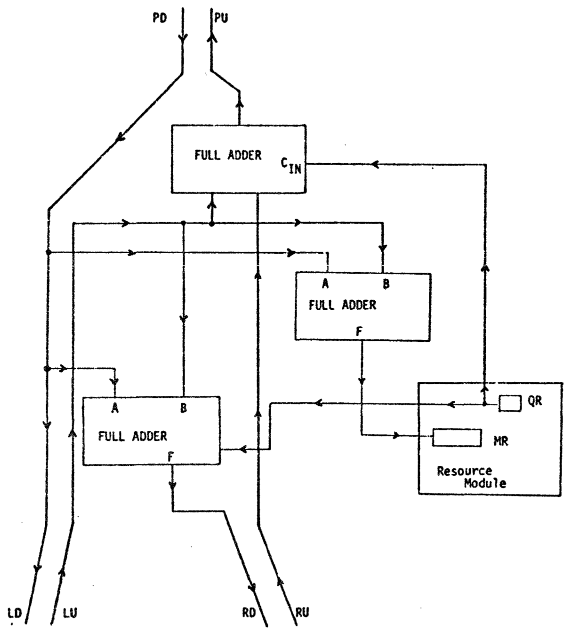


Figure 4 A node of a Binary tree structured allocator with the resources attached to all nodes

Like the above described design we can use SN74283 ICs. This structure has the advantage that no external tree need be implemented. The resource modules would have to be linked to their neighbors in the appropriate manner. Resources connected to the tree as shown in Fig. 5. The node numbers indicate their physical location, with node 1 assumed to be the left most node. The node allocation basically follows an in-order tree traversal path. This arrangement reduces the length of the links (wires) in the tree structure, which is necessary to reduce the cross talk and the cost of the tree.

The cost can be further reduced by simplifying the leaf nodes to contain only the serial register MR, the QR flip-flop and the

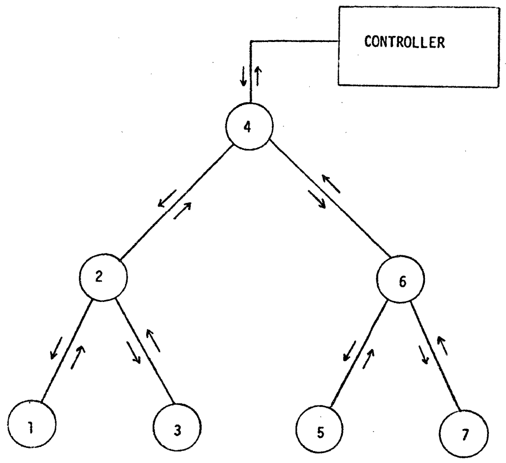


Figure 5 Assigning resources to nodes of the tree structured allocator.

resource control hardware (as in Fig. 2). The number of pins required on each node IC for a totally parallel data transmission tree are $w = \lceil \log_2(M) \rceil * 6$.

The execution time for this tree structure is as follows -

$$Ta = (\lceil \log_2(M + 1) \rceil - 1) * ND$$

$$Ts = (2 \lceil \log_2(M + 1) \rceil - 3) * ND \quad \text{for } M \geqslant 2$$

This architecture can be used, when the resource modules are implemented as ICs, or are placed physically close to each other. The node can then be placed within the IC, and using serial data transmission the pins can be reduced. Even if an IC implementation is not required, the node design given above is cheaper in terms of hardware, if the leaf nodes are simplified as explained above.

4.1.3 Group-fit Algorithm In networks which have their resources organized in groups, it may be advantageous to select the resources from a single group. If we can allocate all the required number of resources from the same group, then we can save on the number of links allocated. This saving is in terms of reducing the possibility of blocking other available resources in the following allocations [4].

An example of this design for the TRAC architecture [10] is presented here. Other applications can use this by modifying the node design as required. We use a ternary tree, with the resources attached to the leaves. This allows for a simple node design as shown in Fig. 6. A request for partition is sent to the root of the tree, from which it moves towards the leaf nodes as described below. A node selects a particular son if that subtree has the required number (or

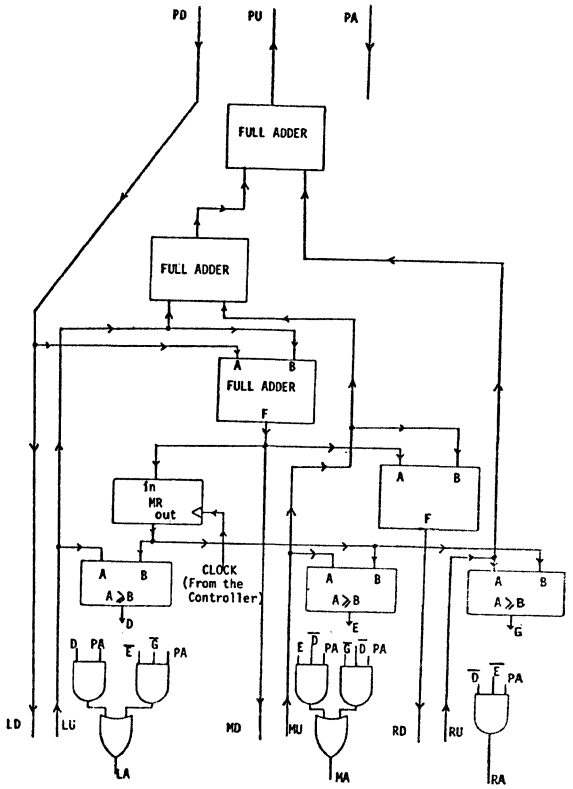


Figure 6 A node of the group selection tree structured allocator

more) of qualified resources, and none of its elder brothers (if any) have the required number of qualified resources in their subtrees. If none of the sons satisfy the above condition, then the first "rr" resources in that subtree are selected. To correctly identify the active subtree an extra line is required in the node. This is set if the parent of a node has selected this branch of the tree (i.e. line PA).

Qualified resources transmit their QR values. The resulting sum at the root and the serial numbers are calculated. At the root cell a comparator checks the sum, the PA line is asserted if there are equal to or more than "rr" qualified resources. A similar check is done at all nodes and the respective control lines are asserted (see Fig. 6). If a node lies in an inactive subtree then the LU, MU and RU links are pulled down to 0. This can be done by designing them to be wire-OR lines. At any node in the tree, the respective lines are then made 0 if their compare function is 0. This extra hardware (not shown in the Fig. 6) is needed to correctly serialize the resources that are in the active (selected) subtree. The resources allocated are those which are qualified, have their PA set, and have serial numbers less than "rr".

The tree will have the following worst case timing :--

$$Ta = \lceil \log_3 M \rceil * ND$$

$$Ts = \begin{cases} ND & \text{for } 1 \leqslant M \leqslant 3 \\ (3 \lceil \log_3 M \rceil - 1) * ND & \text{for } M > 3 \end{cases}$$

4.2 Linear Structured Allocator

First-fit Algorithm This architecture does not require the tree structure, instead it uses a counter in each resource module (see Fig. 7). Each resource module requires a

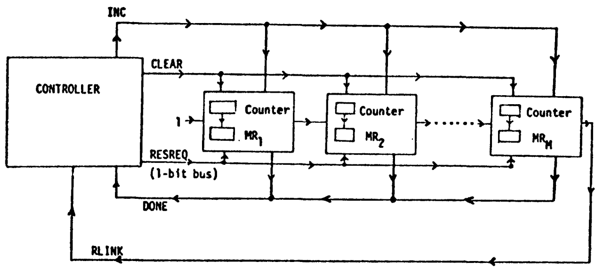


Figure 7 Linear structured allocator with a counter in each resource module

register (MR) to hold the value "rr". A single line bus (RESREQ) from the controller is used to serially transmit this value to all resource modules. Another line INC, to increment the counter is also required from the controller to all modules. Each module constantly compares the counter output with "rr", and sets the DONE line as soon as an equality match is obtained.

The resource modules are qualified to participate in the present allocation run, as done in the other designs. The QR flip-flop is appropriately set, all counters cleared, and the MR register initialized. The controller then sends pulses on the INC line, and all enabled counters are incremented with each pulse. A counter is enabled if QR is set and the resource has not set its right neighbor link (see Fig. 7). The counter is enabled as long as the left neighbor link is not set. On this being set the module increments the counter for the last time, and sets its right link. After this for all following pulses the counter is not incremented. If an intermediate module is not qualified, then its left link is internally joined to the right link, and the counter cleared and disabled.

The incrementing process is stopped as soon as the DONE line is set. It is set by a qualified module, when its counter reaches the value "rr". All resource modules with counter values less than or equal to "rr" select themselves. The counter value for each selected resource is its serial number. If the number of qualified resources are less than "rr" then the RLINK line is set before the DONE line. At this point the controller would select the next processor (if any), and do the above.

This architecture has the advantage of having a low cost, since only few lines and minimal logic is required. It furthers lends itself to be implemented within the resource module itself. A significant loss in speed will be felt only when the "rr" for the allocation is large. This is because the allocation is sequential and has the time complexity of O(rr).

5.0 Ideas on Multi-type Resource Allocators

The above described allocator algorithms considered resources of the same type. We can extend them to handle multi-type resources, as described here. The actual solution chosen, would depend on the cost-speed tradeoffs of the system.

5.1 Tree Structured Allocators

There are two types of multi-type resource allocators - Multi-pass and the Single-pass. In the Multi-pass allocators all the resources of the system are connected to the hardwired allocator as described in the single resource type case. Only the resources of the same type are qualified during a pass. If a task required 3 types of resources to be allocated, then 3 passes would be required.

The Single-pass type allocator allows us to maintain the time complexity of the algorithms described earlier. Here "M" would mean the cardinality of the largest resource type. On the other hand it increases the cost of the hardware, since the width of each link is increased.

Each link for the above defined algorithms had a width of $w = \lceil \log_2(M) \rceil$. To handle multi-type resources in a single pass we have to increase the width to

$$w = w_1 + w_2 + \ldots\ldots + w_i + \ldots\ldots w_n$$

Where "w_i" is the width for resource type "i", and we have "n" different resource types. Each link is logically divided into "n" fields (Fig. 8), but would be considered to be a positive integer number by the

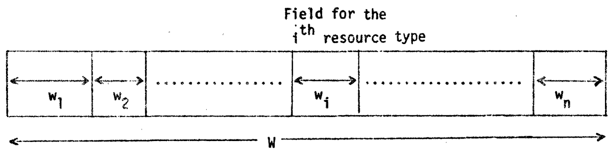


Figure 8 Width of a link in a tree structured allocator

tree adder. If a resource module is qualified it increments only its logical field value. No overflows between adjacent fields take place, since the field width for each resource type is $w = \lceil \log_2(M) \rceil$. The serialization and the summation process of the tree adder would not be changed. But the QR flip-flop connection to the tree and the allocation procedure of the algorithm is slightly altered.

The QR flip-flop output for a resource type is attached to the least significant bit of the respective logical field. While the MR register in each resource module is divided into the above logical fields. For a resource module only its field is made active and the other fields made inactive. This is because during the comparison for allocating a resource, this active field in the MR register is used. It would compare itself with the resulting serialized number obtained at the resource node. The resource would be allocated as done before. The controller allows allocation only if all resource requests are satisfied.

5.2 Linear Structured Allocators

The Multi-pass and Single-pass algorithms can be implemented for this allocator algorithm too. In the Single pass allocator all the resources would be connected together, and only one type of resource would be qualified per pass. While for the Multi-pass allocator we would have a separate controller for each resource type. Resources of the same type are connected to their controller as described earlier. The algorithm would execute as before, the only difference being that the system controller is interfaced to the controllers of each resource type.

6.0 Conclusions

In this paper we have shown how resource allocation functions for reconfigurable, multiprocessing architectures can be delegated to hardwired structures. Such hardwired resource allocators look very attractive for large systems because it relieves the scheduler from the burden of maintaining lists of available resources. Even the resource selection function, which would normally be done serially by the scheduler can be done in parallel using the tree-structured allocators presented here. This would reduce the overhead of serial communication from a central scheduler to the resource modules during the selection phase. The tree-structured schedulers look particularly attractive because of the cost and delay both being of the order of log(M), where the system has "M" number of assignable units. Secondly, the tree-structured schedulers have an inherent capability to capture some of the topological properties of certain interconnection structures, such as banyans, and thereby providing a convenient mechanism to implement a set of intelligent resource allocation algorithms.

Acknowledgements

It is our pleasure to thank Dr. Miroslaw Malek and the refrees, for carefully reading drafts of this paper and for their useful comments and suggestions.

References

1. J.A. Bush, G.J. Lipovski, S.Y.W. Su, J.K. Watson, S.J. Ackerman, "Some Implementations of Segment Sequential functions", Proc. of the Third Annual Symposium on Computer Architecture, 1976, pp. 1-8.

2. D. DeGroot, A.R. Tripathi, D.P.S. Charlu, M. Malek, J.C. Browne, "Report on Simulation and Scheduling of the TRAC

Architecture", TRAC Report 8,1979, Dept. of Electrical Engineering, University of Texas at Austin.

3. D.M. England, "Capability concept, mechanisms and structure in system 250", Symposium on Protection in Operating Systems, IRIA, Recquencourt 78150 Le Chesnay, France, Aug. 1974, pp. 68-82.

4. R. Goke, G.J. Lipovski, "Banyan Networks for Partitioning Multiprocessor Systems", First Annual Symposium on Computer Architecture, 1973, pp.21-30.

5. G.J. Lipovski, A.R. Tripathi, "A Reconfigurable Varistructure Array Processor", First International Conference on Parallel Processing, 1976, pp. 165-174.

6. E.A. Ozkarahan, S.A. Schuster, K.C. Smith, "RAP - An Associative Processor for Data Base Management", AFIPS Conference Proceedings, Vol.44, 1975, pp. 379-387.

7. U.V. Premkumar, R. Kapur, M. Malek, G.J. Lipovski, P. Horne, "Design and Implementation of the Banyan Interconnection Network in TRAC", AFIPS Conference Proceedings, Vol.49, 1980, pp.643-653.

8. H. Richards, A.E. Oeldhoeft, "Hardware-Software Interactions on Symbol-2R's Operating System", Proc. of the Second Annual Symposium on Computer Architecture, 1975, pp.113-118.

9. H.J. Siegel, P.T. Mueller, H.E. Smalley, "Control of a Partitionable Multimicroprocessor System", Proc. of the 1978 International Conference on Parallel Processing, pp 9 - 17.

10. M.C. Sejnowski, E.T. Upchurch, R. N. Kapur, D.P.S. Charlu, G.J. Lipovski, "An Overview of the Texas Reconfigurable Array Computer", AFIPS Conference Proeedings, Vol.49, 1980, pp.631-642.

11. S.Y.W. Su, G.J. Lipovski, "CASSM: A Celluar System for large data bases", International Conference on Very Large Data Bases, 1975, pp. 456-472.

12. A.R. Tripathi, G.J. Lipovski, "Packet Switching in Banyan Networks", Proc. of the Sixth Annual Symposium on Computer Architecture, 1978, pp. 160-167.

13. T.A. Welch, "An Investigation of Descriptor Oriented Architectures", Proc. of the Third Annual Symposium on Computer Architecture, 1976, pp.141-146.

14. Texas Instruments Inc., "The TTL Data Book for Design Engineers", Second Edition, 1976.

RESOURCE CONTROL IN A DEMAND-DRIVEN DATA-FLOW MODEL

Bharadwaj Jayaraman
Robert M. Keller

Department of Computer Science
University of Utah
Salt Lake City, Utah 84112.

Abstract -- An approach to the synchronization and scheduling of resources in a demand-driven data-flow model is outlined. It is shown that demand evaluation provides a natural model for resource usage and yields elegant solutions to certain problems, such as the avoidance of busy waiting and resource scheduling. A graph-based applicative language (FGL), with data-flow semantics, is first introduced for explaining our primitives for resource control. A textual version of FGL is later used for presenting examples. The benefits of an applicative language in aiding well-structured design, and the clarity of data-flow models in making indeterminate behavior explicit, are also illustrated.

INTRODUCTION

Data-flow models are well-known representations for achieving asynchronous and concurrent execution of applicative programs ([11], [4]). The term 'data-flow' has, in the past, been synonymous with 'data-driven', since the execution of any operator in a data-flow program is initiated by the availability of its input data. Recently, a demand-driven execution model for a data-flow program has been proposed as the computational basis for an Applicative Multiprocessing System [14]. Demand-driven execution is based on the principle that the execution of any operator is initiated by a demand for its result, rather than by the availability of data. In comparison with data-driven models, the main advantages of a demand-driven model are in avoiding unnecessary computations and allowing conceptually infinite data-structures to be constructed, without requiring them to be manifest all at once.

The work presented here is motivated by the need to introduce the concept of a resource, and techniques for their control, in a demand-driven data-flow model. The emphasis is on a reliable and modular approach to designing many of the synchronization and scheduling functions of an operating system.

This research is supported by the National Science Foundation under Grant MCS-77-09369 A01.

A demand-driven model of execution has led us to develop an extension for resource usage that is also "demand-driven." It is worth noting that a model for resource usage based explicitly on demands is not unrealistic since user programs may be viewed as making demands on the underlying operating system to allocate/access resources.

Some aspects of resources that are ostensibly alien to a demand-driven data-flow model are the following:

1. In purely applicative models, there has heretofore been no notion equivalent to a reference to a data object, since all data objects are values and all computations are value-oriented. As a consequence, there is no notion of updating a data object. Instead, "modified" data objects are essentially new data values. A resource, on the other hand, requires some notion of a reference to it in order to be shared in a concurrent environment, and is updated for sake of efficient storage utilization.

2. Pure data-flow programs are determinate, since their output is determined solely from their input data values, regardless of the timing of operations. On the other hand, due to the unpredictability in the timing of operations and the updatable nature of resources, the behavior of a resource could be indeterminate. Since any legal interconnection of pure data-flow programs can only result in determinate programs [12], it is necessary to explicitly introduce operators for expressing indeterminate behavior.

The main advantages of a demand-driven model for resource control are the following:

1. Since waiting is fundamental to demand evaluation, the avoidance of busy waiting is accomplished without a need for explicit protocols for "putting to sleep" and "waking up" a task. In fact, no additional mechanism is necessary for creating and destroying tasks.

CH1569-3/80/0000-0118$00.75 © 1980 IEEE

It is possible to eliminate the use of "bracketing" operations around a critical operation, e.g. the operations **startread** and **endread** around a **read** operation, as in Monitors [9]. These bracketing operations are effectively substituted by the demand on a resource to perform an operation and the return of a result by the resource.

2. Unlike conventional models, where indeterminacy is caused by concurrent operations updating some shared global data, in a demand-driven data-flow model, indeterminacy is a local effect that must be explicitly introduced, and is associated with the time-dependent arrival of demands or data at some operator. This has the advantage of being able to easily identify indeterminate behavior as occurring at well-defined points of the program.

3. Since any indeterminacy must be explicitly introduced, the arbitration and scheduling of operations becomes explicit and also user-programmable.

AN APPROACH TO RESOURCE CONTROL

We summarize the salient features of our approach to resource control. A logical resource consists of two main components:

- the actual resource and associated operations on it.

- the specification of its synchronization and scheduling.

In the interest of modularity, these components can be independently defined. At this stage, we wish to treat the actual resource as an abstract object whose structure and representation are not of critical interest. Hence, we will omit detailed definitions of the access operations. The issues of access rights, i.e. the protection of resources from unauthorized or improper access, have not been considered in this presentation. However, owing to the modularity of our approach, such constraints can be specified separately. The focus for the rest of this presentation will be on synchronization and scheduling.

Our solution to the problem of specifying synchronization is to indeterminately order concurrent accesses by using queue primitives. It is perhaps worth noting that almost all synchronization schemes proposed in the literature rely on some mechanism for queueing. We feel the concept is basic to synchronization, and hence have introduced it explicitly as a primitive. Thus, multiple queues may be defined, and different types of accesses can be allocated different queues, the allocation policy being under the control of the programmer. The availability of multiple queues, each independently accessible, also overcomes the problem of a single input queue bottleneck.

Scheduling consists of selecting some order for serving these queues. Hence, primitive operators are provided for waiting and removing requests from queues. In a model where demands are the only means of initiating the evaluation of any operation, the evaluation of the actual operation to be performed on the resource may be viewed as being intiated by two demands: the first is the user's demand to access the resource; the second is the demand from the resource scheduler to start evaluation. Thus, primitive operators for evaluation control are also provided. When used in conjunction with the queuing operators, these operators allow a programmer to tailor the scheduling of evaluation of concurrent accesses according to many desired specifications.

FUNCTION GRAPH LANGUAGE

The data-flow language described here is a graphical variant of a pure applicative language, and is called Function Graph Language (FGL) ([14], [12]). An FGL program is a "graph grammar" in which each production rule associates a programmer-defined node (the antecedent of the production) with a directed-graph (the consequent of the production). A well-formed graph is any arbitrary interconnection of nodes and arcs, including cycles, with the following properties:

1. The graph, and each node in it, has a (possibly empty) set of arcs directed into it, called input arcs, and exactly one arc directed out of it, called the output arc.

2. Every arc in the graph (except for its input and output arcs) is directed between two nodes. An output arc may fan out into two or more arcs, but merging of arcs into a single arc is not allowed.

3. Every node has a name which may be either pre-defined or programmer-defined, i.e., the antecedent of a graph production.

The nodes in the graph represent operators, and pre-defined names correspond to primitives of the underlying machine. An operator is a pure function whose output is determined solely by its inputs, and does not have any side-effects. Arcs represent data paths between operators, the actual data values being either atomic, e.g. **integer**, **string**, etc., or tuples of arbitrary function graphs. Atomic data values are created by 0-ary operators and represent constant functions. Tuple data values are created by the operator **cons**, and can be used to construct conceptually infinite data structures, as will be explained.

Figure 1 illustrates a typical graph production in this language. Our convention is to use ellipses for primitive operators, and rectangles for programmer-defined operators. The primitive operators **car** and **cdr** select the first and last components of a tuple respectively. The programmer-defined operator **apply-to-all** constructs its output recursively by applying the function **f** in its first argument to every component of the infinite sequence **x** in its second argument. The sequence x can be constructed using nested pairs, e.g. cons(x_1, cons(x_2, cons(...))), where x_1, x_2... are the components of x. However, owing to demand-driven evaluation, only those components of the output that are demanded are in fact constructed.

Figure 2 presents snapshots during the evaluation of the operator apply-to-all when the first component of the output is demanded by a car. We indicate the presence of a demand using an asterisk and, for sake of brevity, we denote a sequence created by cons using angle brackets.

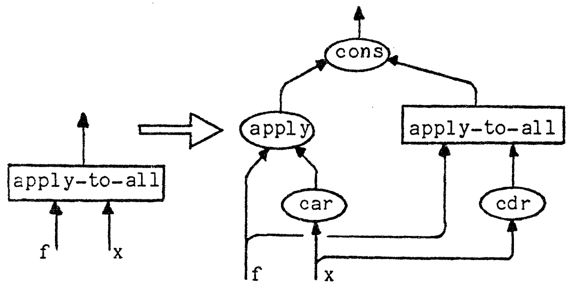


Figure 1: FGL program for stream processing

Demand-driven evaluation

The execution of any operator is initiated by a demand on its output arc, and its completion causes the computed result to be returned to the source of demand. A demand on the output arc of the graph initiates all evaluation. Demands propagate along arcs in the graph when arguments to operators are evaluated. Propagation of a demand along some arc terminates when it reaches a 0-ary operator, e.g. an **integer**, or the tuple creating operator **cons**. A data value (a reference to the tuple in the case of **cons**) is then returned to the demanding node, which in turn propagates its computed value back. The evaluation of the graph is complete when the result at the output node is finally computed.

An important feature of the evaluator is its ability to exploit asynchronous and concurrent evaluation of all independent operators. Two operators are independent if the result computed by one is not needed, either directly or indirectly, as an input of the other. For example, in figure 1, the operators **car** and **cdr** are independent; however, apply and car are not. Independent operators are the only source of all concurrency in this model, which occurs due to operators having multiple input arguments. Asynchronous evaluation allows independent operators to execute at their own speed without any centralized timing constraint. Thus, the propagation of demands, computation of values, and propagation of values can all proceed concurrently in different parts of the graph.

Types of operators

Primitive operators are evaluated in one of two ways: In the case of strict operators, e.g. **add**, all arguments are demanded concurrently, and the operator is applied only after all arguments have completed their evaluation. Non-strict operators, e.g. **cond**, do not require all their arguments to be evaluated in order to compute their result. Therefore, only some subset of their arguments is evaluated, possibly in some fixed order. The result of evaluation of a primitive operator causes the operator to be transformed into a 0-ary operator whose function is the constant function corresponding to the computed value.

Cons is a notable example of a non-strict operator that does not evaluate any of its arguments. Evaluation is done when selector functions, e.g. **car**, **cdr**, etc., are applied to extract specific components of the tuple. It is this property of **cons** that allows conceptually infinite sequences to be constructed, since the components of **cons** could be function graphs that recursively construct tuples, and are never evaluated until actually needed [5].

When a programmer-defined operator is demanded, the corresponding graph is substituted in place of the operator. A demand is placed on the output arc of the graph, which in turn initiates further evaluation. As a consequence, only those arguments of a programmer-defined operator that are needed to compute its final result are evaluated.

In the case of both primitive and programmer-defined operators, one may assume that after an operator has notified all sources of demand the availability of its result, it is deleted from the graph[a]. Thus, the computation may be visualized as a dynamically expanding and shrinking graph.

Evaluation of shared subgraphs

One of the properties of an FGL graph is that it may have shared subgraphs, since the output arc of some operator in the graph may fan out into two or more arcs. Shared subgraphs correspond to common subexpressions, since the result of their evaluation may be used as the input of more than one operator. Sharing of some external graph implicitly occurs when the input arc of a graph fans out into two or more arcs. When two independent operators share a common subgraph, it is possible for both of them to demand the shared subgraph concurrently. In general, it is possible for the output node of a shared subgraph to be demanded concurrently along all its (fanned) output arcs.

The synchronization needed here is a trivial case

---

[a] In practice, however, storage will be deleted only in units of entire function graphs.

of the synchronization needed when a shared resource is accessed: concurrent or multiple demands on a shared node are treated by propagating only the first demand that arrives at the node, thereby avoiding re-evaluation of the common subexpression; the result of evaluation is then returned to all sources of demand.

Further details of the features of the language and the demand-driven evaluator may be obtained from [14]. A loosely-coupled architecture, with FGL as its machine language, is described therein. Other features of the language and some details of an implementation are also discussed.

OPERATORS FOR RESOURCE CONTROL

A resource is accessed by applying an access operator to a pair of arguments: the first being the resource itself, and the second, a tuple consisting of all arguments needed to perform the actual operation. If the actual operation requires no arguments, then the resource will be the only argument to the access operator. Unlike operators discussed thus far, which are purely applicative, access operators could result in side effects. As a consequence, the behavior of a resource could be history sensitive.

Figure 3a illustrates a typical use of resources[b]. The programmer-defined operator **filesystem** represents an abstract file system for sequential files that is accessed by the set of operators, **openfile**, **readnext**, **endoffile**, **closefile** and **readerror**, which have the usual meanings. Side effects are caused by the operators, **openfile**, **readnext** and **closefile**. For example, **openfile** returns **nil** if it was unable to open the file; otherwise, it returns a reference to the file and, as a side effect, actually opens the file. Consequently, the operator **filesystem** is history sensitive. For example, the result of a **readnext** operation depends on whether an **openfile** was first performed, and on the number of **readnext** operations that preceded it.

Synchronization

As indicated earlier, the synchronization of concurrent accesses is achieved by queues. These queues are created inside the resource, and an access operator is synchronized by enqueueing a request onto the appropriate queue. We illustrate the use of the operator **enq** (defined below) by defining the access operator **openfile** of figure 3a:

[b]The primitive operator **seq** sequences the evaluation of its arguments; the programmer-defined operator **compute** is not described here, but is assumed to perform some computation.

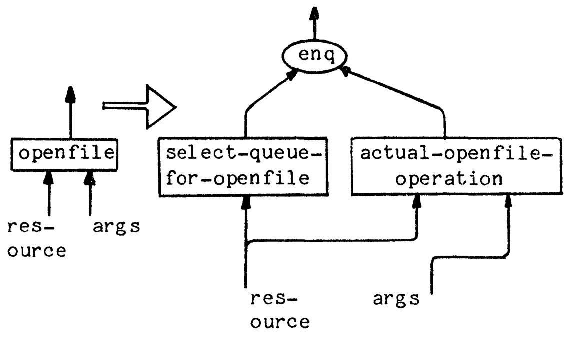


Figure 3b: Synchronization of the operator openfile

When a resource is accessed for the first time, its scheduler gets demanded and in turn proceeds to examine its input queues. The scheduler is essentially a non-terminating iterative program that controls the order of removal and evaluation of requests on its input queues. However, the scheduler may have to wait occasionally, i.e. when there are no requests to be served. Hence, operators for waiting and dequeueing are also provided. We first informally define the primitive operators for queueing, and then illustrate their use with examples.

Operators for queueing

In the following definitions, **q** represents a queue created using a **gqueue**, and **a** is any operator or FGL expression whose evaluation is to be synchronized. In all our examples, **a** will be the actual access operation to be performed on the resource. A reference to a subgraph **a** is obtained by cons(a)[c], and is similar to an unevaluated expression (since **cons** does not evaluate its arguments). The evaluation of **a** is initiated by taking the **car** of such a reference.

gqueue()	creates an empty, updatable FIFO queue.
enq(q, a)	synchronizes the evaluation of **a** using **q**; the result of evaluating **a** is the value of **enq**.
deq(q)	the first request, a, is removed from **q**; a reference to **a** is the value of **deq** (**a** is not yet demanded).
waitq(q)	the demand on **waitq** is satisfied and returns **T** only when **q** becomes non-empty.
nonempty(q)	T if **q** is non-empty, **nil** otherwise.

Mutual exclusion on all queueing operators accessing a particular instance of a queue is

[c]In FGL, **cons** can take an arbitrary number of arguments, including just a single argument.

assumed. However, the evaluation of a takes place outside this exclusion, and is under the control of the dequeueing program. It should be noted that accesses to different queues can occur independent of one another. Furthermore, any waiting that occurs does not block out other queueing operators from accessing the queue.

Avoidance of busy waiting

There are three occurrences of waiting in the above operators:

1. an **enq** operator waiting for **a** to be dequeued and evaluated, before returning its result.

2. a **deq** operator waiting for **a** to be enqueued, before removing **a**.

3. a **waitq** operator waiting for **a** to be enqueued, before returning T.

In order to avoid busy waiting in the above cases, we must first detect two conditions to be true before initiating some action. For example, the evaluation of a requires that a be both enqueued as well as dequeued. In terms of demands, this suggests the need to wait for two demands before evaluating an operator. We therefore introduce a special operator for this purpose:

djoin(cr) evaluates cr only after two demands are received.

Figures 4, 5 and 6 present snapshots of the important transitions that occur when a queue is accessed by the queueing operators. As before, asterisks indicate the presence of a demand. It should be remembered that the operator **cons** does not evaluate its arguments until demanded by selector operators, which in this case are **car** and **cdr**. Also, the queue created by the operator **gqueue** is updatable, and hence is referenced.

Scheduling

Scheduling involves two main tasks: waiting for some subset of queues, based upon some condition, to become non-empty, and selecting one such non-empty queue for dequeueing, followed by evaluation. The types of information that are referred to in these conditions determine the flexibility of scheduling that is achieved. A partial list [2] of these types is the following: the type of access operator, its relative order of arrival, the actual arguments needed for the operation, the state of synchronization, the state of the resource, and the history of accesses on the resource. The operators introduced here, however, are mainly for evaluation control. We first give informal definitions of their behavior, followed by examples of their use.

seq($a_1,\ldots,a_n$)
 evaluates $a_1,\ldots,a_n$ sequentially; returns the result of evaluating a_n.

par($a_1,\ldots,a_n$)
 evaluates $a_1,\ldots,a_n$ concurrently; returns the result of evaluating a_1, as soon as it is ready.

spar($a_1,\ldots,a_n$)
 evaluates $a_1,\ldots,a_n$ concurrently; returns the result of evaluating a_n, after all arguments have been evaluated.

arbit(a_1, a_2)
 evaluates a_1 and a_2 concurrently; returns nil if a_2 evaluates its result before a_1, else T; i.e. **arbit** favors a_1 in case of a tie.

The operators **seq** and **spar** are strict since they require all their arguments to be evaluated before returning their result; however, the operators **par** and **arbit** are not. The operator **arbit** is indeterminate, since its result depends on the relative speed of evaluation of its arguments. Except for the indeterminacy associated with **arbit**, these four operators are similar to all other operators of the base language in that they do not require any extension of the demand evaluation semantics we have described here.

An example: mutual exclusion of two acceses

We illustrate use of the queueing operators and operators for evaluation control by constructing a resource scheduler for a simple problem: mutual exclusion of two types of concurrent accesses. Figure 7 shows a scheduler **mutex** that enforces mutual exclusion in the evaluation of requests on its two input queues **p** and **q**. The programmer-defined operator **mutex** is essentially a non-terminating iterative program, although recursion is used to achieve this[(d)].

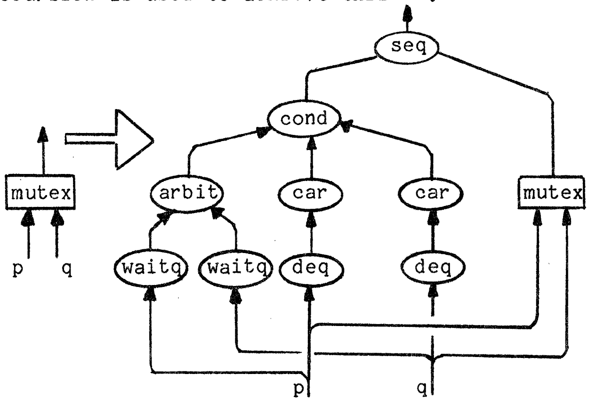


Figure 7: Mutual exclusion of two accesses

The operator **seq** first demands **cond** which in turn causes the **arbit** to be demanded. **Arbit** then demands the **waitq** operators on its two inputs. As soon as an access is enqueued on to one of the

(d)This form of "tail" recursion can easily be detected, and hence the the storage for each recursive invocation may be deleted as soon as it has been completed.

queues, the corresponding **waitq** will return T to **arbit**. In case both queues are non-empty at the same time, there will be a "race" between the two **waitqs** in notifying **arbit** of their results.

Arbit will return **T** if its left input was selected, and **nil** otherwise. Depending on whether **arbit** returned T or **nil**, **cond** will demand its second or third input argument respectively. This causes the selected queue to be dequeued, followed by evaluation of the access operator. Upon completion, the result of evaluation will be notified to **cond**, which in turn returns the result to **seq**. This causes the second argument of **seq** to be demanded, thereby starting another iteration.

Since each iteration evaluates only one access operator, mutual exclusion among the access operators is guaranteed. However, owing to the possibility of a race condition, it is possible for a particular queue to be ignored indefinitely. Hence the above scheduler does not guarantee fairness in serving its input queues. Fairness can be guaranteed by a simple extension to the above scheduler, i.e., by testing the non-emptiness of each queue in strict alternation, using the operator **nonempty** and serving a request on a queue if it is non-empty.

Before concluding this section, we present a rough sketch of how the queues and the scheduler are encapsulated inside a resource, and how access operators are synchronized by them. Figure 8 presents snapshots of some possible sequence of transitions during the operation of a resource that uses the scheduler **mutex**.

EXAMPLES OF RESOURCE CONTROL IN TEXTUAL FGL

The need for a textual representation is motivated by the fact that although graphs are useful during initial program development, and are suitable for representing concurrency, they could lead to awkward program structures. This is because every data dependency has to be explicitly indicated by an arc, thereby complicating the physical layout of such graphs. On the other hand, if one were to name input arcs of a graph and any shared subgraphs within it, these data dependencies can be expressed simply by referring to these names.

The correspondence between a graph and its textual equivalent is quite straightforward, hence we will not discuss the translation in detail. In order to illustrate this correspondence, we define the operator **mutex** in the textual language (see figure 9).

The precise syntax of the language is defined in [15], along with examples of their use. However, we will explain the special features of the language as and when we introduce them. In this regard, the reader may note that the serial composition of unary functions, e.g. **f(g(10))**, may be written without parentheses, i.e., as **f g 10**. We use the keyword, function, when a mathematical function is being defined; otherwise, we use the keyword, procedure.

```
procedure mutex (p, q)
begin seq(if arbit(waitq p, waitq q)
             then car deq p
             else car deq q,
          mutex(p,q))
end
```

Figure 9: The operator **mutex**, in textual FGL

An important feature of the textual language is the ability to name any expression, using the let clause, and to refer to these names. When a graph has no shared subgraphs, e.g. the graph of figure 1, the textual equivalent will require no additional names, apart from those required for the input arcs of the graph. Even in such cases where graphs do not have any shared subgraphs, naming common subexpressions can be a useful abbreviation, besides avoiding unnecessary computation.

A simple version of the Readers and Writers problem

Figure 10 shows a scheduler for a simple version of the Readers and Writers problem [3], in which neither fairness nor any fixed priority is enforced. The only control enforced is the mutual exclusion of readers from writers, and the exclusion of a writer from all other readers and writers. Thus, readers are allowed to execute concurrently with one another.

```
procedure readwrite1(wq, rq, rr)
let removeread be deq rq,
    read be car removeread,
begin if arbit(waitq wq, waitq rq)
        then comment service writer;
             seq(rr,
                 car deq wq,
                 readwrite1(wq, rq, nil))
        else comment service reader;
             seq(removeread,
                 spar(read,
             readwrite1(wq, rq, spar(read, rr))))
end
```

Figure 10: The readers and writers problem: a simple version

The scheduler **readwrite1** has two queues **rq** and **wq** for read and write accesses respectively. The input argument **rr** maintains the set of running readers, as will be described, and is **nil** at the outermost call.

When the queue for writers is selected by **arbit**, a write access is evaluated after ensuring all running readers have completed their evaluation. When the queue for readers is selected, the **seq** (line 10 of the program) first dequeues a read access without evaluating it, then causes **spar** (line 11 of the program) to evaluate the dequeued read access concurrently with the next iteration of **readwrite1**. Thus, as long as the queue for readers is being selected on consecutive iterations, all read accesses will be evaluated concurrently.

The set of running readers, i.e. the set of concurrently executing read accesses, is maintained by the input argument **rr** and is constructed recursively when consecutive read accesses are evaluated. Since the input arguments of a programmer-defined operator, in this case the operator **readwrite1**, are not evaluated until demanded, the input argument **rr** and hence the chain of **spars**, is evaluated only when demanded explicitly (by **seq** in line 6 of the program). This ensures that all running readers have completed when a writer is about to start.

As indicated earlier, this version of the Readers and Writers problem guarantees neither fairness nor any fixed priority in evaluating read and write accesses.

The Readers and Writers problem with 'write' priority

Suppose that in addition to the exclusion constraints of the simple version of this problem, it is required to give a waiting writer priority over waiting readers. Figure 11 shows the resource scheduler that achieves the desired scheduling requirements.

```
procedure readwrite2(wq, rq, rr)
let write be seq(rr,
                 car deq wq,
                 readwrite2(wq, rq, nil)),
    removeread be deq rq,
    read be car removeread
begin if nonempty wq
      then comment service writer;
           write
      else if arbit(waitq wq, waitq rq)
              then comment service writer;
                   write
              else comment service reader;
                   seq(removeread,
                       spar(read,
          readwrite2(wq, rq, spar(read, rr))))
end
```

Figure 11: The readers and writers problem: writers priority

The basic idea is to examine the queue for writers at the start of each new iteration. If there is a waiting writer it will be allowed to execute after ensuring all running readers that have been previously scheduled have completed. Thus, as long as the queue for writers is non-empty, writers will have priority over readers. When the queue for writers becomes empty, waiting readers are allowed to execute concurrently with one another, and will continue to do so until the queue for writers becomes non-empty[(e)].

[(e)]It should be remembered that the operator **nonempty**, unlike **waitq**, does not result in any waiting, but merely returns the current status of the queue.

In the transient situation when both queues are empty, and become non-empty simultaneously, a reader may be selected in preference over a writer. If this situation occurs infinitely often, to consider the worst case, then readers and writers will be served in alternation.

We finally present a skeletal description of a resource **database** that uses the above scheduler, in order to illustrate the overall structure of the typical resource in textual FGL (see figure 12). The dots in the program indicate the absence of details. The where clause allows the nesting of functions and procedures, and is similar to the block-structure of conventional languages. However, the scope of a name does not extend by default over all nested functions and procedures, but must be explicitly imported.

```
resource database()
let actualdatabase be ...
access procedure write ...
       procedure read ...
scheduler dbmanager()
      queues write: wq,
             read: rq
      begin readwrite2(wq, rq, nil)
      where procedure readwrite2(wq, rq, rr)
                          ...
      end
end
```

Figure 12: Skeletal structure of a resource

RELATED WORK

The main thrust of previous work in data-flow models (both data- and demand-driven) has been on determinate and so-called "value-oriented" computations. Efforts at handling indeterminate behavior and the problems of resource control have been few.

We summarize some aspects of related work:

1. **Dataflow Monitors** [1] are closely related to our approach. Their scheduling and arbitration of requests is also explicit and user-programmable, but the underlying computational model is data-driven. Since the transfer of data between operators is the only means of initiating any computation in a data-driven model, two types of operators **entry** and **exit** are used for indeterminately merging all input requests to a resource and for returning the results back to the requesting source[(f)]. For the same reason, data signals, such as **readenable** and **readdone** for a **read** operation, are needed within the scheduler

[(f)]In our approach, a single operator **gqueue** performs both the entry and exit functions. The external demand to access the resource corresponds to the **entry**, and the return of the result by the resource corresponds to the **exit**.

for signalling the start and termination of an operation.

2. The work of Friedman and Wise on Applicative Multiprogramming is also related [6]. An indeterminate constructor **frons** is used for constructing a **multiset**, the order of whose elements is determined only upon access. However, the synchronization of concurrent accesses and resource scheduling are not handled at the level where **frons** is used.

3. **Serializers** [8] have some similarities with our approach. A Serializer is a high-level synchronization construct that has been developed in an Actor message-passing model of computation. Serializers do not require the aforementioned "bracketing" operations, and hence provide good modularity. However, there is a fixed underlying arbitration and scheduling discipline, which is perhaps less flexible than desirable.

4. **Sentinels** [13] come closest to our approach to synchronization and resource scheduling, although the concept has been developed in an Algol-like language, extended with some tasking facilities. A Sentinel is a sequential process that controls the order of evaluation of requests on its input queues. The arbitration and scheduling of requests in a Sentinel is also explicit and user-programmable. In comparison, our scheduler may be thought of as a Sentinel that controls the order of evaluation of FGL expressions.

CONCLUSIONS AND FUTURE WORK

We have presented an approach to resource control that has been influenced strongly by a demand-driven model of resource usage and an applicative style of programming. We have shown that some problems of resource control, such as the avoidance of busy waiting and scheduling, are solved in a more elegant manner under demand evaluation than in conventional models of evaluation. Furthermore, the clarity of our examples indicates that a demand-driven model of execution is well-suited to conceptualizing resources and their control.

The operators introduced here for synchronization and scheduling are representative of a class of machine primitives for resource control in applicative languages, and are not meant to be exhaustive. Although many standard problems in synchronization can be solved quite elegantly using our primitives, in general the adequacy of our primitives from the standpoint of efficiency and convenience of use might be subject to question.

Heretofore, the applicative style of programming has not been explored as a vehicle for resource control in depth. Although we have introduced indeterminate and side-effect operators for arbitration and queueing, the applicative style actually aids well-structured use of these operators, since the order of evaluation of arguments to a function is the only means of achieving any form of "control."

In order to enhance reliability and well-structured use of these primitives, we are also developing an expression-based language, in the sense of Path Expressions [7], for specifying the behavior of our scheduler [10]. We envisage that resource scheduling in FGL will eventually be programmed using such expressions, and a compiler will automatically translate them into the primitives described in this paper. For example, the expression,

$$(p + q)^*$$

specifies that the scheduler may serve any arbitrary <u>sequence</u> of **p**'s and **q**'s. The translation of this expression is the scheduler, **mutex**, described earlier.

The main advantages of such an expression-based language for specifying resource control in FGL are a) the specifications are concise, elegant and the notation makes good stylistic sense, b) the semantics of such expressions can be formalized in terms of FGL graphs, and c) the structure of the translated programs closely preserve the structure of the defining expressions, hence the correctness of the translation may be demonstrated more easily. The main disadvantages are a) it is possible to write ambiguous specifications, and b) additional notation seems necessary for specifying fairness criteria and exclusion/priority constraints based on parameters to operations. Thus, their expressive power is short of being complete.

REFERENCES

[1] Arvind, K.P. Gostelow, and W. Plouffe. Dataflow Monitors. In Proc. of the Sixth ACM Symposium on Operating Systems Principles, (1977), pp. 159-169.

[2] T. Bloom. Evaluating Synchronization Mechanisms. In Proc. of the Seventh ACM Symposium on Operating Systems Principles, (1979), pp. 24-32.

[3] E.J. Courtois, F. Heymans, and D.L. Parnas. Concurrent Control with readers and writers. <u>Communications of the ACM 14</u>, 10 (October 1971), 667-668.

[4] J.B. Dennis. First version of a data-flow procedure language. Lecture Notes In Computer Science, New York, (1974), pp. 362-376.

[5] D.P. Friedman and D.S. Wise. **Cons** should not evaluate its arguments. Automata, Languages, and Programming, Edinburgh, (1976), pp. 257-284.

[6] D.P. Friedman and D.S. Wise. An Indeterminate Constructor for Applicative Programming. In Proc. of the Seventh Annual Symposium on Principles of Programming Languages, (1980), pp. 245-250.

[7] A.N. Habermann and R.H. Campbell. The Specification of Process Synchronization by Path Expressions. Lecture Notes in Computer Science, (1974), pp. 89-102.

[8] C.E. Hewitt and R.R. Atkinson. Specification and Proof Techniques for Serializers. IEEE Transactions on Software Engineering SE-5, 1 (January 1979), 10-23.

[9] C.A.R. Hoare. Monitors: an Operating System Structuring Concept. Communications of the ACM 17, 10 (October 1974), 549-557.

[10] B. Jayaraman. Resource control in a Demand-driven Data-flow Model. Ph.D. Dissertation proposal. Department of Computer Science, University of Utah.(June, 1980).

[11] R.M. Karp and R.E. Miller. Properties of a Model for Parallel Computation: Determinacy, Termination, and Queueing. SIAM Journal of Applied Mathematics 14 (November 1966), 1390-1411.

[12] R.M. Keller. Semantics of parallel program graphs. UUCS-77-110, University of Utah, (July, 1977).

[13] R.M. Keller. Sentinels: a Concept for Multiprocess Coordination. UUCS-78-104, University of Utah, (June, 1978).

[14] R.M. Keller, G. Lindstrom, and S.S. Patil. A loosely-coupled Applicative Multiprocessing System. AFIPS Proc., (1979).

[15] R.M. Keller, B. Jayaraman, G. Lindstrom, J.B. Marti, A.K. Nori, and D. Rose. FGL Programmers' Guide. Unpublished manuscript.(March, 1980).

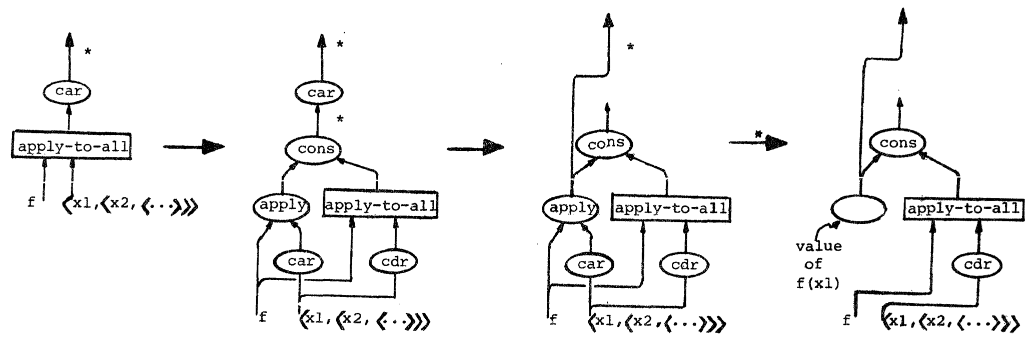


Figure 2: Snapshot evaluation of apply-to-all

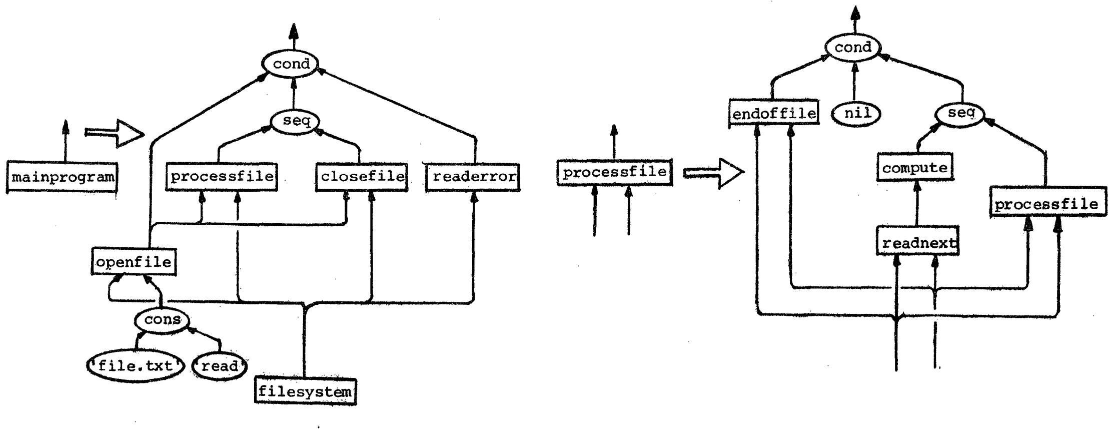


Figure 3a: FGL program for sequential file processing

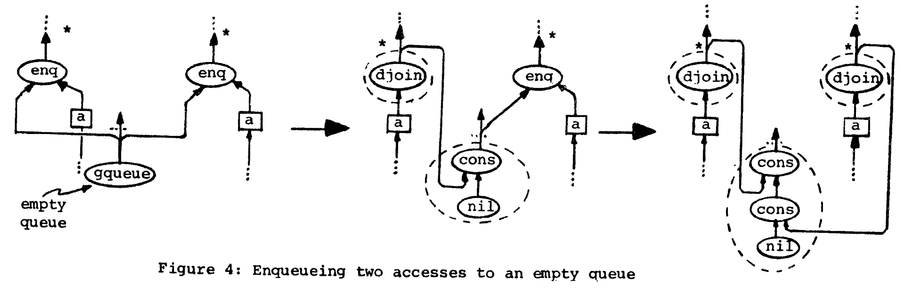
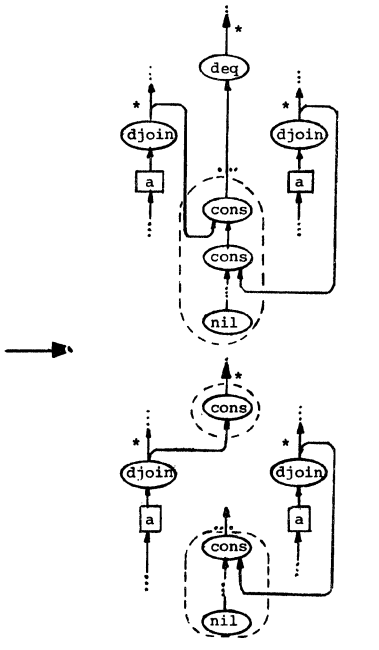


Figure 4: Enqueueing two accesses to an empty queue



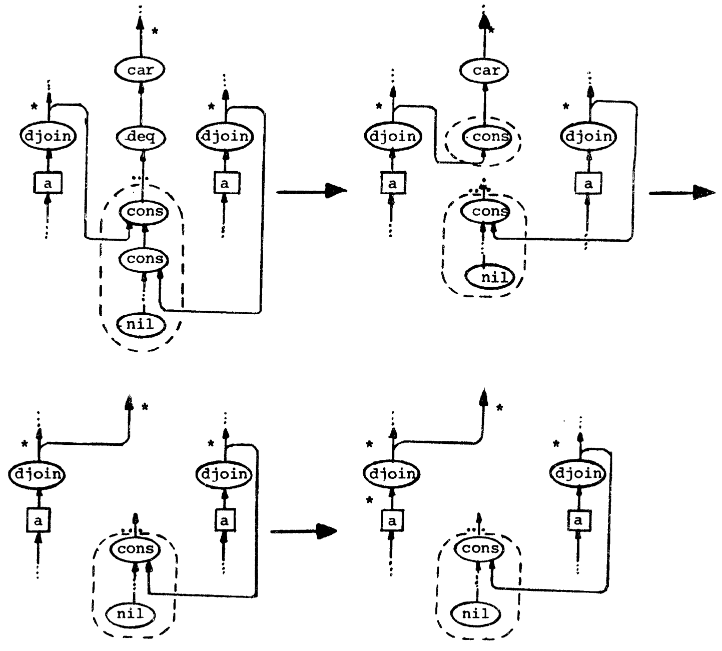


Figure 5: Dequeueing a non-empty queue

Figure 6: Dequeueing a non-empty queue, followed by evaluation

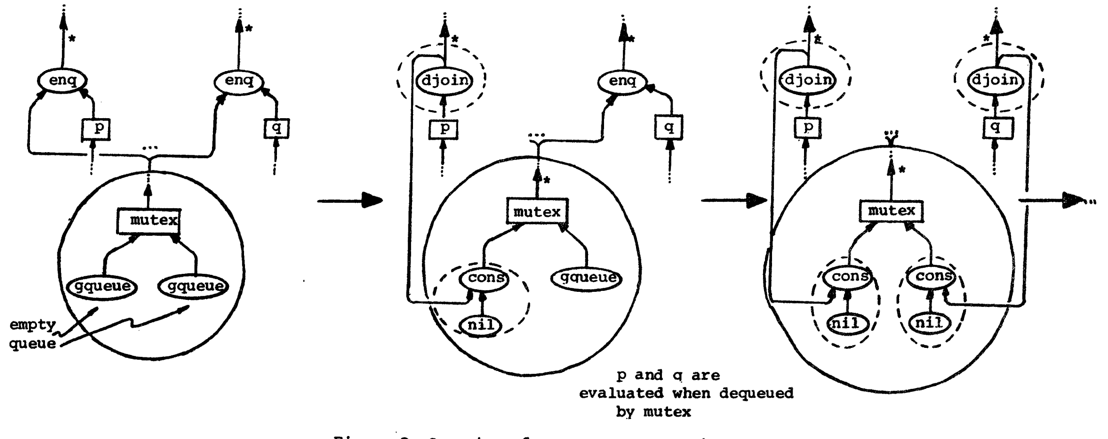


Figure 8: Overview of a resource operation

POSTER SESSION

HIGH-LEVEL OPERATING SYSTEM FORMATION IN NETWORK COMPUTERS

André M. van Tilborg and Larry D. Wittie
Dept. of Computer Science
State University of New York at Buffalo
Amherst, New York 14226

Summary

A network computer is an MIMD computer built from an interconnected collection of independent, asynchronously executing, and loosely coupled processing nodes. Each node consists of at least one CPU attached to a local RAM. The memory of one node is not directly accessible by any other node. An example of such a machine is described in [2].

To control such networks, we have proposed [1] a high-level operating system schema structured as in Fig. 1 as a framework for implementing distributed control techniques. The hierarchical structure can be formed in networks with entirely different physical connection topologies. Each blackened circle represents a node. The links indicate management paths; they do not necessarily represent direct physical connections between nodes. The nodes at level-0 (workers) are available for user tasks. Those at higher levels (managers) are responsible for maintaining the integrity of the local communications subnetwork and for performing resource allocation in ever larger subregions. Although the exact number will vary, each manager node can probably directly handle about 10 to 20 subnodes. To avoid processing bottlenecks, higher level managers use more condensed summaries of allocation information than do lower level managers.

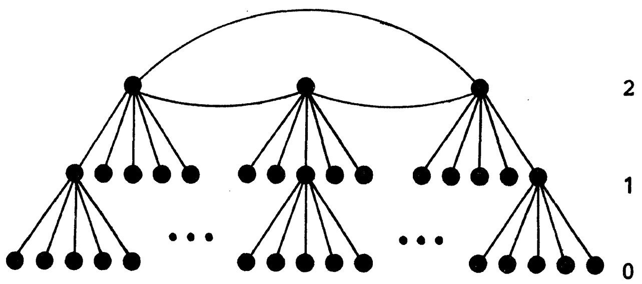


Fig. 1

Since the control schema outlined above is meant to be implemented in arbitrary, and possibly even dynamically changing, network computer topologies, an automatic procedure for creating hierarchies with close to minimal delays between linked nodes is desirable. The "FOCUS of activity" initialization technique introduced in this paper partitions a network computer into layers of processor clusters consisting of managers and their subnodes. To form a management hierarchy, all meaningful activity is initiated by and surrounds a single node, the FOCUS. To approximate heavy message load conditions, all processors send meaningless local messages when they are not participating in FOCUS-initiated activity. The FOCUS is not fixed in one location but rather progresses through the network trailing a chain of inter-foci pointers that are used in later phases of the initialization.

The control hierarchy is built up from the leaves. To produce a hierarchy of L levels, there are L-1 separate but almost identical phases to the initialization. Each phase selects the next higher level of managers until at last a top level (oligarchy) is formed. The technique assumes that the lowest level communications kernel that passes messages between physically connected (neighboring) nodes already exists in each node before hierarchy initialization begins. One arbitrarily selected node is known as the SOURCE. During phase 1 it is both the first and last FOCUS. It is also the last node to be active during each phase.

The objective of the procedure is to assign N subnodes per manager at each level of a tree with the constraint that each path from a level-j manager to its subnodes be shorter than Rsubj physical links. Rsubj is computed from Rsub1 and N. N is supplied as a parameter; Rsub1 may either be supplied or estimated from N.

The following labeled steps and cross-referenced example describe the FOCUS of activity technique in detail:

A. The SOURCE broadcasts a message telling all nodes to obtain the identifiers of their physically connected neighbors and to start sending dummy messages to provide communications background activity. The SOURCE becomes the first FOCUS for phase j=1.
B. The FOCUS sends limited (by Rsubj) broadcast "connect" messages to its neighbors in level-(j-1).
C. Other nodes less than Rsubj links away send "reply" messages back to the FOCUS if they are not yet managers in phase j nor subnodes in any other phase.
D. The FOCUS stores paths to the first N or fewer nodes which reply before a timeout interval expires.
E. If zero nodes reply, the previous FOCUS must select a new FOCUS at step K.
F. Otherwise, the FOCUS accepts each of the N or fewer replying nodes into a new cluster and sends each a list of the identifiers of all the others.
G. Each accepted subnode sends the FOCUS a list of its "connections" outside the cluster. In phase 1 all the physical connections are used; in phase j>1 only the forward and backward pointers to foci in phase j-1 are used.
H. The FOCUS and the worker "nearest" each subnode (in a message-delay sense) in the new cluster send one message to every other node in the cluster. Each worker sends the sum of the reply delay times to the FOCUS.
I. If the FOCUS has the least delay sum it becomes manager for the cluster. Otherwise, the worker with the least sum becomes both FOCUS and manager and receives all the information about the cluster from the previous FOCUS. In phase 1 a deposed FOCUS becomes a subnode of the new FOCUS; in later phases it again becomes a subnode of its previous level-1 manager.
J. The FOCUS lists as possible next temporary foci all the nodes physically connected externally to the cluster, with the less frequently connected (i.e. farthest) ones first.
K. To spread groups far apart, the FOCUS finds the first (farthest) potential temporary FOCUS which is not yet a subnode nor a manager in this phase (j) by polling all the connected nodes. In phase 1 the temporary FOCUS is a worker. In other phases the temporary FOCUS selects a worker from its subtree. The old FOCUS stores the identifier of the worker. The worker becomes the next FOCUS at step B. If no temporary FOCUS is left, then the FOCUS for the previously formed cluster in this phase must select the next FOCUS at step J. If there is no previous FOCUS in this phase then phase j is complete and step L is performed.
L. A FOCUS for a new phase j+1 must be selected by the SOURCE if more than 2N managers were chosen in phase j. Starting with the SOURCE, the subnodes of old foci are searched until a worker is found. The worker becomes the first FOCUS of the new phase j+1 at step B.
M. Otherwise, the 2N or fewer managers in the phase j FOCUS chain exchange identifiers and use limited broadcasts to find short paths to each other. They form the oligarchy of the hierarchy.
N. The SOURCE broadcasts a message to all nodes telling them to stop sending dummy messages.

CH1569-3/80/0000-0131$00.75 © 1980 IEEE

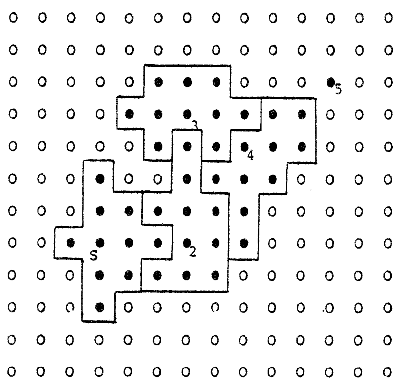


Fig. 2

Figure 2 shows a 4-neighbor mesh network with the SOURCE positioned at point S. For readability the communications links which attach each node (O) to its 4 nearest neighbors are not drawn. Hierarchy formation begins phase 1 with the SOURCE as the first FOCUS. In phase 1 the FOCUS tries to form clusters of N=9 level-0 nodes close to itself by sending limited extent (Rsub1=3) broadcast connect messages (steps A&B). Level-0 nodes which receive the broadcast and are not yet subnodes of any manager respond to the FOCUS (C). In general, during phase j, level-(j-1) nodes which are not subnodes of any manager respond. The cluster of 9 nodes surrounding the SOURCE was formed first by the procedure (D).

Once a cluster is formed, the nodes in that cluster determine among themselves which one can communicate best with all the others (E-I). In phase 1 all of the level-0 nodes exchange messages and pass the total delay times to the FOCUS. In phases j>1 the level-(j-1) managers in the new cluster all tell the "nearest" level-0 node to do this. In this way only worker nodes can become level-j managers and the already formed lower levels of the hierarchy are undisturbed. Thus, a well-situated worker node becomes the FOCUS and manager of a new cluster.

The node which was the FOCUS around which the cluster was formed is only a temporary FOCUS because all record that it was ever the FOCUS is discarded. A chain of forward and backward pointers connects all of the "true", i.e. non-temporary, foci in the order in which they became managers. Since the chain is acyclic there must always be at least one pointer to a level-(j-1) FOCUS outside of a newly formed cluster. The new manager forms a list of all such external connections. In phase 1 the list consists merely of direct physical connections outside the cluster.

To find the next FOCUS in phase j, the current FOCUS scans the list of external connections (J). If one of those nodes is not yet a member of any level-j cluster, then it is given the task of choosing a nearby worker to be the next FOCUS of phase j. In phase 1 the level-(j-1) node is already a worker. When the FOCUS can find no more candidate foci the previous FOCUS in the chain assumes the search for the next FOCUS (K). The phase finishes when all previous foci cannot find an uninitialized node to be the next FOCUS. Since the foci form a tree, the FOCUS will be back at the SOURCE then and the next phase, if needed, can start (L-N).

Figure 3 shows an actual hierarchy formed by the technique during simulations with a network of 27**2 nodes. Each node is marked by a three digit cluster number to which it belongs. Level-1 managers are circled. The number assigned to a manager represents its position in the FOCUS chain. All worker nodes are assigned the same number as their manager. Level-1 clusters are enclosed by straight boundaries; level-2 clusters by curved boundaries. A 729 node mesh is small enough that the edges have an undue influence on level-2 clusters for many combinations of input parameters. Still, the clusters produced by the technique are generally quite compact.

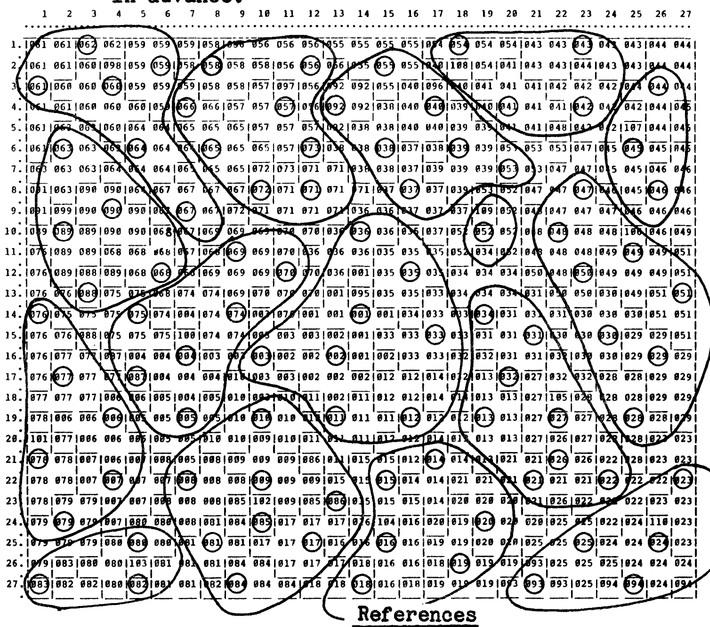
The FOCUS of activity hierarchy formation technique has both good and bad characteristics. One strong point is that the technique does not need to be told anything about the global topology of the network to produce "good" hierarchies. In experiments with mesh networks it has consistently been able to form hierarchies with average link utilizations and path densities only slightly higher than minimal.

Another nice feature is that clusters of nodes can be made as "tight" as desired. Since the nodes in a cluster will tend to communicate often with each other during the solution of multiple task-multiple node problems, it is important that message delays between them be short. Based on the experiments it appears that compact clusters with about N members can be achieved consistently.

Finally, the FOCUS technique does not generate oscillations in hierarchy structure nor does it depend on race conditions. Although the technique will not produce the same structure repeatedly in a given network, the hierarchies will all have almost identical characteristics. Since oscillations are guaranteed not to occur the technique produces a control structure in a short time even for a large network. The FOCUS technique can be regarded as an algorithm for producing hierarchies in network computers.

The technique also has some undesirable features. First, it produces a hierarchy more slowly than parallel techniques we have discovered. Second, all clusters do not necessarily wind up with the same number of nodes. Extra clusters and even levels of hierarchy may be formed. Lastly, there is no way to predict nor control exactly which nodes will cluster together. In certain situations it might be desirable to specify some cluster connections in advance.

Fig. 3



References

[1] L.D. Wittie and A. van Tilborg, "Control Hierarchies for Arbitrarily Connected Microcomputer Networks", SUNY/Buffalo Computer Science Dept. Tech. Rep. 126, (May 1977), 37pp.

[2] L.D. Wittie, "MICRONET: A Reconfigurable Network for Distributed Systems Research", Simulation, (Nov. 1978), pp. 145-153.

DESIGN OPTIMIZATION FOR A SPECIAL-PURPOSE MULTIPLE-COMPUTER*

C. F. Summer
Naval Training Equipment Center, Orlando, FL

and

R. O. Pettus, R. D. Bonnell, M. N. Huhns, and L. M. Stephens
University of South Carolina, Columbia, SC

SUMMARY

The design and performance analysis of the architecture of a special-purpose multiprocessor is presented. The architecture is a hierarchically structured and functionally distributed type. Its operating system is a multilevel structure implemented in an optimal combination of hardware, firmware, and software. This architecture is suited to any application, such as process control or real-time system simulation, in which the basic computational tasks are dedicated and do not change in time.

Each processor has a dedicated memory space in which program tasks are stored. In addition, there is a system bus to a global memory which is used primarily for communication among the processors. To minimize contention for this system bus, selected areas of global memory are duplicated at each processor. This allows the processor to obtain needed information by using a local bus rather than the global, system bus. All write operations to the shared memory are global and the information is duplicated at processors having shared memory at that address. Read operations then become primarily local and can occur in parallel.

Control functions are distributed among the processors; the scheduling and execution of control and application tasks are governed at each processor level by a local, real-time executive. This executive is implemented primarily in firmware to minimize overhead. However, the control structure is designed to be independent of implementation so that a variety of processors can be utilized together. Moreover, it is possible to add to each processor an additional subprocessor which implements the executive in hardware.

A block diagram of the system is shown in Figure 1. Each processor has its own local memory and I/O interfaces as required. In addition, each processor has access to a global shared memory. Access to the shared-memory bus is controlled by a bus arbitration module which implements a multiple-priority, daisy-chained structure. Arbitration is overlapped to provide maximum bus utilization. The control processor occupies the position nearest the arbitration module, giving it the highest priority at each

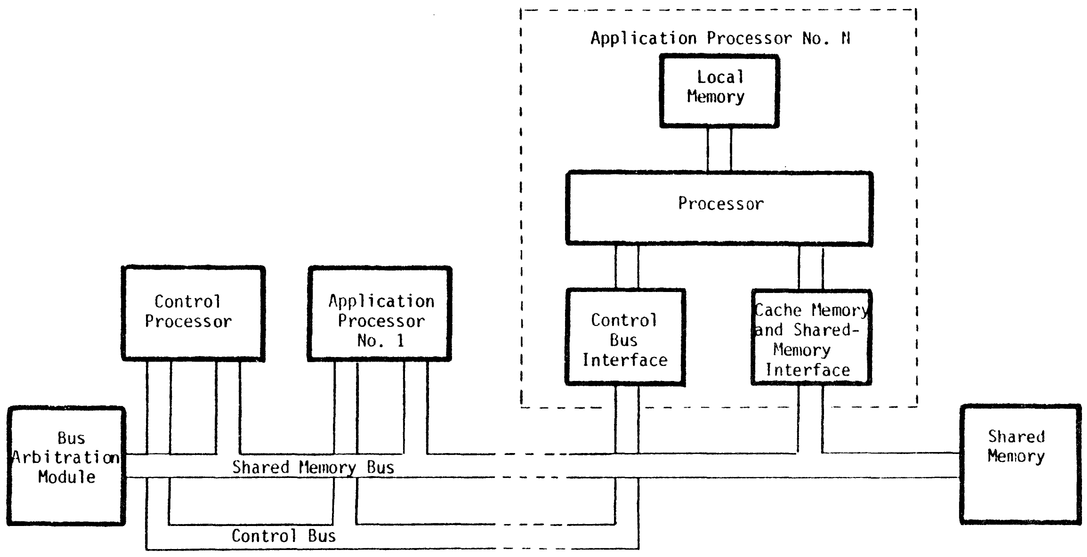


Figure 1. System Block Diagram

*This work was performed for the Naval Training Equipment Center, Orlando, FL, under contracts N61339-78-C-0157 and N61339-79-C-0096.

US Government work not protected by US Copyright

level. Each processor has a control port which is accessed by the control bus. No arbitration is required for this bus as only the control processor may act as the bus master.

The key to successful operation of a multiple-instruction-stream, multiple-data-stream (MIMD) computer is effective communications among the processors. As discussed previously and shown in Figure 1, there are two system buses--one for communicating data and the other for communicating control information--which are common to all of the processors. The most critical system resources are these global buses which, by being shared by all of the processors, become the limiting factor in the overall performance of this multiple-computer system. It is thus crucial that the design and utilization of these buses be optimized.

The architecture of the entire system can be designed to minimize bus usage. Most of the system control functions are distributed among the processors and are handled by the local executive. Also, because the programs to be executed are fixed, each processor is assigned its function in advance. Hence, although one processor is designated as a control processor, it needs to communicate only a minimum of control information during normal system operation. This control information is transmitted on the control bus so as not to interrupt the data flow on the other bus.

One way for processors to communicate is by writing messages and results into a shared memory where other processors can access this information. For the MIMD system described herein, all of the system memory is distributed among the processors. Part of the memory for each processor is local and can be accessed only by that processor. This allows most run-time memory operations to be local, thereby avoiding contention for the global buses. The rest of a processor's memory is global and available to all processors for memory-write operations. This global portion is designed in a dual-port configuration so that it can be read locally while being written globally. Also, all processors can read in parallel without any possibilities for contention or deadlock. By removing all global read operations from the bus, the bus traffic is reduced by much more than half.

As an example of this reduction, if a parameter calculated by one processor is needed by four other processors, a simple shared memory would handle this transfer in five cycles (one to write and four to read). With the shared memory duplicated at each processor, only one cycle is required to simultaneously write the parameter to all processors which need it. The destinations for a parameter are determined by its location in the memory address space. The read operations then occur locally and independently.

An additional architectural feature which maximizes the bandwidth of the global data bus is synchronous operation. This reduces the overhead associated with each data transfer and allows most data transfers to be scheduled.

The utilization of the bus can be further minimized because the system is to be used for a single dedicated application. The program for this application will be partitioned into tasks and assigned to processors for execution in a way that minimizes the interprocessor communications. Also, the communications can be scheduled in advance to minimize idle period for the bus and wait periods for processors, both of which add to communications overhead. Neither of thes optimizations are readily available in a general-purpose MIMD system.

For the multiple-computer system presented in this paper, a cycle is the time allowed to complete a write plus a read on the global shared-memory bus. During each cycle, a set of calculations is also performed by the individual processors. The physical sampling period which consists of several cycles is a function of the significant highest natural frequency of the system being simulated. The sampling period is established by the control processor for all applications processors. Because the total computation is performed by a repetitive sequence of cycles, the speed-up ratio which is a system efficiency measure is based on only one cycle.

Consider a multiple-computer system which has n individual processors and a total computation load of M tasks where a task is a self-contained portion of this load. The average computation time for one task is denoted by T_A. The average time for data exchange on the shared-memory bus per task with only global shared memory is denoted by T_C. The average time for data exchange on the shared-memory bus per task with both local and global shared memory, T_C', is given by

$$T_C' = k\,T_C$$

where k is the local shared memory factor ($0<k\leq 1$). A lower bound for k is $1/(n-1)$.

The average processor utilization for computation, α, is given by

$$\alpha = \frac{T_A}{T_M} \quad (0<\alpha\leq 1)$$

where T_M = the maximum time allowed for computation. Given the above parameters T_A, T_C', n, M, and α the speed-up ratio for the multiple-computer system without distributed control, $\beta_{\bar{d}}$, is given by

$$\beta_{\bar{d}} = \frac{MT_A}{T_D + MT_C' + \frac{M}{n\alpha}T_A}$$

where T_D = duration of control phase. The speed-up ratio for the multiple-computer system with distributed control and with local shared memory β_d^W is thus given by

$$\beta_d^W = \frac{T_S}{T_{P_D}} = \frac{1}{\frac{1}{n\alpha} + \frac{T_C'}{T_A}} .$$

NUMERICAL COMPUTATIONS ON CM*

Peter G. Hibbard and Neil S. Ostlund[a]
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

Summary

Experience related to the suitability of multiprocessors for large scale computations has been mainly limited to synchronous SIMD machines such as Illiac IV, and array and pipeline processors. Relatively little work has been performed on asynchronous MIMD machines such as C.mmp [1], Cm* [2], and S-1 [3]. Algorithmic decompositions which are suitable for such organizations have been studied by Kung [4] and Baudet [5] , but these investigations were not concerned with software organizational problems of task force management, or with the effects of different strategies for memory and processor allocation, or with the effects of different synchronization techniques on the performance of programs using these algorithms.

For the past year we have been involved with assessing the suitability of Cm*-like architectures for large-scale scientific computations, specifically in the area of the approximate solution of Schrödinger's equation for molecular systems, and in the area of the statistical mechanics of liquids. Our current efforts are directed towards the study of a Monte Carlo simulation of the properties of liquid water. This problem has a non-trivial computational complexity and provides an excellent vehicle for studying memory organization, communication, synchronization, and other factors which affect the efficiency of use of a multiprocessor. In addition, such simulations of liquids have been of great interest in recent years and the multiprocessor calculations can be compared directly with extensive related calculations on conventional processors.

The Metropolis Monte Carlo algorithm [6] obtains the properties of a macroscopic liquid by averaging over a large number of random microscopic configurations of a collection of individual molecules. A microscopic configuration is represented by the positions and orientations of a finite number (N) of molecules contained in a "central box". The infinite liquid is simulated by having "mirror image boxes" surround the central box. A new configuration is generated from the current configuration by choosing the next molecule in sequence and moving it to a random new position and orientation. The total potential energy E_i of the new i-th configuration is then computed, using an empirical potential energy function, by summing over the changes in the pair interaction energies of the moved molecule with the N-1 unmoved molecules. The new configuration is accepted or rejected according to a simple decision criterion [6] dependent on the change in potential energy ΔE_{ij} in going from the i-th to the j-th configuration. If ΔE_{ij} is negative the new configuration is accepted; if it is positive then the acceptance probability is $exp\{-\Delta E_{ij}/kT\}$, where k is Boltzmann's constant and T is the absolute temperature. If the new configuration is rejected, another configuration is generated from the current one, and the steps repeated; the current configuration is included as a member of the sampled set as many times as is necessary until a new configuration is accepted. In this way a sequence of configurations is generated which sample the appropriate classical Boltzmann distribution, i.e.,

$$\text{configuration probability} \propto exp\{-E_i/kT\}$$

Macroscopic properties are obtained by a simple averaging over a sequence of O(10^6) configurations.

The bottleneck in the serial Metropolis algorithm is the calculation of intermolecular interactions. For a single move, the time complexity is O(N) because there are N-1 new interactions with the moved molecule. The bookkeeping operations of generating the move, accepting or rejecting the move, etc., are constant-time operations Our initial decomposition scheme, which is almost certainly not the optimum one, uses K processors to evaluate the N-1 interactions with a moved molecule. We employ a master-slave relationship among the processors with the master processor performing the bookkeeping operations and the slave processors evaluating the intermolecular interactions. The algorithm is a synchronized lock-step algorithm; all slaves complete their current activity before new activities are assigned by the master. An asynchronous algorithm would be preferred provided it could be shown to converge to the same Boltzmann average as the present synchronous one. This first attempt at a parallel Monte Carlo algorithm is potentially capable of a speedup which is linear in K the number of processors available. However, to obtain linear speedup requires that memory contention and interprocess bus contention are small, and that synchronization and latency costs are negligible. Initial experiments show that this is far from true and synchronization appears to be particularily costly. Developing an asynchronous algorithm, which we are currently trying to do, is more an exercise in statistical mechanics than in algorithm development.

CH1569-3/80/0000-0135$00.75 © 1980 IEEE

Our initial results have been confined to a small number of interacting atoms and do not use the periodic boundary conditions mentioned above, appropriate to an infinite system. As a simple example, 26 atoms and 25 processors (the master is its own slave) leads to a speedup of 18-20 for unoptimized versions of the program. One of the limiting factors in this sample computation involves having to add up the interaction energies calculated on separate processors. No number of processors *K* can reduce this O(*N*) operation to better than O(log_2N). While the speedup obtained in this sample calculation appears quite reasonable, it is somewhat misleading. The present version of the program uses software double precision floating point routines. With double precision floating point hardware, the amount of local computation relative to global communication would be considerably reduced, and the speedup that could be obtained would be considerably smaller. One of the problems in using Cm* to investigate numerical algorithms is the poor floating point performance of the LSI-11 processors and the consequent need to extrapolate present results to those for a hypothetical multiprocessor with a better floating point capability. From a detailed examination of the Metropolis algorithm, however, it appears that many of the difficulties with the present program can be solved, particularily when the number of molecules *N* increases relative to the number of processors *K*. We are relatively confident that a multiprocessor architecture such as that of Cm* can provide an efficient solution to Monte Carlo calculations of the structure of liquids.

The present program is written in Bliss-11 and runs on the bare Cm* hardware. It is being converted to run on top of the Medusa [7] operating system. We are also extending the current programs to include molecular interactions and periodic boundary conditions, for a system of 256 water molecules using all the 50 processors of Cm*.

References

[1] W.A. Wulf and C.G. Bell, "C.mmp - A Multi-Mini-Processor", Proceedings of the AFIPS 1972 Fall Joint Computer Conference, (December,1972), pp. 765-777

[2] S.H. Fuller, J.K. Ousterhout, L. Raskin, P.I. Rubinfeld, P.J. Sindhu, and R.J. Swan, "Multi-Microprocessors: An Overview and Working Example", Proceedings of the IEEE, (February, 1978), pp. 216-228

[3] T.M. McWilliams, L.C. Widdoes, Jr., and L.L. Wood, Advanced Digital Processor Technology Base Development for Navy Applications: The S-1 Project, Office of Naval Research Technical Report N00014-77-F-0023, (September, 1977)

[4] H.T. Kung, "Synchronized and Asynchronous Parallel Algorithms for Multiprocessors", in Algorithms and Complexity: New Directions and Recent Results, J.F. Traub, editor, Academic Press, New York, (1976), pp. 153-200

[5] G.M. Baudet, The Design and Analysis of Algorithms for Asynchronous Multiprocessors, Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, (April, 1978)

[6] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller, "Equation of State Calculations by Fast Computing Machines", Journal of Chemical Physics, (June, 1953), pp. 1087-1092

[7] J.K. Ousterhout, D.A. Scelza, and P.S. Sindhu, "Medusa: An Experiment in Distributed Operating System Structure", Communications of the ACM (March, 1980), pp. 92-105

[a] Permanent Address: Department of Chemistry, University of Arkansas, Fayetteville, AR 72701.

AN ORGANIZATION OF A THREE-DIMENSIONAL ACCESS MEMORY

H. Shirakawa and T. Kumagai
Dept. of Information Science
Utsunomiya University
Utsunomiya, JAPAN 321

SUMMARY

Multidimensional access memory was proposed by [1]. In this paper a three-dimensional access method will be discussed in detail.

Consider the array given by $w(i,j,k)$ $i,j \in \Omega_N$, $k \in \Omega_{LN}$, where $\Omega_N = \{0,1,2,\ldots,N-1\}$, $\Omega_{LN} = \{0,1,2,\ldots,LN-1\}$, $N = 2^n$, $L = 2^l$ and $L \leq N$. Let $k = k' + k''L$, $k' \in \Omega_L$, $k'' \in \Omega_N$, where $\Omega_L = \{0,1,2,\ldots,L-1\}$. As might be expected, L-bit is used to represent data in the three-dimensional $N \times N \times N$ array.

We consider two kinds of modes to access the array $w(i,j,k)$. Let us call them mode 1 and mode 2.

mode 1

i-slice; $\{w(\alpha,j,k);\ \alpha \in \Omega_N$, for some j and k$\}$

j-slice; $\{w(i,\alpha,k);\ \alpha \in \Omega_N$, for some i and k$\}$

k'-slice; $\{w(i,j,\alpha+k''/(N/L)\cdot N);\ \alpha \in \Omega_N$, for some i, j and k'' which satisfies $k''/\!/(N/L)=0\}$

k''-skips; $\{w(i,j,k'+\alpha L);\ \alpha \in \Omega_N$, for some i, j and k'$\}$

mode 2

i*k'-slots; $\{w((i+\alpha)/\!/N,j,\beta+k''\cdot L);\ (\alpha,\beta) \in \Omega_{N/L} \times \Omega_L$, for some i, j and k''$\}$

j*k'-slots; $\{w(i,(j+\alpha)/\!/N,\beta+k''\cdot L);\ (\alpha,\beta) \in \Omega_{N/L} \times \Omega_L$, for some i, j and k''$\}$

k''*k'-slots; $\{w(i,j,\beta+(k''+\alpha)/\!/N\cdot L);\ (\alpha,\beta) \in \Omega_{N/L} \times \Omega_L$, for some i, j and k''$\}$

where $\Omega_{N/L} = \{0,1,2,\ldots,N/L-1\}$, and / and // represent, respectively, a quotient and a nonnegative remainder after integer division. Two access modes are illustrated in Fig. 1.

Let i, j, k and $m \in \Omega_N$ be represented by a binary representation form, $i = (i_{n-1},i_{n-2},\ldots,i_0)$, $j = (j_{n-1},j_{n-2},\ldots,j_0)$, $k = (k_{n-1},k_{n-2},\ldots,k_0)$ and $m = (m_{n-1},m_{n-2},\ldots,m_0)$. Define the function $\ell(i,j,k,m)$,

$$\ell(i,j,k,m) = (i_{n-1}\oplus j_{n-1}\oplus k_{n-1}\oplus m_{n-1},\ i_{n-2}\oplus j_{n-2}\oplus k_{n-2}\oplus m_{n-2},\ldots,\ i_0\oplus j_0\oplus k_0\oplus m_0)$$

where $\oplus$ represents Exclusive OR.

A q*r shuffle S_{q*r} is defined as a mapping [2],

$$S_{q*r}(i) = (qi + i/r)/\!/qr \qquad 0 \leq i \leq qr-1$$

We define three functions as follows,

$$F(\alpha) = \ell(S_{L*N/L}((i+\alpha/L)/\!/N),S_{L*N/L}(j),\alpha/\!/L,S_{L*N/L}(k''))$$

$$G(\alpha) = \ell(S_{L*N/L}(i),S_{L*N/L}((j+\alpha/L)/\!/N),\alpha/\!/L,S_{L*N/L}(k''))$$

$$H(\alpha) = \ell(S_{L*N/L}(i),S_{L*N/L}(j),\alpha/\!/L,S_{L*N/L}((k''+\alpha/L)/\!/N)) \qquad \alpha \in \Omega_N$$

Theorem 1; If $\sigma_{(p,q,r)}$ $p,q,r \in \Omega_N$ can be expressed by the description,

$$\sigma_{(p,q,r)} = \begin{pmatrix} 0 & \ldots & N-1 \\ \ell(p,q,r,0) & \ldots & \ell(p,q,r,N-1) \end{pmatrix}$$

Then $\sigma_{(p,q,r)}$ is the permutation on the set Ω_N, and can be realized by an n-stage shuffle-exchange network[3].

The permutations required for access mode 1 are defined as follows.

$$\sigma_{i,j,k'} = \sigma_{(i,j,k'\cdot N/L)}$$
$$\sigma_{i,j,k''} = \sigma_{(i,j,k'')}$$
$$\sigma_{i,k',k''} = \sigma_{(i,k'\cdot N/L,k''/(N/L)\cdot(N/L))}$$
$$\sigma_{j,k',k''} = \sigma_{(j,k'\cdot N/L,k''/(N/L)\cdot(N/L))}$$

It can be shown that $\sigma^{-1}_{i,j,k'} = \sigma_{i,j,k'}$, $\sigma^{-1}_{i,j,k''} = \sigma_{i,j,k''}$, $\sigma^{-1}_{i,k',k''} = \sigma_{i,k',k''}$ and $\sigma^{-1}_{j,k',k''} = \sigma_{j,k',k''}$. Control signals required in the realization of the $\sigma_{(p,q,r)}$ are n and are uniform in each stage.

Theorem 2; If σ_i, σ_j and σ_k can be expressed by the description, respectively,

$$\sigma_i = \begin{pmatrix} 0 & \ldots & N-1 \\ F^{-1}(0) & \ldots & F^{-1}(N-1) \end{pmatrix}$$
$$\sigma_j = \begin{pmatrix} 0 & \ldots & N-1 \\ G^{-1}(0) & \ldots & G^{-1}(N-1) \end{pmatrix}$$
$$\sigma_k = \begin{pmatrix} 0 & \ldots & N-1 \\ H^{-1}(0) & \ldots & H^{-1}(N-1) \end{pmatrix}$$

Then σ_i, σ_j and σ_k are the permutations on Ω_N, and can be realized by an n-stage shuffle-exchange network.

Theorem 3; The inverse permutations of σ_i, σ_j and σ_k, denoted σ_i^{-1}, σ_j^{-1} and σ_k^{-1}, respectively, can be realized by an n-stage shuffle-exchange network.

It should be noted that in general $\sigma_i \neq \sigma_i^{-1}$, $\sigma_j \neq \sigma_j^{-1}$ and $\sigma_k \neq \sigma_k^{-1}$. Control signals required in the realization of σ_i, σ_i^{-1}, σ_j, σ_j^{-1}, σ_k and σ_k^{-1} are $(l+1)\cdot N/L-1$.

Three-dimensional access memory is physically implemented by N RAM chips, each organized as a one bit by N^2L-word memory. A cell location is denoted by $m(I,J)$ $I \in \Omega_N$, $J \in \Omega_{N^2L}$, where $\Omega_{N^2L} = \{0,1,2,\ldots,N^2L\}$.

WRITE operation to the three-dimensional access memory is performed by applying a permutation σ on the set of data indices and by addressing for each memory chip. Conversely, READ operation is performed by addressing for each memory chip and applying an inverse permutation σ^{-1} on the set of chip numbers I.

Storage schemes for mode 1 and mode 2 are given as follows.

mode 1; A memory cell $m(I,J)$ contains a three-dimensional array entry $w(\ell(I,J/\!/N,(J/N)/\!/L\cdot(N/L),J/NL),J/\!/N,J/N)$ $I \in \Omega_N$, $J \in \Omega_{N^2L}$. Conversely, a three-dimensional array entry $w(i,j,k)$ is stored in a memory cell $m(\ell(i,j,k'\cdot N/L,k''),j+kN)$ $i,j,k'' \in \Omega_N$, $k' \in \Omega_L$, $k \in \Omega_{NL}$.

mode 2; A memory cell $m(I,J)$ contains a three-dimensional array entry $w(S_{N/L*L}(\ell(I,J/\!/N,(J/N)/\!/L,J/NL)),S_{N/L*L}(J/\!/N),(J/N)/\!/L+S_{N/L*L}(J/NL)\cdot N)$ $I \in \sigma_N$, $J \in \Omega_{N^2L}$. Conversely, a three-dimensional array entry $w(i,j,k)$ is stored in a memory cell $m(S_{L*N/L}(\ell(i,j,S_{N/L*L}(k'),k'')),S_{L*N/L}(j)+k'N+S_{L*N/L}(k'')\cdot NL)$ $i,j,k'' \in \Omega_N$, $k' \in \Omega_L$, $k \in \Omega_{NL}$.

Addresses J, permutations σ and inverse permutations σ^{-1} for each access mode are summarized in Table 1.

Addressing circuitry is realized by Exclusive OR gates for mode 1. However, N/L n-bit adders are required for mode 2. Rewriting address J for memory chip I in a more convenient form, we get for i*k'-slots,

$$J = k''/\!/(N/L)\cdot 2^{n+2l}+k''/(N/L)\cdot 2^{n+l}+\ell\{j/(N/L),I/\!/L,[i+\ell(j/\!/(N/L),I/L,N/L-1,k''/\!/(N/L))]/(N/L),k''/(N/L)\}\cdot 2^n+j/\!/(N/L)\cdot 2^l+j/(N/L)$$

The coefficient of 2^n shows that N/L n-bit adders can determine the address. Similar arguments can be applied for both j*k'-slots and k"*k'-slots.

Now, we consider a block-oriented access of slots, i.e., $i/\!/(N/L)=0$ for i*k'-slots, $j/\!/(N/L)=0$ for j*k'-slots and $k''/\!/(N/L)=0$ for k"*k'-slots. In this case, the coefficient of 2^n is $\ell(j/(N/L),I/\!/L,i/(N/L),k''/(N/L))$. Thus, addressing circuitry can be implemented using only AND and Exclusive OR gates. Moreover, control signals of the shuffle-exchange network in mode 2 become n and are uniform in each stage as well as in mode 1.

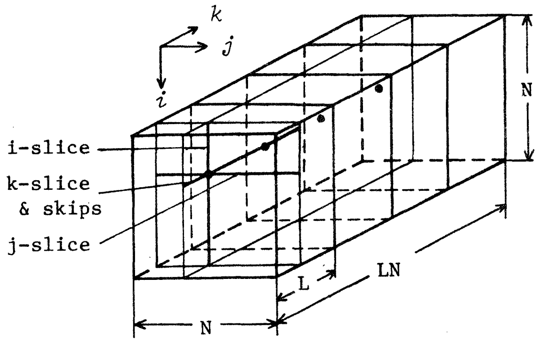
There exists an algorithm of data exchange in the memory to switch from one access mode to the other mode. Data exchange is performed through memory-to-memory data path with data permutation network. Necessary permutations are implemented by an n-stage shuffle-exchange network and a wired shuffle network connected in cascade.

The algorithm may be solved in $O(3N^2L)$ steps, where step is a total time of a fetch cycle time, a propagation time through the data permutation network and a store cycle time.

As a result, a technique for organization of a three-dimensional access memory is given.

REFERENCES

[1] Batcher,K.E.,"The Multidimensional Access Memory in STARAN", IEEE Trans. Comput., Vol. C-26, No.2, pp.174-177 (1977)
[2] Patel,J.H.,"Processor-Memory Interconnections for Multiprocessors", Proc. 6th Annual Symp. on Computer Architecture, pp.168-177 (1979)
[3] Lang,T. and Stone,H.S.,"A Shuffle-Exchange Network with Simplified Control", IEEE Trans. Comput., Vol.C-25, No.1, pp.55-65 (1976)



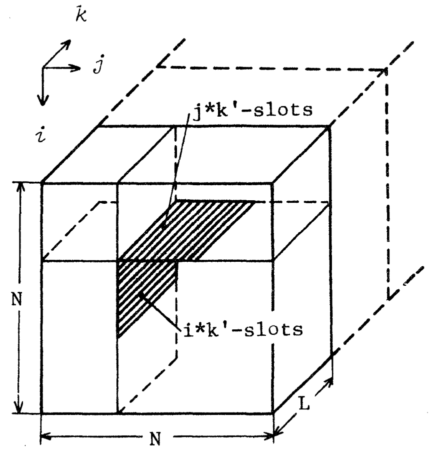


Fig.1. Three-dimensional access

access	address J	σ	σ^{-1}
mode 1			
i-slice	$j+kN$	$\sigma_{j,k',k''}$	$\sigma_{j,k',k''}$
j-slice	$\ell(i,I,k'\cdot N/L,k'')+kN$	$\sigma_{i,k',k''}$	$\sigma_{i,k',k''}$
k'-slice	$j+S_{N/L*L}(\ell(i,j,I,k''))N+(k''/(N/L))N^2$	$\sigma_{i,j,k''}$	$\sigma_{i,j,k''}$
k"-skips	$j+k'N+\ell(i,j,k'\cdot N/L,I)NL$	$\sigma_{i,j,k'}$	$\sigma_{i,j,k'}$
mode 2			
i*k'-slots	$S_{L*N/L}(j)+(F^{-1}(I)/\!/L)N+S_{L*N/L}(k'')NL$	σ_i	σ_i^{-1}
j*k'-slots	$S_{L*N/L}((j+G^{-1}(I)/L)/\!/N)+(G^{-1}(I)/\!/L)N+S_{L*N/L}(k'')NL$	σ_j	σ_j^{-1}
k"*k'-slots	$S_{L*N/L}(j)+(H^{-1}(I)/\!/L)N+S_{L*N/L}((k''+H^{-1}(I)/L)/\!/N)NL$	σ_k	σ_k^{-1}

Table 1.

Loop Decomposition in the Translation of Sequential Languages to Data Flow Languages*

Stephen J. Allan
Department of Computer Science
Colorado State University
Ft. Collins, CO 80523

Arthur E. Oldehoeft
Department of Computer Science
Iowa State University
Ames, IA 50011

Summary

This paper deals with compile-time exposure of parallelism in high level sequential programs for eventual execution on a data flow computer. Specifically, it deals with the analysis and restructuring of loops which are written for sequential execution, transforming them into loops whose iterations may be conceptually executed in parallel. The technique presented in this paper differs from other techniques in that outer loops are examined, even if some of the inner loops need to be executed sequentially. There are certain types of parallel computers (e.g., data flow computers) on which the parallel execution of outer loops may yield significant reduction in execution time even though some inner loops are executed sequentially.

A data flow machine [2,3,5,6] is a highly parallel, asynchronous computer. The assumption underlying a data flow computer is that a program is not a sequence of instructions that cause changes to a memory space, but instead a program is a collection of computations related to each other by the need for data values that are produced and consumed. The order of execution of the computations is not directly stated by the program but rather by the partial ordering provided by the data dependencies. The derivation of this partial ordering is detailed in [1]. Therefore, transforming sequential loops to parallel loops, whose iterations are independent, may greatly reduce the execution time of a program executing on a data flow machine.

Loop decomposition has been proposed as a technique which attempts to decompose the body of a loop into several smaller loops while maintaining the data dependencies between the statements. In the loop decomposition technique by Lo [4], the entire loop is initially analyzed to see if the iterations are independent. If they are independent, the loop is directly transformed into a parallel loop. If they are not independent, the loop is examined to see if the iterations can be made independent through forward substitution or saving of values in a temporary array. If the loop cannot be transformed into a parallel loop using these transformations, the loop is then decomposed into smaller loops. Each of these smaller loops is analyzed to see if its iterations are independent or if the iterations can be made independent through the use of the above transformations.

Loop decomposition techniques are applied to the innermost loops first. If an inner loop cannot be transformed into a parallel type loop, no attempt is made to transform the enclosing loops. The reason for this is parallel machines typically do not take advantage of the parallelism available in the outer loops if some of the inner loops are executed sequentially. In a data flow environment this is important because parallel execution of the outer loops may yield significant reductions in execution time even though the inner loops are performed sequentially. For this reason, all loops are analyzed regardless of the type of statements that appear in the body of the loop.

The algorithm discussed below extends in two ways the method introduced by Lo. First, the requirement that an array name appear on the left side of an assignment statement only once in the body of a loop has been eliminated. This facilitates the transformation of non single-assignment high level sequential languages to a data flow language. Second, the requirement that the body of the loop consist of only assignment statements has also been eliminated. Any type of statement, including compound statements, may appear in the body of the loop.

A brief general description of the algorithm is given here. There are two matrices associated with the algorithm called "order" and "try". The "order" matrix contains a row and a column for each statement in the body of the loop. Entries in the "order" matrix indicate that an ordering relation exists between two statements in the body of the loop. A "t" in order(i,j) indicates that statement i must be executed before statement j because of data dependencies or interference in the usage of storage caused by parallel execution of the loop. The data dependencies may occur across iterations of the loop. Each time an entry appears in the "order" matrix, a corresponding entry appears in the "try" matrix. An entry in the "try" matrix has a list of transformations which are applicable in breaking a cycle in which the two statements might appear. A cycle indicates that the statements in the cycle have data dependencies on each other. The different transformations used in restructuring a loop for parallel execution are forward substitution of expressions, saving values in a temporary array, or changing a scalar value into an array value. If none of these transformations are applicable in breaking a particular cycle, an indication of this is placed in the "try" matrix. Both of these matrices are constructed at compile time during the analysis of the body of the loop.

The compile-time loop analysis proceeds as follows. All statements in the body of the loop

----

* Research reported herein was supported in part by the National Science Foundation under Grant NCS77-02467

CH1569-3/80/0000-0139$00.75 © 1980 IEEE

are analyzed to determine their relationship with the other statements in the body of the loop. If any statement in the body of the loop is a compound statement, all the statements in the body of the compound statement must also be analyzed to determine its relation with the other statements. The details of the data flow analysis needed to analyze the data dependencies appear in [1]. A statement is analyzed in the following manner. Every value defined by the statement is analyzed by finding all the uses of the value in the body of the loop. The uses are found in a list which is associated with each value defined by the statement. Each use is compared with its definition in the loop to find its relation. If the definition in the loop is prior to its use in the same iteration, it is possible to use the forward substitution transformation to break a cycle in which the statements might be contained. If a use appears prior to its definition in the same iteration, or a previous iteration, it is possible to save the old values in a temporary ar in which the statements are involved. These facts are noted in the "try" matrix. If a scalar value is assigned, a note is made in the "try" matrix indicating that the scalar value must be changed to an array value if the statement is to appear in the body of a forall construct. All values defined by a given statement are analyzed, as described above, and the "order" and "try" matrices are formed.

Once the "order" and "try" matrices have been formed by the data flow analysis routine, the loop decomposition algorithm proceeds in the following manner. The "order" matrix is analyzed for cycles. If no cycles are found, the iterations of the loop are independent and the loop may be transformed directly into a forall construct. If there are cycles, an attempt is made to break the cycles using the transformations mentioned above. If the attempt is successful, the loop is transformed into a forall construct. If the attempt is unsuccessful, the loop is decomposed into minor loops. A minor loop contains either a cycle or a single statement. Each minor loop that contains only a single statement can be transformed into a forall construct as long as the single statement does not have a data dependency on itself. If the single statement has a recursive data depencency, it must be executed as a sequential loop. Each minor loop which contains a cycle is analyzed to see if the cycle can be broken by the transformations noted in the "try" matrix. If the saving of values in a temporary array or forward substitution techqniues break the cycle, the minor loop is transformed into a forall construct. If not, the minor loop must be executed sequentially. If any scalar values are assigned in a loop which has been transformed to a forall construct, the scalar value must be changed to an array value.

Consider the program segment in Figure 1 which multiplies two matrices, a and b, and produces a matrix c. Assume that the array a is l x n, the array b is l x m, and the array c is m x n.

```
do i = 1 to l
  do j = 1 to n
    a(i,j) := 0
    do k = 1 to m
      a(i,j) := a(i,j) + b(i,k) * c(k,j)
    end
  end
end
```

Figure 1 Matrix multiplication

This program segment is analyzed using the technique given above and finds that the innermost loop has to be executed sequentially, but the outer two loops may be transformed into forall constructs. This is done giving the program segment in Figure 2.

```
forall i in (1,l) do
  forall j in (1,n) do
    a(i,j) := 0
    do k = 1 to m
      a(i,j) := a(i,j) + b(i,k) * c(k,j)
    end
  end
end
```

Figure 2 Transformed matrix multiplication

The resulting speedup of the transformed loop depends on the manner in which the forall construct is implemented. Assuming the index values in a forall construct are generated sequentially, it is possible for the program in Figure 2 to be executed in O(l+m+n) time. As it appears in Figure 1, the data dependencies are not known so that the code generated results in O(l*m*n) execution time.

References

1. S. J. Allan and A. E. Oldehoeft, "A Flow Analysis Procedure for the Translation of High Level Languages to a Data Flow Language," IEEE Transactions on Computers (to appear).

2. Arvind and K. P. Gostelow. A Computer Capable of Exchanging Processor Elements for Time, Computer Science TR-77, University of California, Irvine, (January, 1976).

3. J. B. Dennis and D. P. Misunas, "A Preliminary Architecture for a Basic Data Flow Processor," The 2nd Annual Symposium on Computer Architecture, IEEE, New York, (1975), pp. 126-132.

4. D. Lo, Transformation of Loop Programs for Parallel Execution, Ph.D. Thesis, The University of Michigan, (1976).

5. J. Rumbaugh, "A Data Flow Multiprocessor," IEEE Transactions on Computers (February, 1977), pp. 138-146.

6. K. Weng, Stream-Oriented Computations in Recursive Data Flow Schemas, M.S. Thesis, M.I.T., Cambridge, Mass., (1975).

GOODYEAR AEROSPACE CORPORATION'S MICROCOMPUTER ARRAY PROCESSOR SYSTEM

F. G. Carty and R. H. Ries
Digital Technology Department

Goodyear Aerospace Corporation
Akron, Ohio 44315

Goodyear Aerospace Corporation's Microcomputer Array Processor System [1] is a programmable multiprocessor computer system designed for Electronic Warfare applications for the Air Force Avionics Laboratory (AFAL). The applications involved sorting, identifying and tracking emitter signals in real time for very dense radar environments. The main problem in achieving this goal is that the signal densities constitute a severe data processing load which greatly exceeds the capability of present airborne computer systems.

The architecture of this system (Figure 1) retains many of the classic multiprocessor design concepts including a master-slave relationship among its microprocessors in a tightly coupled structure. Each processor is a 32-bit programmable computer with its own dedicated memory and a capability to execute approximately four million instructions per second. Each processor can communicate with several banks of common memory (referred to as global memory). The global memory modules and their communication structure tie the individual processors together in a symmetrical multiprocessor computer architecture. The multiprocessor system is modular and can contain at least two and at most eight processors coupled with up to sixteen banks of global memory and executes up to 32 million instructions per second. Expansions beyond these limits are possible if every processor does not have to have access to every global memory module. Currently, a four processor system (with three banks of global memory) is installed at Wright Patterson AFB for use by AFAL. This system will be expanded to six processors during 1980. This multiprocessor subsystem occupies approximately 1.6 cu. ft. and consumes under 400 watts.

Global memory is implemented as several independent memory banks to allow simultaneous accesses (i.e., one concurrent access per bank). Each memory bank contains at least 1024 32-bit words and can be accessed by each processor.

Global memory can be used to store data common to several processors, to swap programs with or add subprograms to those in local memory of any processor and to facilitate communication between processors and/or other subsystems. This last use can be accomplished by message switching techniques and may be initiated via software polling or via hardware driver interrupts.

Conflicts between microprocessors in accessing global memory are generally minimal in the current application for three reasons. First, each microprocessor has its own dedicated memory which contains its program instructions and local variables. Next, in the current application we can predict relative accesses due to various parameters so that algorithms were chosen which distributed global memory accesses uniformly. Finally, each microprocessor executed many more computational instructions than global memory accesses.

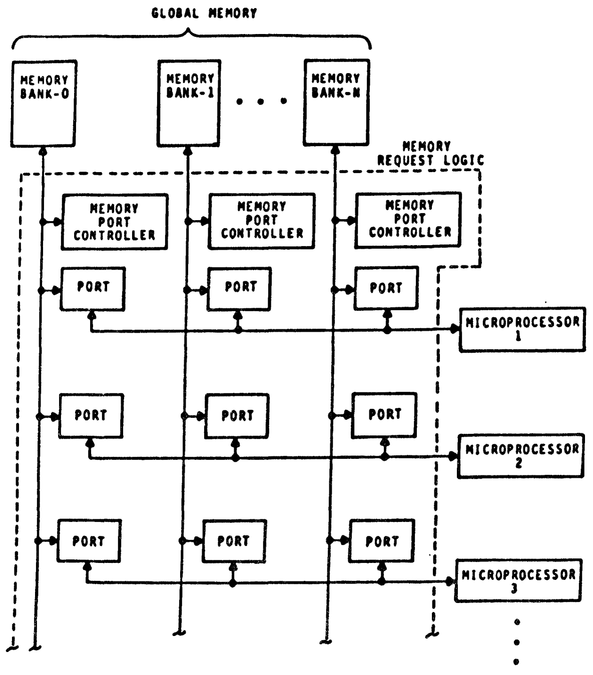


Figure 1. Architecture of the Microcomputer Array Processor System

CH1569-3/80/0000-0141$00.75 © 1980 IEEE

The memory request logic connects the global memory banks to each processor (and peripheral device) in a manner which will support as wide a communication bandwidth as possible without requiring an unreasonable amount of hardware. The logic structure also allows expansion in the number of processors and/or memory banks. These features led to the multi-port multi-bus communication structure. A port structure and a port controller are attached to each memory bank. Any processor (or device) which is to communicate with a given memory bank must be connected to a port associated with the bank. A microprocessor initiates a global memory access by issuing a request over its output bus. Each port determines if the request belongs to the address space of its memory bank and hence, only the proper port will accept the request. If the requested memory bank is not currently busy the request will be serviced immediately. Otherwise, the request is held in the port. When the memory becomes available all requests held (a maximum of one per processor) are queued into the memory port controller and serviced on a priority basis. Each request requires approximately 200 ns to be serviced. It takes a minimum of 750 ns for a processor to make a request. Thus, if the average rate of accessing for any given global memory bank is less than three requests per 750 ns period the memory access structure is transparent to the processor and no time is lost by the processor.

Each of the processors contains a CPU, program memory, a microprogram sequencer, a pipeline register, a condition decoder, clock and timer and an interrupt control. The CPU contains sixteen addressable registers and an arithmetic logic unit. The program memory contains microcode for the program and local data. The microprogram sequencer causes program memory to sequence through its microcode in proper order. The pipeline register holds the current instruction being executed so that program memory maybe released to fetch the next instruction. The condition decoder is used to facilitate conditional branching. The clocking and timing unit allows each instruction type to be executed at the fastest rate possible. The interrupt control allows the processor to respond to asynchronous external stimulus without resorting to polling.

The architecture of the individual processors is centered around utilization of LSI bipolar bit slice technology rather than the MOS "computer on a chip" which is more commonly associated with the term microprocessor. Each processor is composed of eight 4-bit CPU chips which are cascaded together to form a high speed 32-bit processor. The resulting processor is capable of executing register-to-register operations (i.e., adds, subtracts and logical operations) on 32-bit data words in under 300 ns.

Bench testing of the system has shown that multiprocessor based systems are a practical solution to the application problem. During bench testing two observations were made. First, that prolonged operation of this system has clearly demonstrated that (for this application at least) very high subsystem interaction rates can be supported in a cost effective manner through proper hardware design. The second observation was that for dedicated use in a master/slave mode maintaining control and coordination in this application among various processors is not an overly difficult task.

****

This work was supported by Air Force Avionics Laboratory, Air Force Systems Command, Wright-Patterson Air Force Base, Ohio 45433 under Contract F33615-75-C-1179 Project 7633.

References

[1] R. H. Ries, Microcomputer Array Processor, Air Force Avionics Laboratory, AFAL-TR-78-157, (October, 1978) 131 pp.

SIMULTANEITY OF EVENTS IN PETRI NETS

R.C.O. Martins and K.B. Irani
Department of Electrical and Computer Engineering
The University of Michigan
Ann Arbor, MI 48109

SUMMARY

Probably the most obvious and intuitive property of concurrency (or parallelism) is not simultaneity of events. Although various different formal models of parallelism have been proposed which model different aspects of parallelism and synchronization ([1]), simultaneity of events has not been directly represented in any of those models. Rather it has always been represented by interleaving distinct events into sequences and it has been analyzed by studying the properties of the set of all such sequences. Thus, in Petri nets we have "firing-sequences", which are total orderings of occurrences of events. In this context, two events a and b are "simultaneous" if xaby and xbay are possible sequences of events in the modelled system, for some sequences x and y. It is very simple and convenient to represent simultaneity in terms of sequences with implicit interleaving. However, as was shown by Miller and Yap ([4]), interleaving alone is a weaker notion than simultaneity and only under certain conditions can simultaneity be represented by a form of interleaving . It should also be pointed out that this inability of modelling simultaneity of events exactly is not peculiar to ordinary Petri nets, but is evident in all its previous extensions as well.

In this work we have developed a new version of Petri nets, called "Timed Petri nets", which directly represents simultaneity of events. We are mainly interested in the effects of this extension (i.e. modelling of simultaneity) on the complexity of the formal properties which can be tested on the nets and which are useful for system analysis.

A Timed Petri net (TPN, for short) is defined as a pair (PN,Tm), where :

- PN = (P,T,I,O,Mo) is a generalized Petri net (see [2]) such that every transition has at least one input place.
- Tm : $(Zo)^n \times T \rightarrow N$, where N=Zo − {0} and Zo is the set of nonnegative integers and n=|P| (the cardinality of set P). Tm is the "firing time function".

This definition implies that a TPN has the same structure as a Generalized Petri net. However it has different flow of tokens, as we will see later on.

The function Tm assigns to each transition t in T a "firing time", the time interval that the transition t takes to fire. The firing time of each transition is also a function of the current marking of the net.

This research was supported by INPE and CNPq (Brazil).

Let ζ be a nonnegative real number. We denote by $M(\zeta)$ the marking of the net at time instant ζ. Then $M(\zeta)$ denotes the number of tokens in the i'th place p_i at time instant ζ. As usual, $M(\zeta)$ can be represented by a vector where the i'th component, $M(\zeta)(i)$, is given by $M(\zeta, p_i)$. By definition, M(0) = Mo.

A few attempts have been reported at introducing time as a new parameter in Petri nets, but only to permit the analysis of system performance. We had a different objective and we have followed a distinct approach. Roughly speaking, we characterize parallelism of events by the firing of transitions in a TPN under two very intuitive notions :

- events can occur simultaneously (or in parallel) only if they are independent. The characterization of independence among events (or firing of transitions) at each control state (or marking of a TPN) is crucial throughout the whole paper.
- only independent events can occur simultaneously. Time was introduced in TPN by associating a firing time to each transition in the net only in order to support a primitive notion of simultaneity.

Under that approach, a transition is said to be enabled at a given marking M at a time instant ζ iff it is enabled in the usual sense defined in Petri nets (i.e., $M(\zeta) \geq I(t)$) and t is not firing at this time instant. However only certain subsets of the set $E[M(\zeta)]$ of enabled transitions at $M(\zeta)$ are "simultaneously firable". This is defined by an independence relation on the power set of $E[M(\zeta)]$, called "Simultaneity Relation" $Sy[M(\zeta)]$. If A and B belong to the power set of $E[M(\zeta)]$, then:

- $(A,B) \in Sy$ iff $A \not\subseteq B$ and $B \not\subseteq A$ and
  $M(\zeta) \geq \Sigma I(t)$ (where $t \in A \cup B$).

Sy is a symmetric and irreflexive relation.

Using the relation $Sy[M(\zeta)]$ a family of sets, called $S[M(\zeta)]$, is defined such that each element SF_i of the set S at marking $M(\zeta)$ is a set of simultaneously firable transitions. For every set $SF_i \in S$ the following conditions are satisfied:

a) $|SF_i|=1$ or $\forall B,C \in SF_i$ s.t. $B \not\subseteq C$ and $C \not\subseteq B$ then $(B,C) \in Sy$.

b) $\forall B \notin SF_i$ then $(SF_i,B) \notin Sy$.

S is a cover of the set $E[M(\zeta)]$ and all the transitions in one of these sets SF_i can initiate their firing at the same time instant ζ. The choice of what set $SF_i \in S$ will initiate firing at time instent ζ is arbitrary. However, once a set is selected for firing, every transition t in this set will initiate its firing at time instant ζ^+ by removing I(p,t) tokens from each input place p, and will be "firing" until time instant $Tm(t)+\zeta$, when it deposits O(t,p) tokens in each output place p.

CH1569-3/80/0000-0143$00.75 © 1980 IEEE

Condition b in the previous definition implies that each set SF_i of the cover S is a "maximal" set. Thus a new set of transitions can be enabled only at markings defined by the termination of transition firings, or the initial marking Mo=M(0). These markings are then called "active markings".

In order to specify the state of a TPN at a given time instant ζ, we need then to specify the marking $M(\zeta)$ and the termination times of all transitions which are firing at time instant ζ. If this is done for active markings, the behavior of a TPN can be completely described.

An "instantaneous description" of a TPN at time instant ζ is a pair $(M(\zeta),r)$, where

-$M(\zeta)$ is an active marking.

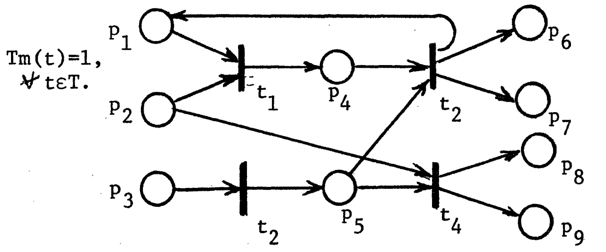
-r is a vector such that r(i) is the remaining firing time of transition t_i, defined at time instant ζ.

Under these firing rules it can be shown that a TPN is able to represent non-monotone predicates on the set of its markings, contrary to ordinary Petri nets that only represent monotone predicates ([4]). For instance, for the TPN in figure 1, if p_2 and p_3 have initially one token each, the net can be used to test whether or not p_1 has a token at M(0), since this fact determines the final marking M(2). However, if the firing rules of ordinary Petri nets are followed, it can be easily seen that this test cannot be made on Petri nets.

Using the net of figure 1, it can be shown that TPN can simulate any 2-counters automaton and, therefore, Turing Machines. It is also possible to show that the simultaneity of events (or firing of transitions) is by itself a sufficient condition for simulation of 2-counters automata by TPN's.

As TPN's retain the structure of generalized Petri nets, they can represent all the problems of parallelism and synchronization modelled by these simpler nets, and also directly represent the simultaneity of events in real systems. It can be shown that TPN's have also enough features to limit, when necessary, the number of simultaneous firing of transitions. In the extreme situation only interleaving can be allowed. In this case the simultaneity relation Sy is empty and each element of the set S, i.e., a set of simultaneously firable transitions, is a singleton set. Therefore, under our formalization of TPN, interleaving can be seen as a degenerate case of simultaneity of events.

FIGURE 1



Mo=(1,1,1,0,0,0,0,0,0) $\rightarrow$ M(2)=(1,0,0,0,0,1,1,0,0)
Mó=(0,1,1,0,0,0,0,0,0) $\rightarrow$ M(2)=(0,0,0,0,0,0,0,1,1)

The direct modelling of simultaneity of events by TPN's has the effect of increasing the complexity of the basic decision problems in TPN. Problems like the reachability, boundedness, coverability, liveness and persistence problems are undecidable. Even for a persistent TPN these first four problems are also undecidable. These results can be derived by proofs similar to that of simulation of any 2-counters automata by TPN.

As it was pointed out, TPN's are "Turing-complete" (in the sense defined in [1]). Previous Petri nets extensions, such as Extended Petri nets ([1]), Priority Petri nets, C-CPM model and EC-CPM models ([5]) are also Turing-complete. However none of these petri net extensions can model all the characteristics of parallelism and synchronization, namely, simultaneity, reentrancy and priority.

For example, TPN models simultaneity of events, but cannot represent reentrancy and recursivity, which need colored tokens (or distinguishable tokens), as shown by Zervos ([5]). On the other hand, the Petri net extensions cited above cannot directly represent simultaneity of events, since they only represent interleaving of events. Thus we can conclude that "Turing-completeness" does not express the fact that a given model is complete in the sense that it can represent all the characteristics of parallelism and synchronization. So far parallelism and synchronization, in all their distinct aspects, have defied precise modelling. More understanding of the fundamental properties of parallelism and synchronization has to be developed, before we are able to perfectly characterize "model completeness".

REFERENCES

[1] Agerwala, T.K.M., "Towards a Theory for the Analysis and Synthesis of Systems Exhibiting Concurrency", PhD. Thesis, John Hopkins Univ., Baltimore, Maryland, 1976.

[2] Irani,K. and C. Zervos, "Modelling of conflicts, priority hierarchies and reentrancy in concurrent synchronization structures", Proceedings of 1979 International Conf. on Parallel Processing, pp. 196-204.

[3] Keller, R.M., "Vector Replacement Systems: A formalism for Modelling Asynchronous Systems", Tech. Report 117, Computer Systems Lab., Princeton University, Princeton, 1972.

[4] Miller, R.E. and Yap, C.K., "On Formulating Simultaneity for Studying Parallelism and Synchronization", to appear in Journal of Computer and System Sciences.

[5] Zervos, C.R., "Colored Petri-Nets: Their Properties and Applications" Ph.D. Thesis, CICE Program, Univ. of Michigan, Ann Arbor, Michigan, January 1977.

PARALLEL COMPUTER ARCHITECTURE EMPLOYING FUNCTIONAL PROGRAMMING SYSTEMS

John C. Peterson
Denver Research Institute
University of Denver
Denver, CO 80208

and

William D. Murray
University of Colorado
Denver, CO 80202

Summary

A new approach to computer architecture is suggested by functional programming (FP) systems (see Backus [1]). An FP system provides a machine language with no variable names, free of side effects, executable in a parallel manner using data flow techniques, easily translated to from procedural languages, and straightforward to implement.

In an FP system, objects represent all data and data structures. These objects can represent any data type. FP functions can be designed for the functions or operations of any language. Functional forms (functions which use other functions as parameters) can model any control flow (procedural) aspect of a language.

Like other data flow systems [2,5] the FP-based machine obtains its parallelism directly from natural data dependencies among operations in a program.

Five items describe an FP system: a set of objects, a set of primitive functions, a set of functional forms, a set of function definitions called D, and the operation of application. An object is either an atom, a sequence of objects, or $\perp$ ("bottom" or "undefined"). Atoms include numbers and identifiers. The special atom ϕ denotes the empty sequence. Sequences are represented by enclosing the sequence elements in < and >. In the FP system of the authors [3] the incomplete object is introduced. An incomplete object contains portions which have yet to be determined, but will be filled in later if needed. These objects will be used to represent the partial result of a function which has not yet completed its execution. Incomplete objects are expressed with the incomplete atom ω, the fundamental unit of incompleteness, capable of assuming any value on completion. An ω can be viewed as a placeholder, representing the result of an arbitrary function which has not yet completed execution. An ω resembles a suspension [4], except that the function associated with the ω is active instead of suspended.

Each ω is associated with a completion function which will eventually specify a value to be used in place of the ω. Many references to a single ω can exist and replacing an ω with a value may alter many objects.

When an ω is a sequence for an append function, a new incomplete object, Ω, is created. Ω is the arbitrary subsequence and indicates a section of an arbitrary length sequence which has not yet been filled in. Ω is like a suspended CDR except that it can occur anywhere in a sequence.

Conceptually, an incomplete object represents a set of objects containing all possible values the object may assume on completion. A partial ordering of incomplete objects can be constructed using the containment operation on sets. An incomplete object, X, is more complete than another Y, if the set of objects associate with X is a proper subset of the set associated with Y.

Arguments require no names since all functions have only one argument. Because programs are composed only of functions, variable names are eliminated. Functions which would normally require more than one argument are applied to a sequence containing all arguments needed. Examples of primitive functions are:

apply:<f,x>	Apply function f to object x.
n:x	The nth element of seq. x.
tl:x	Remove first element of seq. x.
id:x	Identity. Return x unchanged.
eq:<x,y>	Test if x & y are equal objects.
reverse:x	Reverse elements of seq. x.
distr:<s,x>	Seq. pairs of elements of s,x.
length:x	The length of a sequence.
+:<x,y>	Add x and y.
apndl:<x,seq>	Append x to the left of seq.

A $\perp$ is produced when a function is applied to an improperly formed object. All functions (but not necessarily forms) are $\perp$ preserving, returning $\perp$ when applied to $\perp$.

Functional forms use other functions creating expressions involving functions. The principle functional forms are:

f∘g:x	Composition returns f:(g:x).
$[f_1,\ldots,f_n]$:x	Construction, seq. f_1:x,f_2:x...
(p→f;g):x	If p:x is T, f:x, else g:x.
/f:x	Insertion of f into seq. x.
αf:<x_1,x_n>	Apply f to all elements of **x**.

A computer based on an FP system will have three basic components: a set of processors, a memory, and a READY list. The processors apply functions to objects, the memory holds these objects, and the READY list (which may reside in memory) holds functions waiting to be executed.

A list element (instruction) contains:
<function, object, ω_r, D>.
The function and object describe an application, ω_r (ω_{result}) indicates the atom being completed (a function awaiting completion of this instruction), and D defines the program being executed. All instructions of a particular program will

CH1569-3/80/0000-0145$00.75 © 1980 IEEE

have the same D.

The processors execute elements from the READY list with three possible results: if the ω_r is not referenced, the instruction can be discarded; the object may be insufficiently complete for function execution and the instruction is attached to the incomplete atom blocking its execution; the instruction can be executed, the result is installed into ω_r and all functions awaiting completion of ω_r are added to the READY list.

A processor need not be able to execute all functions, but can be specialized for groups of functions in the READY list. All intercommunication among processors is through memory and the READY list. Processors have no state saved between instructions.

Memory contains only objects, which include list elements, D's, functions, and incomplete atoms. Memory must be managed, allowing new objects to be created and removing objects which have become garbage. Garbage must be identified immediately since processors need to know which ω_r's are unused.

In the FP computer incomplete objects control execution and thus introduce parallelism. Two principles govern incomplete objects: all functions are completion functions associated with an ω and the function apply will create incomplete atoms. Thus forms defined in terms of function application generate new ω's. For example, $f \circ g$:x expands to f:(g:x), so that a new atom is created to hold the result of g:x.

Different functions require arguments of different degrees of completeness. Those which manipulate atoms, like + or −, require a complete object. Functions which work with the structure of objects often can be executed with an incomplete operand. (Length:$<\omega_1,\omega_2>$ can be computed without values for ω_1 or ω_2 and 1:$<\omega_1,X>$ evaluates to ω_1.) In postponing the completion of a sequence, the $\perp$ preserving nature of the sequence constructor has been lost and it is natural for an FP system which uses incomplete objects to have a sequence constructor which is not $\perp$ preserving.

The basic forms to be implemented include composition, construction, apply-to-all, condition and insertion. Composition uses an incomplete atom to link the functions being composed. When $<f \circ g,x,\omega_r,D>$ is executed, a new incomplete atom, ω_t, is created. The function g is started by placing the instruction $<g,x,\omega_t,D>$ in the READY list. The function f, represented by $<f,\omega_t,\omega_r,D>$ is attached to ω_t. As soon as g puts a result into ω_t, the function f will attempt to proceed. When ω_t is replaced by an incomplete object, the execution of f and g will overlap if f is able to proceed. This will generally be the case when f and g are highly composite functions.

The construction and apply-to-all forms are similar in that each creates a sequence. Construction applies a variety of functions to the same object, while apply-to-all applies the same function to a variety of objects. In either case function evaluations can proceed in parallel due to the absence of side effects. Since construction brings together multiple arguments for a function parallel argument evaluation results. When $<[f_1,f_2,\ldots,f_n],x,\omega_r,D>$ is executed, $<\omega_1,\omega_2,\ldots,\omega_n>$ is formed and installed into ω_r. Also each $<f_i,x,\omega_i,D>$ is placed on the READY list. The apply-to-all form is similar except that the function will be the same and the argument will differ for each new READY list element.

A special case insertion can be executed in a parallel manner. Insertion computes a result by absorbing each element of a sequence into a dyadic function. With associative functions (which can be recognized before execution) an insert-associative form is used. When the form $</f,<x_1,x_2,\ldots,x_n>,\omega_r,D>$, is executed instructions are created to cause execution to proceed through a binary tree. A function to obtain this tree organization can be defined using parallel construction.

The conditional form is of special interest in parallel processing. To obtain maximum parallelism, p:x, f:x, and g:x would be evaluated in parallel when $(p \rightarrow f;g)$ is executed. The problem is that once p:x is evaluated, either f:x or g:x must be discarded. When f or g are iterative functions (or especially recursive functions), it is best to wait for completion of p:x before starting f:x or g:x.

For the non-parallel conditional, a new form, choose, is introduced. For $<(p \rightarrow f;g),x,\omega_r,D>$ the atom, ω_t is created and $<p,x,\omega_t,D>$ is placed on the READY list. $<$(choose f g x)$,\omega_t,\omega_r,D>$ will be attached to ω_t. When p:x is completed choose will be activated to select f:x or g:x.

References

[1] Backus,J.W.,"Can Programming be Liberated From the vonNeuman Style?,A Functional Style and Its Algebra of Programs,"Comm ACM,Aug.'78.

[2] Misunas,D.P.,"Report on the Second Workshop on Data Flow Computer and Program Organization," Report #MIT/LCS/TM-136,MIT Lab for Computer Science,June '79.

[3] Peterson,J.C. & W.D.Murray, "Parallel Computer Architecture Employing Functional Programming Systems,"Proc.Int'l.Workshop on High-Level Lang.Comp.Archit.,May '80,190-195.(& Peterson, M.S. Thesis, U.Colo. at Denver.)

[4] Friedman,D.P., & D.S.Wise,"Aspects of Applicative Programming for Parallel Processing," IEEE Trans.Computers, Apr.'78, 289-296.

[5] Rumbaug,J.E.,"A Data Flow Multiprocessor," IEEE Trans. Computers, Feb.'77, 138-146.

The Requirements of a Language for Asynchronous Parallel Image Processing

Robert J. Douglass
Department of Applied Mathematics
and Computer Science
University of Virginia
Thornton Hall
Charlottesville, VA 22901

Summary

Image processing has long been considered an important application area for parallel processing because of the large amounts of data involved and because the same operations are performed on every part of an image. Although both parallel architectures and programming languages have been developed for image processing [1], they have been of the SIMD array variety. A large class of image processing algorithms, however, does not fit into an SIMD array format. These algorithms are highly parallel but asynchronous, and they have very different characteristics than the low level filtering, smoothing, and gradient operations that are performed on SIMD machines. This paper lays out the requirements for a high-level language for asynchronous parallel image processing. This work is part of a broader study of parallel language design being undertaken by the Pisces Project on Parallel and Distributed Processing at the University of Virginia.

In contrast to low level image processing algorithms which can be expressed as parallel operations on every point of a two-dimensional array representing an image, higher level image processing uses a more abstract description of the image usually in terms of edges or regions (areas of approximately uniform color and texture). These descriptions can be thought of as a graph where the nodes of the graph represent regions or edges and where the links between the nodes represent the connections between neighboring regions or edges in the image. There is a large class of asynchronous parallel algorithms which process such image graphs by assigning an identical process to each node of the graph. The process updates that node's description using information from the descriptions of neighboring nodes. Relaxation labeling and region matching for change detection are two examples of this class of algorithms.

These algorithms pose particularly interesting problems for the designer of parallel languages since they demand a degree of process interaction that is intermediate between SIMD array languages such as ACTUS [2] or PASCALPL [3] and distributed processing languages such as CONCURRENT PASCAL [4] or Hoare's communicating sequential processes [5]. Their characteristics can be summarized as follows:

1) Division of identical or similar processes to work on different parts of a large common data structure such as an image graph. This type of parallelism is a common feature of SIMD array languages but quite different from concurrent processing languages.

2) Parallelism is at the level of procedures rather than individual operations as in the SIMD array languages.

3) Dynamic creation and destruction of processes and their interconnections. Unlike the static monitors of CONCURRENT PASCAL or MODULA, parallel processes must be created as new nodes in the image graph are created and terminated as nodes are removed. Unlike the fixed 4 or 8 connected configurations of SIMD arrays, asynchronous parallel image processes must be allowed to communicate with an arbitrary configuration of connected nodes in the image graph.

4) Closely coupled processing: since a process on one node must frequently access the information describing neighboring nodes and since the processes are working on parts of one large common data structure, processor communication is best supported by using shared data rather than passing messages.

5) Multiple simultaneous reads of shared data. If all processes sharing the information in a node's description were forced to access it in a strictly sequential fashion as in CONCURRENT PASCAL, then much of the advantage of having parallel processes on each node of the image graph would be lost.

6) Sequential writing of shared data. If several processes are permitted to modify the values of shared data at the same time, the results become dependent on the speeds of the particular processes involved and thus are no longer deterministic. A process which is modifying shared data must be able to lock out all other processes from either writing or reading the data. This problem is the familiar readers and writers problem which has many solutions, but it is so central to the parallelism in image processing that it needs to be solved by the language designer not by the applications programmer as in [6].

The design of a language for asynchronous parallel processing must satisfy the six criteria above. We are exploring an extension of the distributed process concept developed by Hoare [5] and Brinch Hansen [6]. An image graph program is defined by specifying the data structure of a prototype node and the procedures which can process that node. As nodes of an image graph are created, their corresponding procedures are activated to run concurrently with the processes on other nodes. A process running on one node can read the data on any neighboring node but it must call a procedure in the neighboring node to modify the neighbor's data. Synchronization of reads and writes is transparent to the programmer.

An entire image graph is processed by specifying a sequence of actions to be applied to all nodes in parallel in a manner reminiscent of the single instruction stream of an SIMD array machine. The actions on a node, however, are independent processes which resemble a distributed processing network.

An architecture to support the language must be a multiple instruction multiple data stream machine which can dynamically establish communication between different processing nodes. Multimicro-processors with general interconnection networks such as PASM, TRAC, and CM* appear to be good candidates. We feel that new applications in image processing will be opened up using the language to program such asynchronous parallel processors.

References

[1] M. Duff, ed., Proceedings of the Workshop on High-Level Languages for Image Processing, Windsor, England (June, 1979).

[2] R. Perrott and D. Stevenson, "ACTUS - a language for SIMD architectures," Proceedings of the 1978 LASL Workshop on Vector and Parallel Processors, Los Alamos (1978).

[3] L. Uhr, A Language for Parallel Processing of Arrays, Embedded in PASCAL, Univ. of Wis. Computer Science Technical Rpt. #365 (September, 1979).

[4] C. Hoare, "Communicating Sequential Processes," CACM (August, 1978), pp. 666-677.

[5] P. Brinch Hansen, "The Programming Language Concurrent Pascal," IEEE Trans. on Soft. Eng. (June, 1975), pp. 199-206.

[6] P. Brinch Hansen, "Distributed Processes: A Concurrent Programming Concept," CACM (November, 1978), pp. 934-940.

A FASTBUS SYSTEM DESCRIPTION LANGUAGE

T. Christopher, O. El-Dessouki, M. Evens, W. Kabat, S. Wagle

Computer Science Department
Illinois Institute of Technology
Chicago, IL 60616

Summary

Fastbus, the new bus standard which has been jointly developed by the U.S. high energy physics laboratories, presents serious software problems. The first requirement for building and maintaining a large Fastbus data acquisition networks is the development of language facilities for describing the hardware and software architecture of an arbitrary Fastbus system. This paper proposes two levels of network description language: a hardware Network Building Language (H)NBL and a software language (S)NBL.

The current status of the Fastbus standard is well documented in [1]. It is a segmented bus and combines the high local bandwidth attainable at a segment level with global communication via inter-segment connections (SI's). Systemwide addressability is supported and the hardware assumes the responsibility of routing the data to the correct segment and module, be it on the local segement or on a distant segment.

Figure 1 shows a topology typical of those employed in the collection and analysis of data emanating from a particle physics experiment. A bank of data collection computers are employed to collect the massive amount of data coming from the sensors. It is then passed through a filtering network to a host computer where it is recorded and reviewed by the physicists. Besides reducing the data volume, the filtering computers also perform detection of the particle-collision events that are of prime interest in these experiments.

In some ways Fastbus is similar to the architecture of the Cm* system being developed at CMU [3]. Like Cm*, Fastbus can eliminate the need for explicit protocol-based communication among the computers in a local computing network (LCN).

The need for a network description language arises from a desire to develop software that can be easily adapted to a range of system topologies. At the system software level functions that will have to be provided include system initialization, setting up paths between every pair of segments, assisting in the flow of data within the network, loading of software in different computers, and monitoring the status of the network. The applications software, on the other hand, will be written as a collection of parallel activities that can be carried out simultaneously on different computers in the network. In both cases, the software can automatically adapt itself to a particular environment if a description of the current topology is also available on the system in some suitable form. To enter this information in a structured and verifiable manner a network description language is needed. We assume that a typical Fastbus system will have a large number of segments, say about 100.

Requirements of the language:
Such a language, an NBL, could either be procedural and algorithmic or it could be purely declarative. We restrict ourselves to the former variety. For the non-programmers such as hardware engineers we expect to provide an interactive utility through which most of these functions can be carried out; in addition, the utility could provide display of the network in graphic form.

NBL should be capable of describing hardware as well as software resources and current status of all the resources. The hardware resources include network components such as segments, segment interconnects, processors, memories and devices. At the hardware level, NBL should allow specification of how segments are connected and what modules are attached to each segment. To assist the hardware engineers as well as to enable software

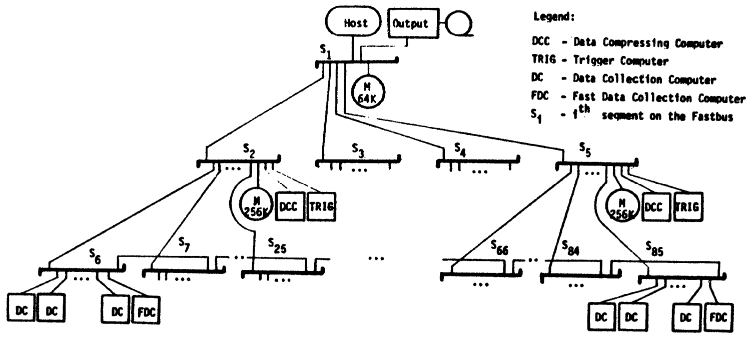

Host
Output
Legend:
DCC - Data Compressing Computer
TRIG - Trigger Computer
DC - Data Collection Computer
FDC - Fast Data Collection Computer
S_i - i^{th} segment on the Fastbus
S_1
M 64K
S_2
S_3
S_4
S_5
M 256K
DCC
TRIG
S_6
S_7
S_{25}
S_{66}
S_{84}
S_{85}
DC
FDC

Figure 1. Model Fastbus tree structure.

CH1569-3/80/0000-0149$00.75 © 1980 IEEE

verification, NBL should also admit the physical description of the network in terms of crates and occupation of slots by hardware modules. The software facilities of NBL should minimally allow one to specify loading of different software modules into different parts of the network and establish data paths among them. At the software level NBL could be thought of as a job control language for a distributed system.

Hardware language:
The example hardware program shows a description of the system pictured below:

```
 10 SEGMENT TOP
 20  FOR I = 1 TO 4
 30   SLOT I HAS TOPSI(I) : SI
 40  NEXT I
 50  SLOT 5 HAS HOST : PDP-11
 60  SLOT 6 HAS OUTPUT : PDP-11
 70  SLOT 7 HAS OUTPUTBUFS : MEM 64K
 80 END SEGMENT TOP
 90 FOR I = 1 TO 4
100  SEGMENT INTERMED(I)
110   SLOT 1 HAS TOPSI(I) : SI
120   FOR J = 1 TO 20
130    SLOT J+2 HAS MEDSI(I,J) : SI
140    SEGMENT LEAF(I,J)
150     SLOT 1 HAS MEDSI(I,J) : SI
160     FOR K = 1 TO 20
170      SLOT K+1 HAS DC(I,J,K) : STP
180     NEXT K
190     SLOT 22 HAS FDC(I,J) : STP
200     IF I <= 4 OR J <= 20
210      SLOT 24 HAS LEAFSI(I,J) : SI
220     ENDIF
230     IF J > 1
240      SLOT 23 HAS LEAFSI(I,J-1) : SI
250     ELSE IF I > 1
260      SLOT 23 HAS LEAFSI(I-1,20) : SI
270     ENDIF
280    END SEGMENT LEAF(I,J)
290   NEXT J
300   SLOT 23 HAS INTERMEDBUFS(I) : MEM 256K
310   SLOT 24 HAS DCC(I) : VAX
320   SLOT 25 HAS TRIG(I) : VAX
330  END SEGMENT INTERMED(I)
340 NEXT I
350 REMOVE LEAFSI(4,2)
360 REMOVE DC(2,20,9)
370 END
```

There are three types of segments. There is one TOP segment with segment interconnects at geographic addresses one through four, a host computer in position 5, an output computer in position 6, and a memory module in position 7. There are four INTERMED segments with a SI to the TOP segment in positions one, SIs to LEAF segments in positions 3 through 22, a memory in position 23, and two computers in positions 24 and 25. There are 80 LEAF segments with a SI to an INTERMED segment in position one, specialized data collection computers in positions 2 through 22, and SIs to preceeding and succeeding LEAF segments in positions 23 and 24. Two devices are presumably out of comission since they are REMOVEd at the end of the specification.

Software language:
The software description language must be able to load the devices mentioned in the hardware description language program, parameterize the various instances of the same program by initializing global variables - especially with addresses of other hardware or software objects on the system, allocate and initialize some standard software objects - such as buffers, and make the initialization conditional upon the existence of neighbouring devices (so that the system can cope with not all devices working all the time).

The example software program fragment given below is to load a data collection program into the hardware system described above. Each DC computer is loaded with a COLLECT program, is given a buffer for its data, and is given the address of its preceeding and succeeding DC computers (so the neighbours can be told to read out sensors near "hits").

The present work is only a part of a much larger network software project under way at Illinois Institute of Technology with generous technical and financial assistance from Fermilab. The hardware version of NBL is being implemented; the translator generates a data base for the network which is designed to support initial program load, user program design and deployment, system maintenance, fault detection, and reconfiguration [4].

```
 10 LET FDCBUFSIZE=600
 20 LET DCBUFSIZE=600
 30 LET OUTBUFSIZE=16380
 40 FOR I = 1 TO 4
 50  FOR J=1 TO 20
 60   LOAD FDC(I,J)
 70    LO FASTCOLLECT
 80    INIT PARENT = FDCBUF(J) IN INTERMEDBUFS(I)
 90    INIT PARENTSIZE = FDCBUFSIZE
100   END LOAD FDC(I,J)
110  NEXT J
120 NEXT I
130 FOR I = 1 TO 4
140  FOR J = 1 TO 20
150   FOR K = 1 TO 20
160    LOAD DC(I,J,K)
170     LO COLLECT
180     INIT PARENT = DCBUF(J,K) IN INTERMEDBUFS(I)
190     INIT PARENTSIZE = DCBUFSIZE
200     LET K1 = K + 1
210     LET J1 = J
220     LET I1 = I
230     IF K1 > 20
240      LET K1 = 1
250      LET J1 = J1 + 1
260     END IF
270     IF J1 > 20
280      LET J1 = 1
290      LET I1 = I1 + 1
300     END IF
310     IF DC(I1,J1,K1) EXISTS
320      INIT SUCC = CSR_PREDINPUT IN DC(I1,J1,K1)
330      INIT SUCCDEFD = 1
340     ELSE
350      INIT SUCC = NULL
360      INIT SUCCDEFD = 0
370     ENDIF
 .
 .  similarly for predecessors
 .
560    END LOAD DC(I,J,K)
570   NEXT K
580  NEXT J
590  LOAD DCC(I)
600   LO DSQUEEZ
610   INIT DCBUFS = DCBUF IN INTERMEDBUFS(I)
620   INIT DCBUFDELEMSZ=DCBUFSIZE
630   INIT DCBUFSDIM1=20
640   INIT DCBUFSDIM2=20
650   INIT PARENT = BUF(I) IN OUTPUTBUFS
660   INIT PARENTSIZE=OUTBUFSIZE
670  END LOAD
680  LOAD TRIG(I)
690   LO TRIGGER
700   INIT TRIGDATA = FDCBUF IN INTERMEDBUFS(I)
710   INIT TRIGELEMSIZE=FDCBUFSIZE
720   INIT TRIGDATADIM=20
730   IF TRIG(I+1) EXISTS
740    INIT NEXTTRIG=CSR_PREDINPUT IN TRIG(I+1)
750    INIT NEXTRIGEXISTS=1
760   ELSE
770    INIT NEXTTRIG=NULL
780    INIT NEXTTRIGEXISTS=0
790   ENDIF
 .
 .  similarly for TRIG(I-1)
 .
870  END LOAD
880  LOAD INTERMEDBUFS(I)
890   ALLOC DCBUF(20,20) : BUFFER(DCBUFSIZE)
900   ALLOC FDCBUF(20) : BUFFER(FDCBUFSIZE)
910  END LOAD
920 NEXT I
930 LOAD OUTPUTBUFS
940  ALLOC BUF(4) : BUFFER(OUTBUFSIZE)
950 END LOAD
960 END
```

References

1. US NIM Committee, "Fastbus, Modular High Speed Acquisition System for High Energy Physics and other applications", January 1980.

2. R. J. Swan, et al., "Cm* - A modular, multi-microprocessor", AFIPS Conference Proceedings, Vol.46, 1977 NCC, pp. 637-644.

3. M. A. Mayor, "A Language for Network Analysis and Definition", SIGPLAN Notices, Vol.15, No. 1, January 1980, pp.130-138.

4. T. Christopher, et al., "A Network Description Language for Fastbus Systems", to appear in COMPCON Fall'80 proceedings.

VSP: BUILDING BLOCKS FOR PARALLEL PROCESSORS

William S. Dowey
Gould Inc., Chesapeake Instrument Division
6711 Baymeadow Drive, Glen Burnie, MD 21061

This paper addresses the Vector Scalar Processor (VSP) solution to the problem of proliferation of high cost specialized processors for computationally large algorithms. As the title implies, the processors can be configured in a variety of distributed parallel processor formats. The Vector Processor (VP) is the title given to the processor block responsible for repeated sum of product operations. The Scalar Processor (SP) is the title given to the processor block responsible for communications, scheduling, and data storing/retrieval. This VSP solution is unique in that the hardware (based on the AM 2900 family) in the blocks is interchangeable within and between VP and SP processors. This interchangeability trades off design complexity for multiple low cost processor blocks which achieve computational requirements.

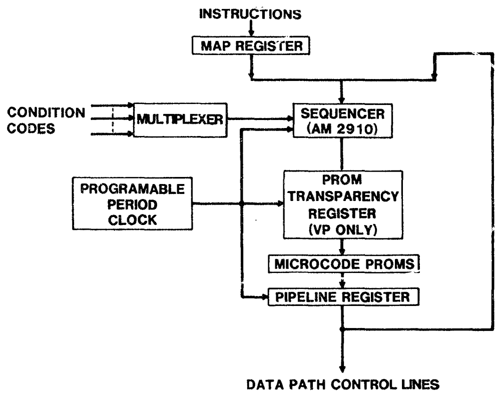


Figure 1. Controller for Vector and Scalar Processors

Both the SP and the VP use a similar controller configuration centered around an AM 2910 sequencer (Figure 1). This sequencer has four modes of control for microcode address selection. An external condition code is used primarily for the data dependent operations of the SP; the Internal Counter equal to zero is used for the structure (algorithm) dependent VP operations. The controller block is completed using a horizontal field pipeline register to allow for simultaneous action of all functional blocks in the data paths. The pipeline ROM which contains the microcode instructions removes the combinational logic from the design process. This logic is replaced with programmable fields which determine the state of the functional blocks for each clock period. Both SPs and VPs have individual clock units, each VP clock being slave to an SP clock. Both clocks are capable of outputing instruction selectable periods (from 60-480 nano seconds in 30 nanosecond increments). This allows matching propagation delay paths to execution times instead of restricting execution times to the longest propagation paths for all instructions.

The Scalar Processor shown in Figure 2 features a Von Neumann type machine modified to incorporate AM 2901 based next address generator (NAG) for retrieval of instructions and data from a common (macro) memory. With the horizontal microcode, ALU operations can be taking place on data while the next address generator is fetching data or the next instruction word. The SP supports both the AN-UYK-20 assembly language instruction and special microcode instructions (beyond the standard instruction set) to achieve operational speed increases for a Direct Memory Access (DMA) data handling operations. The Scalar Processor communicates

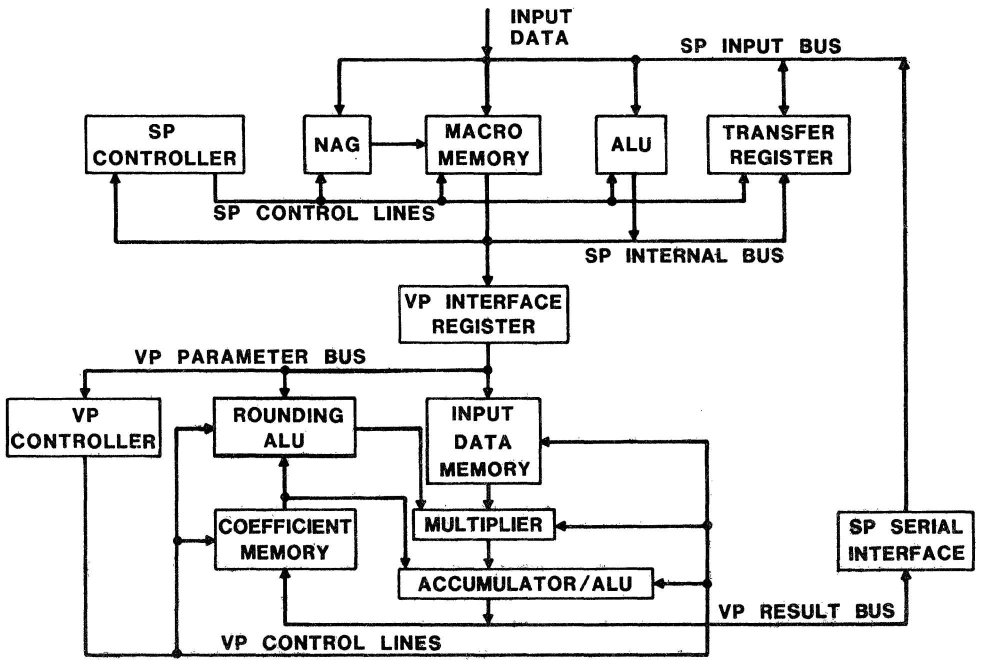


Figure 2. Functional Blocks of the Vector and Scalar Processor

with multiple VPs through a VP data bus and receives status from the VPs in a serial fashion. Communications with other SPs also takes place over other serial lines.

The VP shown in block form in Figure 2 features a controller which performs repeated operations on data with a taper-on and taper-off of data flow required from the data memories. Arranged this way, the VP gains efficiency as the number of identical operations exceeds 8. The Multiplier is interchangeable with a multiply accumulate, allowing for faster execution of sum of product algorithms. If the complexity or throughput requirements of the algorithm do not allow for in-place computations, SPs are used to collect partial processed data, reorder, and pass it on to the next level of VP processing.

The three VSP configurations below illustrate the modularity of the building blocks. The first configuration was the first pilot VSP development. It implements an FIR Interpolation filter of 1 million operations per second (MOPS) per VP. This figure is exclusive of VP overhead operations. This filter configuration, shown in Figure 3, is a computationally distributed network. It features two parallel paths comprised of two SP and VP elements. Here both halves of the parallel network contain duplicate microcode, so that each side is capable of processing either half of the incoming data. The second SP in each half parallel leg collects results and performs half aperture broadside beamforming on the VP result data. While it would be possible to link the VP parallel branches in a Single Instruction Multiple Data (SIMD) fashion, the necessity of half aperture operation overruled this approach. Similarly, the two legs of each VP could be combined under one controller, but the interchange of data at the SP was initially thought to preclude this.

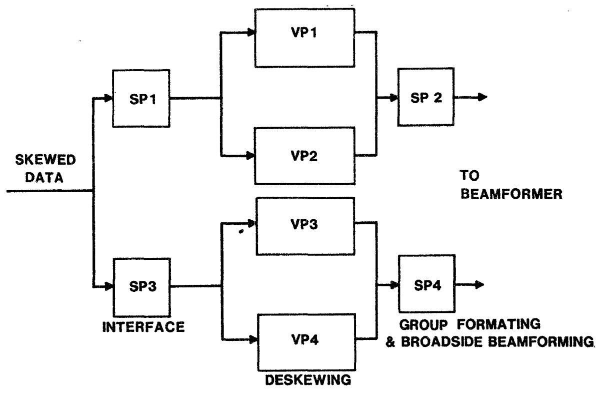


Figure 3. Interpolation Filter of 4 MOPS in 2 Parallel Paths

A Widrow Filter application shown in Figure 4 is in development. It utilizes one SP and 8 parallel VPs composed of 2 parallel arithmetic elements (AEs). The VPs, are under the control of 1 sequencer, in SIMD fashion. Each AE calculates results and interchanges these results with its paired AE, checking for computational inaccuracies. In executing the Widrow filter, there are 5.2 MOPS/AE, which yield a total of 84 MOPS of SIMD instruction. In this configuration, the 24 bit coefficients are updated each cycle and rounded to 12 bits for use in the filter applications.

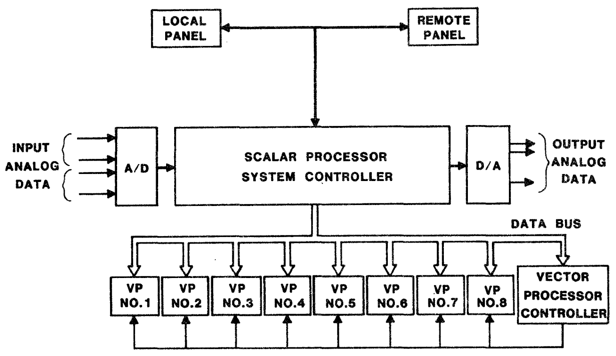


Figure 4. SIMD Control Achieves 84 MOPS in 8 VP Elements

The geometric processor is an embed processor in a simulator for image processing calculations. The computation rate is 9 MOPS for 6000 edges. It is configured as shown in Figure 5, with VP_2 and VP_3 sharing a common controller and executing in a SIMD fashion.

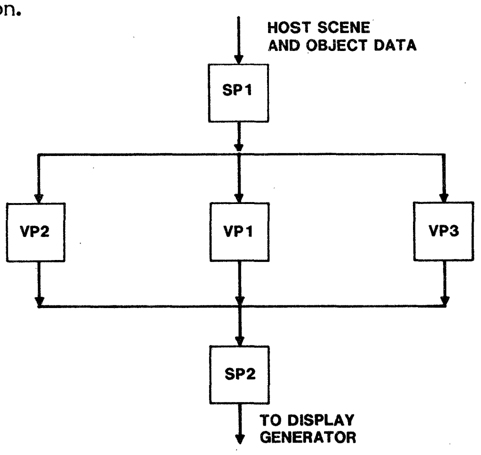


Figure 5. VP_2, VP_3 are SIMD Units for 80% of Computational Time.

The Filter configurations, shown above, are in the process of being built with Standard Electronic Modules (SEM) of the U.S. Navy. All the control sequencer block is available in this format. All but one of the data path cards (the AM 2903 dual port accumulator/ALU) are available as standard modules. The GP is being developed on 5 card types, from which both SP and VP modules can be configured. The five distinct card types (Controller, ROM, RAM, ALU, Multiplier) are each based on the AM 2900 family components.

The Building Blocks processor approach to distributed parallel processing is a viable concept, as the variety of applications cited here demonstrate. Each of the applications uses different algorithms, yet the same approach of stacking the processors has been used to achieve the desired computational throughput from the basic 6 MOPS for the VP to the 84 MOPS for the 8 parallel dual VPs in the Widrow application. The VSP architecture allows simplicity and reliability of design -a cardinal trait of effective building blocks.

A NEW GENERAL-PURPOSE DISTRIBUTED MULTIPROCESSOR SYSTEM STRUCTURE

Jin Lan
Department of Computer Engineering and Science
Tsinghua University
Peking, The People's Republic of China

Summary

One of the basic problems of organizing a multiprocessor is to develop a good system structure, which, from the author's point of view, should be modular, reconfigurable, partitionable as well as tightly-coupled. These properties are necessary for forming a general-purpose system, in which multitasking and multiprogramming can be combined together to enhance the overall system efficiency. As a possible solution of this problem, a distributed multiprocessor system structure was proposed in this paper.

It is noted that among the great varieties of parallel and multiprocessor structures bus [1]-[4] and array [5]-[8] are the widely-accepted approaches to solving interconnection problem in many operational systems. The proposed scheme tends to combine these two structures to form a new one, taking advantages of simplicity and possibility of using commercially available models of processors from the side of bus, and regularity and adaptivity to large-scale systems and high processing power from the side of array. But the new structure has its own special features, not belonging to its "predecessors". Its main difference from the bus structure is the concept of "splitted bus", realized by introducing switches for routing control and elimination of bus contention. The basic structure resembles a mesh connection, but it is more flexible and has shorter message paths than the simple array. These considerations lead to the structure shown in Fig.1 for n*n processors. The circles denote the three-pole bidirectional switches, and the solid dots denote the processor-nodes. This structure can be redrawn symbolically in Fig.2, where lines, crosspoints and shaded triangles represent bus-segments, processor-nodes and switches respectively. Each switch has three poles connected to three processors at its corners. This notation helps in revealing some advantages of the structure owing to the existence of connecting paths along the diagonal directions of the array. This fact causes a significant reduction of message transfer distance between processors, maximum length of which for an n*n array equals to

$$d_{max} = \frac{2n - 2n(\text{modulo } 3)}{3} .$$

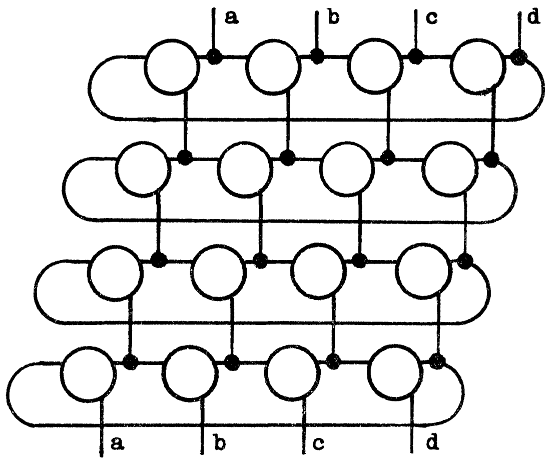


Fig.1 Two-dimensional system structure

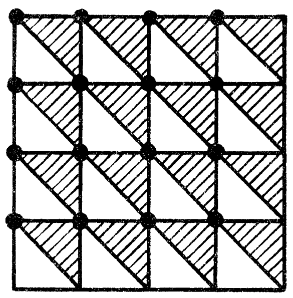


Fig.2 Graphical representation of a planar array structure

It means that n = 2k + 1 for k = 1, 2, ... will be the optimum array size. In comparison with other structures with message distance $O(\log_2 N)$, this structure may take some benefit for moderate systems with total number of processors not exceeding $N = 100$, because in this case $d_{max} = 6$, while $2^6 = 64 < 100$.

Another main characteristic of the proposed structure is the ability of dynamic reconfiguration and programmable partitioning of the system. Two examples are shown in Fig.3, in which the 4*4 array can be transformed into a linear array or ring, or two separate trees with 7 and 9 processors respectively. Another example of transforming a 6*6 array into a 5-level binary tree is shown in Fig.4.

CH1569-3/80/0000-0153$00.75 © 1980 IEEE

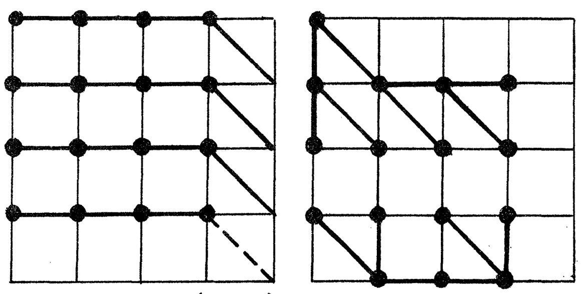


Fig.3 Examples of reconfigured systems

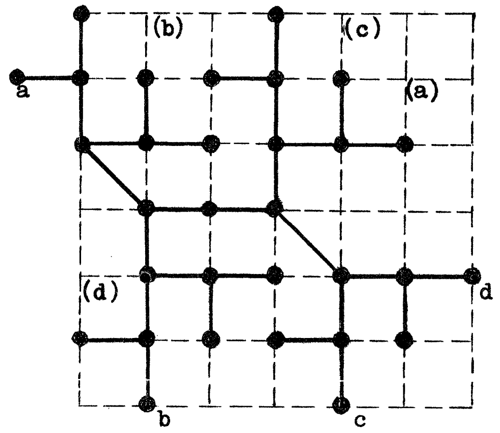


Fig.4 Reconfiguration into a five-level binary tree

A further improvement of the structure can be achieved by using four-pole bidirectional switches with the symbol shown in Fig.5. The number of different states of the switch increases to 15, giving a great flexibility of interconnections. The four poles of it can be imagined to form three planes perpendicular to each other. This provides convenience in organizing a three dimensional cube structure. Every one of the n cross-sections parallel to any surface of this cube contains n*n processors, forming just an array like that shown in Fig.1. This makes the structure useful for organizing an MSIMD system with total number of processors N = n**3, so that the message distance between any two nodes is reduced to $O(\sqrt[3]{N})$, and the optimum size of the system is extended to N = 1000.

Still another way of using the fourth pole of the switch may be to add more message paths to the original array of Fig.1. One of the modified arrays thus obtained is shown in Fig.6. For clarity, only a small part of the additional paths are represented: the 6 paths from one switch (by heavy lines), and the paths connecting one processor to its 12 neighbours.

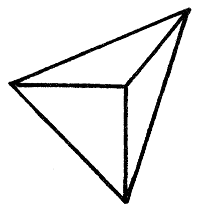


Fig.5 Symbol of a 4-pole bidirectional switch

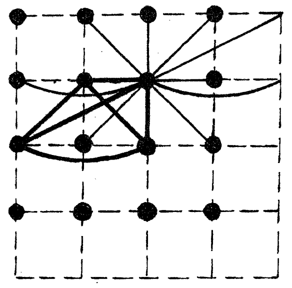


Fig.6 One-to-twelve connections

References

[1] R.Kober,and Ch.Kuznia, "SMS - A Multiprocessor Architecture for High Speed Numerical Calculations", Proc. 1978 Int'l Conf. Parallel Processing (Aug. 1978), pp. 18-24.

[2] E. P. O'Grady, "A Multiprocessor for Continuous System Simulation", Proc. 1979 Int'l Conf. Parallel Processing (Aug. 1979), p. 306.

[3] M. Maekawa, et al, "Experimental Polyprocessor System(EPOS) -Architecture", 6th Annual Symposium Computer Architecture (Apr. 1979), pp. 188-195.

[4] S. Uchida, and T. Higuchi, "A Multi-Mini-Computer System for Picture Processing Experiments and Its Interconnection Mechanism", Proc. 1978 Int'l Conf. Parallel Processing (Aug. 1978), pp. 88-94.

[5] W.J. Bouknight, et al, "The Illiac IV System", Proc. IEEE, vol.60, No.4(Apr. 1972), pp. 369-388.

[6] K. Batcher,"MPP - A Massively Parallel Processor", Proc. 1979 Int'l Conf. Parallel Processing (Aug. 1979), p. 249.

[7] P.M. Flanders, et al, "Efficient High Speed Computing with the Distributed Array Processor", in High Speed Computer and Algorithm Organization,ed. D. J. Kuck, et al, Academic Press,(1977), pp. 113-128.

[8] U. Herzog, et al,"Performance Modeling and Evaluation for Hierarchically Organized Multiprocessor Computer Systems", Proc. 1979 Int'l Conf. Parallel Processing (Aug. 1979), pp. 103-114.

A MULTI-MICROCOMPUTER ARCHITECTURE FOR AN ITERATIVE ALGORITHM

Dan I. Moldovan
Department of Electrical Engineering
Colorado State University
Fort Collins, CO 80523

Summary

This paper analyzes the inherent parallelism in computation of some recursive algorithms. The class of functions considered present at least two levels of parallelism. A multi-computer architecture is proposed for this class of problems. This architecture can be easily implemented with microcomputers, and a high degree of modularity may be achieved. The operation of such architecture, including the computer communication and timing was studied on a simulated model.

Consider the following recursive vector function

$$x(k+1) = f[x(k),x(k-1),\ldots,x(k-m+1)] \quad (1)$$

with $x(k) \in R^n$ and $k \in \{0,1,\ldots,K\}$. The vector $x(k+1)$ depends of its previous m values. The iterative nature of the above expression derives from the fact that the computational procedure repeats when k is incremented. Equation (1) can represent a system of difference equations describing the behavior of some dynamic systems commonly seen in control theory and signal processing. Sometimes, logic equations used in the design of digital systems are put in the form of expression (1).

The first level of parallelism in computing (1) is achieved when all components of vector $x(k+1)$ are computed simultaneously. Next, let us assume that each component can be written as

$$x_i(k+1) = f_i(\phi_{i1},\phi_{i2},\ldots,\phi_{ij},\ldots,\phi_{ir_i}) \quad (2)$$

where $\phi_{ij} = \phi_{ij}[x(k),x(k-1),\ldots,x(k-m+1)]$. Notice that all ϕ_{ij} can be computed independently and simultaneously if they are assigned to different computers, provided that vector x is available. This represents a second level of parallelism which can be exploited. While further levels might be possible, we consider only the first two levels, and this is sufficient of many applications.

Parallel processing is oftenly motivated by the desire to increase the speed of computation. Thus, especially under real-time conditions parallel processing might be the only solution to complex numerical problems. Some recent microprocessors, with their relatively low cost and high computing power open new possibilities to implement powerful multiprocessor systems.

One possible multi-computer architecture for computing x(k+1) is shown in Figure 1.

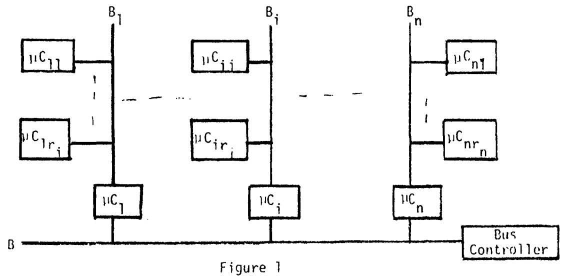


Figure 1

Each function ϕ_{ij} is assigned to one microcomputer μc_{ij}. The functions f_i are computed on microcomputers μc_i. The vertical buses B_i are used to transfer data between μc_i and μc_{ij}, in both directions. The horizontal bus B is used to transfer the newly computed vector components $x_i(k+1)$ from their source to other computers μc_j. Each microcomputer consists of a microprocessor, memory, I/O and control logic. Because of the iterative nature of the problem under consideration, the memory of each computer is relatively small.

Several operations of this computing structure are possible. We choose to operate each computer independently of others and run by different clocks. However, the computer does not start its processing tasks until the required data has arrived. This is considered to be synchronized operation because all necessary computations are performed within one iteration. The synchronized operation is preferred here over asynchronous operation because we want to maintain a high degree of accuracy in computations.

The transmission of data from source to destination is conditioned by the availability of the respective bus and the readiness of the destination. In the model used, a computer cannot be interrupted to receive data while processing.

For our convenience, we partition the activities involved for one iteration in processes. The following types of processes take place for each iteration.

P1. Transfer components of vector x(k) from μc_i to μc_{ij}, as needed.

P2. Compute ϕ_{ij} on μc_{ij}.

CH1569-3/80/0000-0155$00.75 © 1980 IEEE

P3. Transfer the result ϕ_{ij} from μc_{ij} to μc_i.

P4. Compute $x_i(k+1) = f_i$ on μc_i.

P5. Transfer $x_i(k+1)$ from source to other microcomputers on B bus, as needed.

P6. Next iteration, $k \leftarrow k+1$, update variables.

Our goal is to perform simultaneously as many processes as possible.

The study of such architecture was done on a simulated model. The first step is to map the mathematical problem of form (1) into an architecture of the type shown in Figure 1. Since no formal procedure for this mapping was established, and hence is not unique, the aim of the simulation is to estimate the system performances and to identify ways of improving them. We are not interested too much in simulating the execution of a program on microcomputers, instead, we simulate the data flow between computers and the operational strategy. A simulated real-time clock marks the timing events. For each time unit the program scans all the microcomputers, determines their status and initiates or terminates activities. Various processing times and data transfers between computers, dictated by the mathematical problem, are stored in a set of matrices.

The output of the simulation program provides the number of time units required for one iteration, the structure's speedup factor and its efficiency, the utility factors for all microcomputers and the number of bus contentions. An analysis of such output data allows us to "tune" the architecture according with the mathematical problem to achieve the desired performances.

Example: Consider the following problem.

$$\begin{bmatrix} x_1(k+1) \\ x_2(k+1) \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_1^2(k) \\ x_2^2(k) \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \begin{bmatrix} x_1(k)\cdot x_1(k-1) \\ x_2(k)\cdot x_2(k-1) \end{bmatrix} \tag{3}$$

First, partition the problem such that only 6 computers are used. One possible way is:

$$\phi_{11} = a_{11} X_1^2(k) + b_{11} X_1(k) X_1(k-1)$$

$$\phi_{12} = a_{12} X_2^2(k) + b_{12} X_2(k) X_2(k-1)$$

$$\phi_{21} = a_{21} X_1^2(k) + b_{21} X_1(k) X_1(k-1)$$

$$\phi_{22} = a_{22} X_2^2(k) + b_{22} X_2(k) X_2(k-1)$$

$$f_1 = \phi_{11} + \phi_{12} \text{ and } f_2 = \phi_{21} + \phi_{22}$$

Based on this partitioning of the mathematical problem and knowing the characteristics of the microcomputers used, one can estimate the execution time for each function. It is feasible to consider fluctuations in the processing speed of these functions for different iterations. A uniform distribution around a selected mean was assumed in our simulation. For Intel 8086 based microcomputers, it is estimated that $\bar{t}_{ij} = 210$ time units, $\bar{t}_i = 10$ time units and transmission λ between two adjacent computers is 5 time units. One time unit corresponds to approximately 5 clock cycles.

For this input data the simulation program provided the outputs indicated in the first column of Table 1. It can be seen that μc_1 and μc_2 are underutilized when compared with the rest. Next, we want to further decrease the computation time for problem (3) by introducing more computers.

	6 μc	10 μc
Speed up factor	3.72	7.20
Efficiency	0.62	0.72
μc_{11} utility	0.98	0.89
μc_{12} utility	0.97	0.93
μc_{13} utility	Not used	0.97
μc_{14} utility	Not used	0.82
μc_{21} utility	0.95	0.87
μc_{22} utility	0.97	0.93
μc_{23} utility	Not used	0.86
μc_{24} utility	Not used	0.90
μc_1 utility	0.12	0.49
μc_2 utility	0.10	0.46
Iteration time	243 t.u.	125 t.u.

Table 1.

This is done by assigning 4 computers to compute f_1 or f_2 instead of only two as used previously. The results are summarized in the second column of Table 1. Notice that the computation time per iteration is reduced by almost half and the utility factors for μc_1 and μc_2 improved while the others still remained high.

PARALLEL NONLINEAR MINIMIZATION BY CONJUGATE DIRECTIONS

Efthymios C. Housos and Omar Wing
Department of Electrical Engineering
Columbia University
New York, NY 10027

Summary

In the development of parallel algorithms for minimization new requirements, such as minimization of the communication time and the exchange of data among processors, become as important as the classical requirements of good convergence and numerical robustness. In this paper algorithms suitable for the solution of the unconstrained minimization problem on a parallel computer are presented. The algorithms involve the parallel execution of linear searches along conjugate directions. The basic assumptions about the parallel computer are the following:

1) Every processor is able to exchange information with every other processor.
2) The communication time is important and should be minimized.

The algorithms developed are of the conjugate direction type but are different than those reported in [1].

The importance of conjugate directions for the solution of the unconstrained minimization problem has been realized by many researchers [5,6]. The conjugate direction algorithms are based on the conjugacy properties of a set of vectors with respect to a certain matrix. Namely, a set D = $\{\underline{d}_i,\ i=1,\ldots,n\}$ consists of conjugate vectors with respect to a matrix H, if and only if

$$\begin{aligned}(\underline{d}_i,\ H\,\underline{d}_j) &= 0 \qquad \text{for}\quad i \neq j\\ &= 1 \qquad \text{for}\quad i = j\end{aligned}$$

Assuming that there is some way of finding a set of conjugate directions given a matrix H and using theorem 1 below, the solution of the problem

$$\min_{\underline{x}} J(\underline{x}) \qquad \underline{x} \in R^n \qquad (1)$$

where $J(\underline{x})$ is quadratic, could be found in one parallel step involving a linear search along each of the directions.

Theorem 1

The minimum of a quadratic function, $J(\underline{x})=\underline{x}^t H \underline{x} + \underline{b}^t\underline{x} + c$, can be found by searching through a set $D=\{\underline{d}_i, i=1,\ldots,n\}$ of conjugate directions with respect to H once and only once in any order.

This theorem implies that if a set of n conjugate directions with respect to H and a set of n processors were available then the solution of (1) could be found in one major parallel step and an additional step that involves the addition of the local minima. Of course, this is only true if $J(\underline{x})$ is quadratic. If $J(\underline{x})$ is not quadratic the above theorem would suggest finding a set of conjugate vectors with respect to the Hessian of $J(\underline{x})$. Thus the major problem becomes one of finding a set of conjugate vectors in parallel. That is, develop methods of producing conjugate vectors, with respect to a matrix, which are amenable to parallel computation. The difference between the serial and the parallel algorithms comes from the fact that for parallel computation it is necessary to estimate all the conjugate directions before any linear searches are performed. Once this is achieved, all the linear searches may be performed in parallel and, hence, a large part of the total computation time is thus parallelized. This is because a linear search usually involves 3-5 function evaluations which can be time consuming for a fairly complex objective function. The parallel execution of linear search procedures also insures that the utilization of the parallel computer will be high because the communication time (time for the exchange of information among processors) will be a small fraction of the total computation time. It has been shown that for the class of parallel computers such as the SIEMENS SMS 201 the ratio of the communication time to the actual computation time is the most critical factor in achieving a reasonable "speed-up" [2].

The Gram-Schmidt method could be used for finding a set of conjugate vectors with respect to a matrix but this method is both computationally inefficient and not readily parallelizable. For these reasons it would be desirable to have methods of finding conjugate or semi-conjugate directions which are amenable to parallel computation. An algorithm for the solution of (1) based on two theorems proved by M.J.D. Powell [3] will be presented next. Details about the algorithm and computational experience in solving power system problems using this algorithm can be found in [4].

Algorithm

Choose a set of orthogonal vectors as the initial search vectors. Let these vectors be $\underline{d}_i$, i=1,...,n, where n is the dimension of the problem. Choose an initial point $\underline{x}^0$ and calculate $J(\underline{x}^0)$.

Step 1. Find the n one dimensional minima along the n search vectors that is,

$$\min_{\lambda} J(\underline{x}^0 + \lambda\underline{d}_i) = J(\underline{x}^0 + \lambda_i\underline{d}_i) \qquad i = 1,\ldots,n$$

where λ_i is the optimum steplength. Let

$$\underline{x}_i = \underline{x}^0 + \lambda_i \underline{d}_i \qquad (2)$$

STOP if a solution has been found. This step can be implemented in parallel using up to n processors.

Step 2. Set

$$\underline{x}_{n+1} = \underline{x}^0 + \sum_{i=1}^{n} \lambda_i \underline{d}_i \qquad (3)$$

or calculate

$$J(\underline{x}^0 + \alpha^*(\sum_{i=1}^{n} \lambda_i \underline{d}_i)) = \min_{\alpha} J(\underline{x}^0 + \alpha(\sum_{i=1}^{n} \lambda_i \underline{d}_i))$$

and set $\underline{x}_{n+1} = \underline{x}^0 + \alpha^* \sum_{i=1}^{n} \lambda_i \underline{d}_i$

Step 3. Set $\underline{x}^0 = x_i$ such that

$$J(\underline{x}_i) = \min_{k} J(\underline{x}_k) \qquad k = 1,\dots,n+1 \quad (4)$$

Usually $J(\underline{x}_i) = J(\underline{x}_{n+1})$

Step 4. Normalize the current search directions with respect to the Hessian matrix of $J(\underline{x})$. That is, estimate $(\underline{d}_i, H\, \underline{d}_i)$, i=1,...,n, and set

$$\underline{d}_i \leftarrow \frac{\underline{d}_i}{(\underline{d}_i, H\underline{d}_i)^{1/2}} \qquad (5)$$

where H is the Hessian matrix of J.

Step 5. Update the search directions using an orthogonal matrix P as follows:

$$\underline{d}'_i = \sum_{k=1}^{n} p_{ik}\, \underline{d}_k \qquad i=1,\dots,n \qquad (6)$$

That is, every new search direction is a linear combination of all previous search directions.

Step 6. Set $\underline{d}_i \leftarrow \underline{d}'_i \cdot \alpha_i \quad i=1,\dots,n$

where α_i is (with equal probability) either 1 or -1.
GO TO Step 1.

As it can be seen, the algorithm involves primarily the parallel execution of linear searches along semi-conjugate directions. An orthogonal matrix P is used in updating the current set of search directions to another set of directions, which is closer to being conjugate with respect to the Hessian matrix than the original set of search directions. More details about the significance of the orthogonal matrix P and test case results using different orthogonal matrices can be found in [7].

References

[1] E.C. Housos and Omar Wing, "Solution of the Load Flow Problem by a Parallel Optimization Method," 1979 Power Industry Computer Applications Conference, pp. 332-335.

[2] R. Kober, "A fast communication processor for the SMS multiprocessor system," Proc. Second Symposium on Micro Architecture, M. Sami. J. Wilmink, R. Zoks (eds.), EUROMICRO, 1976, pp. 183-189.

[3] M.J.D. Powell, "Unconstrained Minimization Alorithms without Computation of Derivatives," Report T.P. 483, A.E.R.A. Harwell, United Kingdom, April 1972.

[4] E.C. Housos and O. Wing, "Parallel Nonlinear Minimization Methods with Applications to Power System Problems", SIAM Conference Proceedings, "Electric Power Problems: The Mathematical Challenge", Seattle, March 1980.

[5] M.R. Hestenes and E. Stiefel, "Methods of Conjugate Gradients for Solving Linear Systems," Research Journal of the National Bureau of Standards, Vol. 49, pp. 409-436, 1952.

[6] M.J.D. Powell, "An Efficient Method for Finding the Minimum of a Function of Several Variables without Calculating Derivatives," Comput. J., 7 (1964), pp. 155-162.

[7] Efthymios C. Housos, "Parallel Nonlinear Optimization Methods with Applications to Power Systems," Ph.D. Thesis, Columbia University, 1980.

A PARALLEL ALGORITHM FOR SOLVING BAND SYSTEMS OF LINEAR EQUATIONS

Ladislav HALADA
Institute of Technical Cybernetics
Slovak Academy of Sciences
809 31 Bratislava, Czechoslovakia

Summary

In this paper a new parallel direct algorithm for solving band systems of linear equations is discussed. The algorithm is similar to the "shooting method" proposed by Bank and Rose [1] and to the LU decomposition mentioned by Sameh and Kuck [2]. However, the formula from which the algorithm is derived is believed to be new.

Let us consider linear systems of equations $Ax = b$, where A is a regular band matrix of order n with bandwidth $(2m+1)$, i.e. $a_{ij} = 0$ for $|i-j| > m$ and $a_{i,i+m} \neq 0$ for $i = 1,2,\ldots,n-m$. We assume a frequent situation in practise, $m \ll n$.

The algorithm is based on the following assertion [3]. If A is a nonsingular matrix the first m components of the solution vector x satisfy

$$\begin{pmatrix} z^{(1)}_{n-m+1} & z^{(2)}_{n-m+1} & \cdots & z^{(m)}_{n-m+1} \\ z^{(1)}_{n-m+2} & z^{(2)}_{n-m+2} & \cdots & z^{(m)}_{n-m+2} \\ \vdots & \vdots & & \vdots \\ z^{(1)}_{n} & z^{(2)}_{n} & \cdots & z^{(m)}_{n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} = \begin{pmatrix} z^{(0)}_{n-m+1} \\ z^{(0)}_{n-m+2} \\ \vdots \\ z^{(0)}_{n} \end{pmatrix} \qquad (1)$$

where $z^{(i)}_j$ is the j-th unknown of the system

$$T\,z = c_i \qquad i = 0,1,2,\ldots,m. \qquad (2)$$

Here, $c_0 = b$ and c_i, $i = 1,2,\ldots,m$ is the i-th column of A. The matrix T is modified matrix A. It originates from A by omitting columns $c_1, c_2, \ldots, c_m$ and by adjoining columns $-e_{n-m+1}, \ldots, -e_n$ after the last column of A, where e_j is the j-th column of the identity I_n. Thus, T is a lower triangular matrix of order n with bandwidth $(2m+1)$.

If $x_1, x_2, \ldots, x_m$ are known, solving the system

$$T\,y = d \qquad (3)$$

where components of the column vector d are given by $d_i = b_i - \sum_{j=1}^{m} a_{ij}x_j$, $i = 1,2,\ldots,n$, we can obtain other components of x, because $y_i = x_{i+m}$, $i = 1,2,\ldots,n-m$ holds.

Thus, the algorithm consists of the following stages:

Stage 1. The solution of the systems (2). Let us use for solving (2) Algorithm II of [4]. A simultaneous computation of $(m+1)$ triangular systems differing from each other only in the right-hand side by this algorithm requires $\tau^{(1)} = (2+\log 2m)\log n -$

$-(1/2)(\log^2 2m+\log 2m)+3$ time steps using no more than $3m^2n+mn-8m^3$ processors[a].

Stage 2. The computation of the system (1). Solving this dense system of order m using Gaussuan elimination with pivoting requires $\tau^{(2)}=3m(\log m-1)+0(\log^2 m)$ steps using $(m-1)^2$ processors.

Stage 3. The computation of the system (3). Solving (3) by Algorithm II with the computation of d_i requires $\tau^{(3)}=\tau^{(1)}+$ $+\log m+2$ steps using no more than $(1/2)\,m^2n+(1/2)\,mn-m^3$ processors.

Unfortunately, the algorithm fails for the same reason as banded triangular solvers. It suffers from the possibility of over- or underflow. On the other hand, it does not fail if all of the leading principal submatrices are singular and it can be easily modified when the number of available processors is much less than the order of the system, e.g. by using a practical band triangular system solver discessed in [5]. However, we have proved the following theorem.

Theorem. Let A be a regular band matrix of order n with bandwidth $(2m+1)$, where $m \ll n$. Then $Ax=b$ can be solved on SIMD type parallel machine in $(4+2\log 2m)\log n+0(m\log m)$ time steps using no more than $3m^2n+0(mn)$ processors.

We remind the reader that the algorithm can accept effectively matrices with a different number of non-zero super and subdiagonal lines or matrices of the semi-band form, but the elements of the uppermost line above diagonal has to be non-zero.

[a] Throughout this paper $\log p=\lceil \log_2 p\rceil$, and time is measured in steps.

In addition, if A is a matrix of Hessenberg form or more general of regular m-band triangular form ($a_{ij}=0$ for $j-i>m$ and $a_{i,i+m}\neq 0$ for $i=1,2,\dots,n-m$) the equations (1)-(3) are valid, too, but T is dense lower triangular matrix, now. In such a case, if one applies Algorithm I of [4] for the computation of (2) and (3), the total time for solving $Ax=b$ will be $\log^2 n+3\log n+0(m\log m)$ time steps using no more than $(15/1024)n^3+0(mn^2)$ processors.

References

[1] R.E. Bank, and D.J. Rose, "An $O(n^2)$ Method for Solving Constant Coefficient Boundary Value Problems in Two Dimensions", SIAM J. Numer. Anal. /Sept., 1975/, pp. 529 - 540.

[2] A.H. Sameh, and D.J. Kuck, " On Stable Parallel Linear System Solvers", Journal of the ACM /Jan., 1978/, pp. 81 - 91.

[3] L. Halada, "A generalization of Sylvester´s Identity", / to be published/.

[4] A.H. Sameh, and R.P. Brent, "Solving Triangular Systems on a Parallel Computer", SIAM J. Numer. Anal. /Dec., 1977/, pp. 1101 - 1113.

[5] S.C. Chen, D.J. Kuck, and A.H. Sameh, "Practical Parallel Band Triangular System Solvers", ACM Transactions on Mathematical Software /Sept., 1978/, pp. 270 - 277.

LSI IMPLEMENTATION OF MODULAR INTERCONNECTION NETWORKS FOR MIMD MACHINES

L. Ciminiera and A. Serra
Centro Elaborazione Numerale dei Segnali
c/o Istituto di Elettrotecnica Generale
Politecnico di Torino
C.so Duca degli Abruzzi, n. 24 - 10129 TORINO - Italy

Summary

This paper presents the LSI implementation of a class of permutation networks including omega, indirect binary n-cube and flip networks. Since the set of interconnection networks considered belongs to the class of delta networks, defined in [1] , the control functions can be easily distributed among several devices. A new control scheme, suitable for MIMD machines, which allows fully asynchronous operations, is also introduced.

The parallel implementation of this class of interconnection networks, without recirculating or pipelining, is discussed in this paper. The minimization of transmission delay and implementation cost is considered, taking into account the constraints imposed by the current integrated circuit, IC, technology.

The basic block, replicated for constructing the whole network, is a one nxn omega network,with $n=2^p$ instead of the 2x2 crossbar switch. In such a way the total number of modules is reduced by the factor $n/2 \cdot lg_2 n$ and the complexity of the resulting chip is moderate. Since Wu and Feng in [2] state the topological equivalence between a baseline network and the simplified manipulator, flip, omega, reverse baseline and indirect binary n-cube networks, using the nxn omega network as a basic component, it is possible to obtain each of the previously mentioned networks. In the following it will be considered that the number of inputs and outputs of the whole network to be implemented is equal to $N=2^m$ with $m>p$. Another parameter which should be taken into account is the number, B, of signals that are exchanged between each transmitter-receiver pair;it will be assumed B_1 unidirectional signals and B_2 bidirectional signals, with $B_1 + B_2 = B$.

Each integrated circuit, performing the switching function of a nxn omega network,allows the parallel transmission of w_1 unidirectional signals or w_2 bidirectional signals between each input-output pair. Obviously, the larger the values of n, w_1 and w_2 are, the smaller is the number of chips required to implement a given network of the class considered. In effect, n, w_1 and w_2 affect both the complexity of the circuit integrated in a single chip, and the number of connections (pins) required; hence the values of n, w_1 and w_2 should be determined so that the constraints imposed by the current integration and packaging technologies are satisfied. The number of connections required by the implementation of a unidirectional nxn omega network, $\Omega(n,w_1)$, is given by the following formula:

$$P(n,w_1) = 2\, w_1 n + n\, lg_2 n + 2 \qquad (1)$$

assuming a different control signal for every 2x2 crossbar switch, so that the maximum number of allowed presentations may be achieved ($n^{n/2}$).

The complexity of the implementation of one nxn omega network depends on the number of gate levels. Using 2 lg n levels, the complexity is $O(n lg_2 n)$. Implementing the same switching function using only two gate levels, the circuit obtained is faster than the first implementation and the number of gates required is given by the following formula:

$$G(n,w_1) = w_1 n(n+1) \qquad (2)$$

For bidirectional nxn omega networks it is possible to derive analogous formulas.

From equations (1) and (2) it is possible to deduce that the value of the ratio (number of gates)/(number of pins) is smaller than the current values obtained with the LSI technology; thus increasing the values of n, w_1 and w_2 , the pins available are saturated when chip area is still available.

One of the main design goals in MIMD interconnection networks is to distribute the routing functions among several units, each of them controlling a subset of the whole network, so eliminating the centralized control, which introduces performance and reliability battlenecks.

Since the ratio gates/pins of the previously IC proposed is very small, one might guess that it is feasible to put in the same chip both the connecting subnetwork and its control unit. The latter needs a set of input and output signals, therefore many other pins are required. A more attractive solution is depicted in Fig. 2, the control of a subnetwork, built with the ICs proposed here is concetrated in a dedicated chip. It

CH1569-3/80/0000-0161$00.75 © 1980 IEEE

broadcasts the command signals to the unidirectional (A,B) and bidirectional (C) switching elements of the corresponding subnetwork. The mechanism of searching and allocating the path requested through the network is described below . The request generated by a processor is issued at the input to the control unit of the subnetwork in the first stage, connected with that processor; each request is issued with the binary output device address. The control unit in the first stage receives the request signal and $lg_2 n$ bits of the output device address. This set of $lg_2 n$ address bits is chosen on the basis of the type of network implemented. In a omega network, for instance, the most significant $lg_2 n$ bits are connected with the control unit of the first stage subnetwork, the next $lg_2 n$ most significant bits are connected with the second stage control unit and so on. On the basis of the state of the switching elements, the active requests and the addresses related to them, the control unit decides whether or not to accept the request. If the request is accepted at the first stage, a request for the second stage is generated and the status of the switching elements is changed to accomodate the new connection. When the second stage receives the request issued by the first stage, an analogous mechanism starts. Thus, the path requested is searched for and allocated, stage by stage, until the target outlet is reached. If, in any stage, the control unit detects a conflict between the requested path and the connections active at that time, the status of switching elements is not changed and a busy signal is issued back to the processor through the previously allocated connections. When the busy signal is received by the requesting processor, the associated request is turned off and re-issued later. The connections are kept until the processor, which issued the request, terminates the transfer of information at that time, it clears the request and releases, stage by stage, all the trunks which compose the whole connection. A control unit for one nxn omega network, performing the functions above specified, could be implemented using an asynchronous sequential circuit, which may be integrated in a single IC. Using formulas (1), (2) and analogous formulas for bidirectional nxn omega networks, it is possible to find the values of n, w_1 and w_2 leading to the minimum number of chips required for implementing a given network of the class considered in this paper. The results of this calculation, for different values of the pins per package available, Po, are shown in Table I, where the values of n, w_1 and w_2 are calculated for a network having N=16, B_1=26, B_2=16. In this table , the values of the number of chips, C, required for implementing the above specified network, are also shown. From Table I it can be seen that, using the implementation proposed, few chips are required to built an interconnection network.

References

[1] Patel J.H., "Processor-memory interconnections for multiprocessor", Proc. 6th Ann. Symp. on Computer Architecture, April 1979, pp.168-177.

[2] Wu C. and Feng T. "Routing techniques for a class of multistage interconnection networks", Proc. 1978 Intern. Conf. on Parallel Processing, August 1978, pp. 197-205.

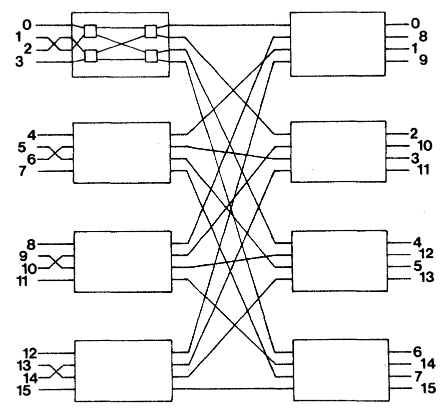


Fig.1. 16x16 indirect binary n-cu be using eight 4x4 omega networks.

TABLE I

P_O	40	60	120
n	4	4	4
w_1	4	6	4
w_2	3	6	13
c	27	18	12

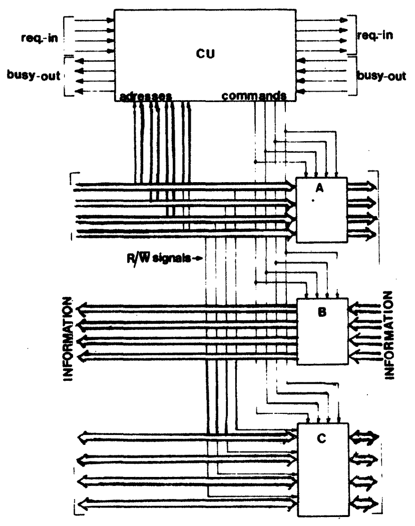


Fig.2. Interconnection between central and switching units.

ANOTHER APPROACH TO MAKING SUPERCOMPUTER BY MICROPROCESSORS--CELLULAR VECTOR COMPUTER OF VERTICAL AND HORIZONTAL PROCESSING WITH VIRTUAL COMMON MEMORY

Gao Qing-Shi Zhang Xiang

(Institute of Computing Technology, Academia Sinica)

Summary

In this paper*, Starting from the "Pipeline Vector Computer of Vertical and Horizontal Processing" ($m \times n_p$ type) [1] which is based on small and medium scale integrated circuits, then we briefly describe "Pipeline CVCVHP with Common Memory" ($m \times n$ type, $m \times n_p$ type) [2], which is introduced because of the development of large-scale integrated circuits. This is a new type of vector computer employing a multiple data stream and multiple instruction stream architecture.

Afterwards, we emphasize a new type of supercomputer, i.e. CVCVHP with "Virtual Common Memory" rather than with "Common Memory". This system may consist of thousands of cells (or microprocessors) [3].

This system has the features as follows:

1. The main part of the system can be implemented by microprocessors. one calls it cell. (It is desirable that the design of microprocessors will well suit the system configuration). There is an arithmetic unit, an instruction unit, a main memory (S2) and a bipolar memory in every cell. The bipolar memory is used for look-ahead and post buffers (L), operating registers (R), local instruction memory (S3) and high-speed memory (S1).

2. The important difference between this system and another microprocessor complex system is that the former does not need a particular OS.

3. According to the physical construction, the system is a multi-dimensional array processor, its memory is distributed. According to the functions, or from the view of user, it is a vector computer, its memory is common. The user can program in vector augmented language on a unified memory space. The move of vectors among cells is automatic, and is overlapping with the execution of arithmetic.

4. According to the different requirements of various users, the number of cells can be, 8, 16, up to thousands or tens of thousands, the system can be used alone (with the addition of I/O peripheral processor) or can be connected to another large system. Of course, a cell can also be used alone (with the addition of I/O interface).

5. The system can execute two kinds of parallel computation "multi-instruction stream" and "multi-data stream". It adopts the principle of virtual common memory, the efficiency is higher and the range of applications is wider than conventional array or vector computer (with same capacity, same speed and same number of cells.)

6. As a simplifies system, the instruction control unit can be omitted from

----

* Part of this paper was completed in Nov. 1973. and part of it was completed in Nov. 1977.

CH1569-3/80/0000-0163$00.75 © 1980 IEEE

all the cells, then the high-level language may be the same as conventional vector computers (Such as STAR-100, CRAY-1).

7. A Virtual memory system is adoptable.

An example:

Using 1o24 cells to construct a ten-dimensional array (2^{10}). The memory capacity of each cell is 16K words, 32-64 bits per word. The speed of each cell is 1 MIPS, work frequency is 15MC, 16 bits transmission with parallel and serial mode, the maximum moving time of fetching process is $1.\dot{7} \sim 2.\dot{6}$ μs, the peak value of system is one billion instructions per second.

In this system, solving a linear algebraic equation set of 4000 orders with the elimination method of column main elements, the efficiency could reach 66 per cent. If take appropriate measures, the efficiency could even over 98 per cent.

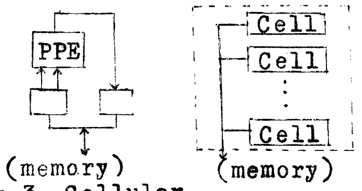


Fig.3. Cellular

REFERENCES

[1] Gao Qing-Shi, Zhang Xiang, A scheme of Pipeline Vector Computer of Vertical and Horizontal processing. Inner report in 1973, 11 and 1975. 7.

[2] Gao Qing-Shi, Zhang Xiang, On the Pipeline Cellular Vector Computer of Vertical and Horizontal Processing. Acta Electronica Sinica 1978 No. 2.

[3] Gao Qing-Shi, Zhang Xiang, A general-purpose cellular Supercomputer-CVCVHP With Virtual Common Memory. Chinese Journal of Computers Vol. 2, No. 1. 1979.

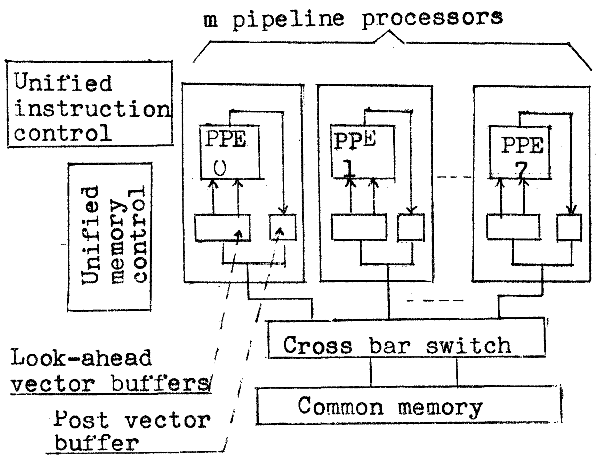


Fig.1. Pipeline Vector Computer of V.H.P.

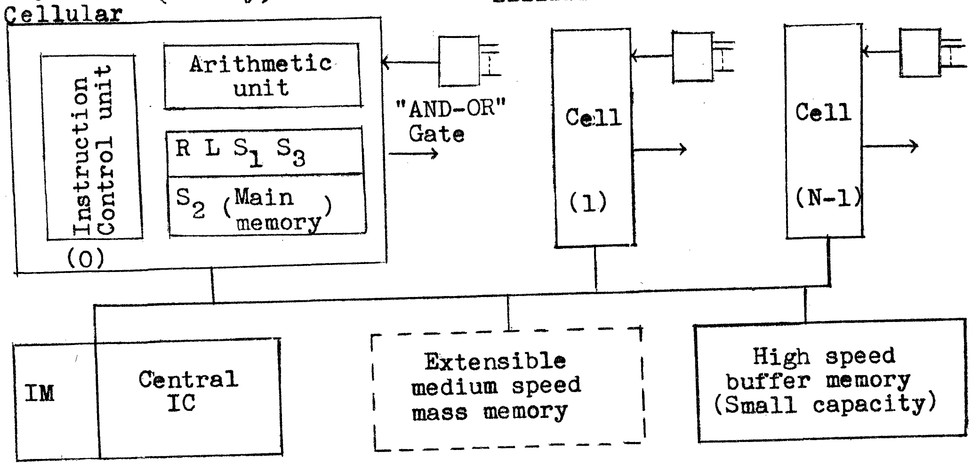


Fig.2. Cellular vector computer of V.H.P. with "virtual common memory."

An Algorithm of Parallel Processors for Theorem Proving and Its Applications

Xian Chang Zeng
Institute of Computer Science
Computer Science Department
Wuhan University
Wuhan, Hubei
People's Republic of China

Abstract. An algorithm on parallel processors is discussed for solving three artificial intelligence problems. The Robinson's resolution principle in the field of theorem proving is simplified. A formula is obtained to calculate the number of universal trails in a digraph. And the order of a free distributive lattice can be explicitely expressed by the lengths of given generating chains. The main results are described in Theorem R, Theorem E, and Theorem D.

KEY WORDS AND PHRASES: deductive algorithm, synthetic algorithm, Wuhan Parallel Processor, resolvent of well-formed formulas, universal trail on a digraph, the order of a free distributive lattice generated by chains, the speedup of a parallel algorithm.

(1) Introduction

This short paper describes some results obtained with Wuhan Parallel Processor(WuPP), which is intended to speed up digital computing by use of parallelism in Wuhan University, and has been studied by our group for almost two years. WuPP is a system of computers for MIMD parallel processors intended to support programs which consist of many independent parallel subroutines [1].

The goal for speed up computation with WuPP is restricted to much lower levels of hardware and software than many other MIMD machines. At this low level, parallelism is to befound in virtually every program, and the softwares must be rewritten or reorganized to speed up computation. But the main subject of multiprocessor research has been the effort to discover parallel programs which constitute the independent parts of the principle computation.

We have examined to solve many problems which can be programed by parallelism described above. Among the idealized models of parallel multiprocessing we have found the best ones are the following artificial intelligence problems, whose general solutions can not be obtained immediately. We should examine many special cases, and execute many programs on WuPP, then from the output datas we could find some desired results, and discover the algorithm for the general solution. We have considered the following three problems.

(2) The Robinson's problem

Let $F_1, F_2, \ldots, F_n$ be n given well-formed formulas, and G be another formula of the first order logic. If the formula $F_1 \wedge F_2 \wedge \ldots \wedge F_n \rightarrow G$ is valid, then G is called a consequence of F_1, $F_2, \ldots, F_n$. The formula $F_1 \wedge F_2 \wedge \ldots \wedge F_n \rightarrow G$ is called a theorem, and the formulas $F_1, F_2, \ldots, F_n$ are axioms [2].

The particular formula G is also called the conclusion of the theorem . A demonstration that a conclusion follows from axioms is called a proof. A procedure for finding proofs is called an algorithm of mechanical theorem proving in which a major breakthrough was made by Robinson. What is the algorithm of mechanical theorem proving, it is called the J. A. Robinson's problem.

In the field of theorem proving, the Robinson's resolution principle is well- known. According to this principle, the algorithm of mechanical theorem proving can be implemented on a digital computer, and we have programed such an algorithm on WuPP. Now we define such an algorithm with deduction as the following.

As described above, we know what a particular formula G is called the conclution of a set F which contains n given axioms $F_1, F_2, \ldots, F_n$. A deduction of G from F is a finite sequence $G_1, G_2, \ldots, G_k$ of formulas such that G_i either is a formula in F or a resolvent of formulas preceding G_i, and $G_k = G$. A deduction of empty formula from F is called the proof procedure of F, or the inconsistency of F is to be proved. The method used to get the proof procedure is called the algorithm for proving F.

We divide such algorithm into two kinds, According to their usefulness for different problems.

The first is a deductive algorithm, which is used to deduce the conclusion of a theorem from some axioms, as many authors had done in the field of theorem proving [1, 3]. But in our group the following theorem is always utilized to simplify the softwares and the parallel programs executed on WuPP for speed up computation.

THEOREM R. Let P and Q be two given sets. If there exist two sets L_1, L_2 such that $P = L_1 \vee A$, $Q = L_2 \vee B$, $L_1 = \sim L_2$, then $P \wedge Q$ is a subset of $A \vee B$.

If P, Q are formulas of the first-order logic then $A \vee B$ is the resolvent of P and Q. This theorem is equivalent to the Robinson's resolution principle, and can be easily proved. We need no background in symbolic logic to prove it, only a basic knowledge of elementary set theory is enough.

CH1569-3/80/0000-0165$00.75 © 1980 IEEE

Similariy, we can simplify some other theorems in the field of theorem proving.

The second is a synthetic algorithm which is used to get the general result of a problem from given conditions. Sometime we shall consider some problems, from which we can obtain only partial datas and certain results at special conditions, but we can not make precise decision about the conclusion. For example, how many universal trails in a given directed graph [4], what is the order of a free distributive lattice generated by n given elements [6]. We can not get immediately the general solutions of such problems.

We should examine many special cases, sometimes we need some programs and a lot of computing works on parallel processors, then the algorithm for general solutions might be found by synthetic method. We have got some results with WuPP, and shall state some in the following.

(3) The Euler's problem

An eulerian trail in a digraph G is a closed spanning walk in which each arc of G occurs exactly once. A digraph is eulerian if it has such a trail [4]. This means an eulerian digraph can be traversed by such a trail. The number of eulerian trails of a digraph was obtained by Tutte in the year 1941 [5].

If a given digraph H can be traversed by at least t eulerian trails in which each arc of H occurs exactly once, and these t trails are arranged in a fixed order. Then every such arrangement is defined as a universal trail of H. What is the number of universal trails in a digraph, it is called the Euler's problem.

If H is a general digraph, it may not be eulerian, then the outdegree and indegree of some vertex may not be equal. Let p be a point distinct from any vertex v of H, then we can join p and v with suitable or no directed arcs to make $od(v) = id(v)$. When v runs over all vertices of H, we get a new digraph D, which is called the corresponding graph of H, and this D should be an eulerian ones.

We have examined many special digraphs with their corresponding graphs, and calculated the universal trails on WuPP duing the past two years. We have the following:

THEOREM E. Let H be a general digraph, according to the method described above, we can construct an eulerian digraph D corresponding to the given H. Then the number u of universal trails of H is equal to the value of eulerian trails of D, it can be calculated by the formula

$$u = C \cdot \prod_{i=1}^{n} (d_i - 1)!$$

where $d_i = id(v_i)$ and C is the common value of the cofactors of M_{od}, and n is the number of vertices of H [4].

This theorem is a generalization of the result of Tutte and Harary [4, 5]. Let G BE a given eulerrian digraph, if the eulerian property is preserved, how many ways can be found to orient its arcs. This is still an open question. But we have examined some eulerian digraphs on WuPP, and found that the number w of ways to orient a general digraph H can be calculated by $w = 2^u$.

(4) The Dedekind's problem

Let $P_1, P_2, \ldots, P_n$ be n given propositions, which may be used to generate well-formed formulas with finite many applications of "conjunction" and "disjunction", but the application of "negation" is not permited. What is the number of non-equivalent formulas generated by n given propositions with applications of conjunction and disjunction, it is called the Dedekind's problem.

This problem was considered in 1897 by DEDEKIND [6]. He stated as the following: Let $P_1, P_2, \ldots, P_n$ be n positive integers. Where the law of conjunction means to find the greatest common divisor, and the disjunction means to find the least common mutiple. What is the number of integers generated by $P_1, P_2, \ldots, P_n$ with finite many applications of conjunction and disjunction. It is the original form of Dedekind's problem.

In the theory of lattices the problem is stated as the following: Let L be the free distributive lattice with n generators, f(n) be the order of L, i. e. the total number of elements contained in L including zero and unit. What is the exact expression of f(n), it is the form of Dedekind's problem stated by Birkholf [7]. We have known that $f(1) = 3$, $f(2) = 6$, $f(3) = 20$, $f(4) = 168$, $f(5) = 7581$, [8], which are agree with Muroga's work and can be calculated by hand. Using computer, Ward in 1946 obtained f(6) is equal to 7,828,354 [9].

It took about thirty three years, nobody could accept or refuse the Ward's result. Now we have executed programs on WuPP, and we are confident the result obtained by Ward is right. And even more we have an algorithm to calculate f(n) for n is greater than 6, which will be published in another paper.

From the proposition calculus, we know that $f(n) < 2^{2^n}$, even using computer it is difficult to find out the explicit expression of f(n). In the following we shall make a little generalization by considering the problem in another way.

Let L(k) BE the free distributive lattice generated by the following k chains with lenghs r, s, ..., t respectively:

$$\begin{aligned} 0 &= x_{10} < x_{11} < x_{12} < \cdots < x_{1,r} < x_{1,r+1} = I, \\ 0 &= x_{20} < x_{21} < x_{22} < \cdots < x_{2,s} < x_{2,s+1} = I, \\ &\cdots\cdots\cdots\cdots\cdots\cdots \\ 0 &= x_{k0} < x_{k1} < x_{k2} < \cdots < x_{k,t} < x_{k,t+1} = I, \end{aligned}$$

where 0, I are the zero and unit elements of L(k) respectively.

For $k \leqslant 3$, we have completely solved the Dedekind's problem, i. e. we can explicitly express the order of L(3) in terms of the lengths of the generating chains. The result is the following:

THEOREM D. Let L(3) be the free distributive lattice generated by three chains of lengths r, s, t respectivelly. And let [r, s, t] be the order of L(3), then we have

$$[r,\ s,\ t] = \frac{(r+s+t+2)!!\,r!!s!!t}{(r+s+1)!!(s+t+1)!!(t+r+1)}$$

where $r!! = r!(r-1)!\ldots 3!2!1!$ when $r = s = t = 2$, we get $[2,\ 2,\ 2] = 980$. When $t = 0$, then we have $[r,\ s] = (r+s+2)!/(r+1)!(s+1)!$, which was obtained by Birkhoff [7].

When k = 3, it is called the case of 3 variables for Dedekind's problem. We have also obtained some special results for 4, 5, 6 variables. For example, we have the following explicit forms:

$$[1,\ 1,\ 1,\ t] = \binom{t+5}{4} + 15\binom{t+6}{6} + 48\binom{t+7}{8}$$

where $\binom{s}{t} = \frac{(s+t)!}{s!t!}$, and we have

$$[1,\ 1,\ 2,\ t]\quad (t+4)\cdot\left[\frac{1}{3}\binom{t+6}{5} + \frac{21}{2}\binom{t+7}{7} + \frac{432}{5}\binom{t+8}{9} + \frac{631}{3}\binom{t+9}{11}\right].$$

$$[1,\ 1,\ 3,\ t] = \binom{t+7}{6} + 98\binom{t+8}{8} + 2580\binom{t+9}{10}$$
$$26668\binom{t+10}{12} + 11730\binom{t+11}{14} + 183958\binom{t+12}{16}.$$

$$[r,s,t]\cdot[r,s,t] = [r,s-1,t+1]\cdot[r,s+1,t-1] + [r-1,s,t]\cdot[r+1,s,t].$$

The last formula can be used to calculate the large values of [r, s, t] from the smaller ones of r, s, t.

We had examined many special cases and execut--ed a lot of programs on WuPP, before we obtained the formula described in Theorem D. Similarly, we have got the following explicit form for four variables

$$[1,1,4,t] = a_1\cdot\binom{t+8}{7} + a_2\cdot\binom{t+8}{8} + \cdots + a_{14}\cdot\binom{t+8}{20}$$

Since the a's are very large, the programs used to calculate them should be parallel, otherwise the running time would be very long. We have:

$a_1 = 1$, $a_2 = 188$, $a_3 = 9468$, $a_4 = 204700$,
$a_5 = 2353308$, $a_6 = 16185984$, $a_7 = 71429138$,
$a_8 = 210763120$, $a_9 = 424570176$, $a_{10} = 585753336$,
$a_{11} = 544446914$, $a_{12} = 325903320$,
$a_{13} = 113456486$, $a_{14} = 17454844$.

If we define the comlete set for the polynomial [1, 1, s, t] as in the next paper [12], then the complete set of [1, 1, 6, t] contains the following 28 constants:

```
               1               538             74349
         4401070         141734859        2846807080
     38890518764      383234658030     2840173234855
  16321468185832    74410146837079   273816862566536
 923954320708219  2047246583910286  4229330326988924
7297544167083306
                 10541417443127500
12886752737504718                  12750255339454502
                 10834085412245518 75193649367395l4
4259430756992310 1936572050862474  6894621461504l2
 1851083363634l2   352370078914 28   4238527206900
    242201554680
```

Knuth stated a problem: "Investigate three-dimensional arrays, in order to see how many of the properties of two-dimensional Young's tableaux can be generalized." [10]. We conjecture that if this problem is solved, then the Dedekind's problem will be also solved.

If we define the speedup of a parallel algorithm as Lemme and Rice [11], then the Dedekind's problem in the case of four variables can be implemented on (q+1)(r+1)(s+1) processors, and [q, r, s, t] can be calculated.

References

1 Zeng X.C., (1978) Mechanical Theorem Proving and Dedekind's Problem, Science Report of Wuhan University, Wuhan, China, 4, 19-31.
2 Chang C.L., (1973) Symbolic Logic and Mechanical Theorem Proving, Academic Press, New YORK.
3 Ballantyne A.M. and Bledsoe W.W., (1977) Automatic Proofs of Therems in Analysis, Using Nonstandard Techniquess, J. ACM 24, 3, 353--374.
4 Harary F., (1972) Graph Theory, Addison-Wesley, Reading, Mass.
5 Smith, C.A.B. and Tutte W.T. (1941), On Universal Paths in a Network of Degree 4, Amer. Math. Monthly 48, 233--237.
6 Dedekind R. (1897) uber zerlegungen von Zahlen durch ihre grossten gemeinsamen Teiler, Ges. Werke, Bd. II, 103--148.
7 Birhoff G., (1965) Lattice Theory, 3rd ed. A. M. S. Collquium Pub., New York.
8 Muroga S., (1971) Threshold Logic and Its Applications, John Wiley, New York.
9 Ward M., (1946) Note on the Order of the Free Distributive Lattice, Bull.A.M.S.
10 Knuth D.E., (1973) The Art of Computer Programming, Reading, Mass.
11 Lemme J.M., and Rice J. R., (1979) Speedup in Parallel Algorithms for Adaptive Quadrature, J. ACM 26, 1, 65--71.
12 Zeng X.C., (1980) An Algorithm on Parallel Processing for Theorem Proving and Solving Dedekind Problem, This Proceeding.

An Algorithm on Parallel Processing for Theorem Proving and Solving Dedekind's Problem

Xian Chang Zeng
Institute of Computer Science
Computer Science Department
Wuhan University
Wuhan, Hubei
People's Republic of China

ABSTRACT. First in the field of theorem proving, the Robinson's resolution principle is considered how to be applied to general resolution. Second in the field of graph theory, let H be a generaldigraph which can be traversed at least by t trails. Suppose this property of H is preserved, how many ways to orient H? This problem is solved, and the Theorem E in the previous paper can berigorously proved. Third in the field of Lattice Theory, some properties and more details about the algorithm for solving Dedekind's problem are obtained. The complete sets of polynomials [1, 2, 3, t], [1, 1, 6, t] and [1, 1, 1, 2, t] are calculated. Finally an algorithm for computing pairs of twin primes is described.

(1) The Robinson's problem

In the previous paper [1], the algorithm for solving three artificial intelligence problems was discussed. Now we describe some new results and more details of this algorithm obtained about one year ago on Wuhan Parallel Processor (WuPP) with parallel programs by our group in Wuhan University.

First we consider the Theorem R in the previous paper. Robinson in an unpublished paper wrote the following opinon:

Perhaps Theorem R can be applied to general resolution by taken the elements of the sets to be models?

He had given a hint to solve this problem and said: "See discussion in my book on compactness, topology, and completeness." [4]

But we have examined many special cases and execute many programs with WuPP and found that it is intemately related to the foundations of logic and lattice theory, so Theorem R may be proved by axiom method. We will discuss it in another paper.

(2) The Euler's problem

Second we consider the Euler's problem. Let H be a given general digraph which can be traversed at least by t trails. According to the method described in the previous paper, we can construct an eulerian digraph D corresponding to H. And we can prove the number u of universal trails of H is equal to that of D. Therefore it can be calculated by the following formula:

(A) $$u = C \cdot \prod_{i=1}^{n} (d_i - 1)!.$$

where the meaning of C, n, d_i can be found in the previous paper. And we have:

Corollary E. Let H be a given general digraph, which may not be eulerian. If its property traversed at least by t trails is preserved, how many ways can be found to orient its arcs. We will solve this problem. Let w be the ways to orient its arcs, then we can prove the following:

(B) $$w = 2^{u}.$$

where u is the number of universal trails of H. The rigorous proof of (A) or (B) is a little long, we omit it. But it is not difficut to prove (B) from (A). We just need to consider the definition of universal trail of H. If the t trails used to traverse H are not arranged in a fixed order, then the equality should be $w = t! \times 2^{u}$.

(3) The Dedekind's problem

Third we consider the Dedekind's problem. Using the similar method described in the previous paper, we can discuss the case of four variables for solving Dedekind's problem as the following. Let L(4) be the free distributive lattice generated by 4 chains of length q, r, s, t respectively, and let [q, r, s, t] be the order of L(4), then we have:

Corollary D. (1) The order [q, r, s, t] is a symmetrical function of the 4 variables q, r, s, and t. For example [q, r, s, t] = [q, s, r, t] = [q, r, t, s] = [s, r, t, q].

(2) When any three, say q, r, s of the four variables are fixed, then [q, r, s, t] is a polynomial of the remaining t, and with degree $(q+1)(r+1)(s+1)$. For example [1, 1, 1, t] is a polynomial of t with degree 8.

(3) The polynomial [q, r, s, t] vanishes when $t = -2, -3, -4, -5, \ldots, -(q+r+s)-2$. For example [1, 1, 1, t] has 4 zeros at $t=-2$, $t=-3$, $t=-4$, and $t=-5$.

(4) When $t=-1$, the value of [q, r, s, t] is equal to 1, so that [q, r, s, -1] = 1.

(5) When the variable t is replaced by $-(q+r+s+t+4)$ then the value of [q, r, s, t] is not changed, i. e.

[q, r, s, t] = [q, r, s, $-(q+r+s+t+4)$]

For example [1, 1, 1, t] [1, 1, 1, -t-7], especially [1, 1, 1, 1] = [1, 1, 1, -8] = f(4) = 168.

(6) When one of the variables, say t, is equal to zero then this variable can be stricken out from the function [q, r, s, t]. For example [q, r, s, 0] = [q, r, s].

It is not difficulty to prove the properties (1)--(6) in the above Corollary D. But it is long for writing in a short paper, so we omit the proof. Using these properties we can easily find the formula and the algorithm with parallel programs for computing [q, r, s, t]. For example the degree of polynomial [1, 1, 1, t] is equal to 8, so we need 9 values to determine its coefficients. From Corollary D, we have

$$\begin{array}{ll} [1,1,1,1] = 168, & [1,1,1,0] = 20, \\ [1,1,1,-1] = 0, & [1,1,1,-2] = 0, \\ [1,1,1,-3] = 0, & [1,1,1,-4] = 0, \\ [1,1,1,-5] = 0, & [1,1,1,-6] = 1, \\ [1,1,1,-7] = 20, & [1,1,1,-8] = 168. \end{array}$$

We already have 10 values of [1, 1, 1, t], so it can be completely determined. Therefore we have:

$$[1, 1, 1, t] = \binom{t+2}{1} + 18\binom{t+2}{2} + 111\binom{t+2}{3} + 311\binom{t+2}{4} + 540\binom{t+2}{5} + 495\binom{t+2}{6} + 240\binom{t+2}{7} + 48\binom{t+2}{8}$$

where $\binom{t+2}{k} = \frac{(t+2)!}{(t+2-k)!k!}$, which is agree with the result of previous paper. The following set of constants:

1 18 111 311 540 495 240 48

is defined to be the complete set of the polynomial 1, 1, 1, t . Similarly we have found the complete set of polynomial [1, 2, 3, t] contains the following 24 constants:

1	488	58075
2857576	74739914	1198664320
1289252957	98739543042	561446104803
2442211577600	8304538700959	22420686173258
48579030087180	85044074724532	120670489307306
138726071967266	128680989347607	95484963754408
55868886745375	25197675728000	8450421827031
19841050181136	29131681235	200711150430

And the complete set of polynomial 1, 1, 6, t contains 28 constants, which can be found from the previous paper.

One year ago we computed the constant 183957 for the complete set of [1, 1, 3, t], the running time was half an hour. But now we use parallel programs, it is about two minutes to get the number 12886752737504718 for the complete set of [1, 1, 6, t]. We hope the general formula of [1, 1, s, t], can be found from the constants given by this paper. This problem is closely related to the Muroga's works [2].

(4) The twin primes problem

Finally we have examined many special cases with computer and find an algorithm to computing the number Z(N) of the pairs of twin primes less than N. We will describe the method which is very similar to the sieve of Eratosthenes:

Let the sequence formed by the natural numbers $\leq N$ be the following:

(S) 1, 2, 3, 4, ..., N 1, N.

Then the algorithm for computing the pairs of twin primes can be defined by 4 steps:

Step (1). All even numbers $\leq N$ are stricken out (sieved out) of (S) and let $i = 1$.

Step (2). Let p_i be the ith odd prime of (S), $p_0 = 1$, $p_1 = 3$, and all proper multiple of p_i are stricken out from (S).

Step (3). All numbers of the form $(k+1)p_i - 2$ less than or equal to N are stricken out from (S), and k runs through all positive integers.

Step (4). When $p_{i-1} > \sqrt{N}$, the algorithm will be stopped, otherwise i willbe replaced by the value of $i + 1$, and go to Step (2).

For example, let N = 200 we get the following sequence:

(T) 3, 5, 11, 17, 29, 41, 59, 71, 101, 107, 137, 149 , 175, 191, 197.

Now we need to prove the following two properties:

(P) If and only if the first member of twin primes willbe contained in the sequence (T).

(Q) When N tends to infinite the sequence (T) willbe also tends to infinite. i. e. There exists infinite many pairs of twin primes.

Let p be an odd prime, if $p+2$ is not a prime, then it is composite, and $p + 2 = q \cdot r$ where q is a prime less than p. Hence $p = q \cdot r - 2$, and $r > 1$, by the property of Step (4), this prime p must be stricken out of (S).

When $p + 2 = q$ and q is an another prime, then $q - 2 = p$ is a prime. This p can not be written as $(k+1) \cdot p_i - 2$, otherwise $q = (k+1)p_i$ will be composite. Also by the property of Step (4), the prime $p = q - 2$, can not be stricken out of (S). Therefore we completely proved the property (P).

It is difficult to prove the property (Q). We belive that the conjecture "There are infinite many pairs of twin primes." is true. And moreover we belive also that the conjecture of Danial Shanks:

$$Z(N) \sim 1.3203236 \int_2^N \frac{dn}{(\log n)^2}$$

is true [3]. But the details are intemately related to the Dirichlet's Theorem: "Every arithematic progression $an + b$, where a, b are relative prime integers, and n runs through all positive in integers." We like to discuss this problem in another paper.

References

1 Zeng X.C., (1980) An Algorithm of Parallel Processors for Theorem Proving and Its Applications, This Proceedings.

2 Muroga S., (1971) Threshold Logic and Its Applications, John Wiley, New York.

3 Shanks D., (1962) Solved and Unsolved Problems in Number Theorem, Spartan Books, Washington D. C.

4 Robinson J. A., (1979) Logic: Form and function, The Mechanization of Deductive Reasoning, Edinburgh University Press, Edinburgh.

SESSION 6: DISTRIBUTED PROCESSING I

Design and Implementation of a Language for Communicating Sequential Processes

Mehdi Jazayeri

TRW Vidar
77 Ortega Avenue
Mountain View
California 94040, USA.

Carlo Ghezzi

Istituto di Elettrotecnica ed Elettronica
Politecnico di Milano
Piazza L. da Vinci 32
20133 Milano, ITALY.

Dan Hoffman

David Middleton

Mark Smotherman

Department of Computer Science
University of North Carolina
Chapel Hill
North Carolina 27514, USA.

ABSTRACT

Aspects of the design and implementation of CSP/80, a language based on Hoare's communicating sequential processes, are discussed. The goal of the design has been to stay as close to Hoare's original notation as possible. The goal of the implementation has been to reduce the amount of reinvention by making utmost use of facilities provided by the operating system (UNIX). This has shortened the implementation time considerably. CSP/80 is to be used for evaluating CSP as a programming language for distributed processing applications.

CH1569-3/80/0000-0173$00.75 © 1980 IEEE

INTRODUCTION

In "Communicating Sequential Processes" [1], Hoare proposed an elegant notation for programming distributed systems. The notation, hereafter called CSP, combines Dijkstra's nondeterministic control structures for sequential programming [2] with "blocked" input/output for communication and synchronization between parallel processes. In order to evaluate CSP's utility as a concurrent programming language, we have undertaken to produce a prototype implementation. Our final goal is to apply the language in programming several distributed systems. This paper presents the design and implementation of our version of CSP, called CSP/80. We discuss and motivate the important design decisions and the deviations from Hoare's version. As Hoare observed in his paper [1, p. 667], his notation "should not be regarded as suitable for use as a programming language...". The work reported in this paper has been aimed at producing a suitable programming language based on Hoare's notation.

CSP is one of several recent proposals for distributed processing. As Brinch Hansen points out in [3], however, these proposals must be evaluated based on their use in practice. In order to do this experimental evaluation, one needs an implementation of the concept. Because the implementation is a vehicle for evaluation of an untried approach, the implementation time must be kept small. Therefore, we have tried wherever possible to use already existing facilities.

We have tried to include in CSP/80 two methodological ideas that have been found to be valuable in designing other types of programs. These are modular programming and strong typing. The idea of modular programming is that it should be possible to design the different modules of a program independently. The requirement in CSP that a process must name all the processes it communicates with violates this rule. We have tried to remedy this by introducing ports and channels. Ports and channels have also allowed us to turn CSP into a strongly typed language (i.e. all type checking can be done at compile time). This is a very important characteristic of a language that helps in the development of reliable software.

In section 2 we briefly review the language concepts of CSP. In section 3 we give the differences between CSP and CSP/80. We do not go into great detail about the reasons for these differences; these are covered in [4]. In section 4 we discuss the implementation of the language and the important design decisions made. Section 5 concludes the paper.

COMMUNICATING SEQUENTIAL PROCESSES

A program in CSP consists of a fixed number of parallel processes (which may run on distinct processors). Each process consists of a series of sequential statements. Statements are provided for assignment, alternation, repetition, input and output. The assignment statement is similar to that in other languages. The alternation and repetition statements are based on Dijkstra [2]. Input and output are the only really novel concepts provided by the notation. An input (correspondingly, output) statement names a process from which (to which) the input (output) is to be received (sent). Upon execution of an input (output) command, the process is suspended until a corresponding output (input) is performed by the named process. At that point, the input/output transaction takes place and both processes continue execution. The I/O commands thus provide for both communication and synchronization between processes. Furthermore, Hoare allows the use of input commands in the guards of the alternative and repetitive statements. Such a guard is selected only if the partner process has already committed (i.e. been suspended due to having requested an output to this process).

In the next section we discuss where, how and why we have deviated from Hoare's notation.

CSP VS. CSP/80

Program structure

A program in CSP/80 consists of a fixed number of (separately compiled) parallel processes, and a list of channel declarations. Each process may have one or more input or output ports through which it communicates with other processes. A channel declaration establishes a link between a port in one process and a port in another process.

As an example, the bounded buffer example of Hoare [1, p. 673] written in CSP/80 is shown in Fig. 1. The complete syntax of CSP/80 is given in Appendix A.

```
process produce ::
output int Y;
int s;
s = 0;
*[ 1 -> s = s + 1;
            !Y = s;
 ]
end process

process consume ::
input int Z;
int s;
int sum;
sum = 0;
*[ 1 -> ?s = Z;
        sum = sum + s;
 ]
end process

process X ::
guarded input int Y;
guarded output int Z;
int in;
int out;
int buffer[9];
in = 0;
out = 0;
*[ in < out + 10;
    ?buffer[in%10] = Y
        -> in = in + 1;
   |out < in;
    !Z = buffer[out%10]
        -> out = out + 1;
 ]
end process


/*buffered version*/
int channel from produce.Y
             to X.Y
int channel from X.Z
             to consume.Z

/*unbuffered version*/
int channel from produce.Y
             to consume.Z

Note: "%" is the modulo operator.
```

Figure 1: produce and consume in two configurations

Inter-Process communication and synchronization

In order to send information from process P to process Q, P must have an output port, x, and Q an input port, y. These ports must be linked by a channel, declared:

<type> channel from P.x to Q.y

<type> is the data type of the information being transferred and must be the same as the types associated with x and y. The actual transfer takes place after both of the following have taken place (in either order):

- x has been used as the target of an assignment statement in P; (!x = expression;)
- y has been used as the source of an assignment statement in Q. (?variable = y;)

The way input/output is done is different from CSP, where communicating processes must name one another and a type mismatch is caught only at run-time. The use of typed ports in CSP/80 allows type checking to be done at compile time. The use of a channel allows a process to be written without explicit knowledge of the name(s) of the communicating partner(s). This allows a process to be connected to different processes without recompiling the process. The connection is performed by a "linker". The ability to reconfigure the system without recompilation is an attractive capability in a distributed system. Figure 1 shows two possible ways that the same producer and consumer may be connected.

The use of typed ports also removes the need for one of Hoare's constructs. Instead of the special signals, e.g. has(n), we can define a port by the mnemonic name, e.g. has. Any I/O through this port then has the meaning of has(n).

Alternative and repetitive commands

CSP/80 is identical to CSP in this respect except that we allow output commands to appear in guards as well. If a port name is to appear in guards, however, the port declaration must declare the port as guarded. This is to enable the detection of the anomalous situation where two communicating processes both have their I/O statements in guards. This situation, which is a form of deadlock, was the reason Hoare ruled out the possibility of output commands in guards. With our solution, we allow more freedom and still provide a measure of protection. This issue is discussed at length in [3,4].

Arrays of processes and channels

Just as in CSP, CSP/80 allows the use of arrays of processes. The effect of the following process declaration:

process P[i:0..9]::...... end process

is the same as having ten processes called P(0),P(1)... P(9). The occurrence of the bound variable i in P is replaced by 0 in P(0), 1 in P(1), etc. Any variables and ports declared in P are local and therefore no confusion can exist in channel declarations. For example,

<type> channel from P(1).x to P(2).y

We also allow channel declarations of the form:

    <type> channel(i:0..9)from P(i).x to P(i+1mod10).y

Both arrays of processes and channels are merely shorthand notations and do not add any power to the language. They can be regarded as a primitive macro processing capability.

IMPLEMENTATION

Goals

The language is implemented on a PDP11/45 in the programming language C under the UNIX [6] operating system. The overriding concern in the implementation has been to limit the time and effort required for implementation and still provide us with programming experience in CSP. Thus, all features have been restricted in such a way as to produce a usable language and also allow for future expansion of the system. For example, currently only (scalar and array) integer and character data types are supported. It is relatively straightforward to add other data types to the language and the current language is sufficiently powerful for investigating distributed processes.

Another example is the implementation of the nondeterministic control structures. We have made no effort to ensure that the selection of guards is completely random. Our goal was not to investigate nondeterminism. Furthermore, our implementation does meet Dijkstra's requirements for an implementation of the guarded commands [2].

System structure

The CSP/80 system consists of a translator and a linker. The translator accepts one CSP/80 process and translates it into a C program (which will run as a UNIX process). The linker accepts the names of some CSP processes (already compiled by the C compiler) and a list of channel declarations. It produces an executable CSP/80 program consisting of the processes communicating via the channels. Both the translator and the linker were written using LEX and YACC, the translator writing tools available under UNIX.

Run-time organization

At execution-time, a CSP program consists of a set of UNIX processes each representing a CSP process, and another UNIX process called the monitor, which coordinates the communication between processes. There is also a direct access file, called the channel file, which contains the required channel buffers. A UNIX pipe connects all processes with the monitor. This organization is shown in Figure 2.

A simple output command from process P to process Q is implemented in the following way: P writes its output in the location in the channel file reserved for output on the appropriate channel. It then sends a message along the pipe to the monitor indicating that it requires an output service. P then puts itself to sleep (by invoking pause, a UNIX primitive). An input command is implemented in a similar way.

The monitor constantly reads its request pipe, responding as soon as a request arrives. The pipe mechanism provides for the queuing of messages that arrive while another one is being serviced. If the request is an input (or output), the monitor sets the status of the requesting process as committed to perform input (or output). It then checks to see whether the other partner is committed. If it is, then the output data is copied from the output to the input section of the appropriate channel buffer (in the channel file). Note that channels and ports are implemented simply by a location within the channel file that the two communicating processes use to write into and read from. The linker assigns the address of this location to the partner processes. Thus channels and ports are conceptual tools for specifying the communication, and they allow strong type checking and can be implemented efficiently in a uniprocessor.

The more interesting interaction between the processes occurs when an I/O command appears in a guard. According to the semantics of the construct, this guard can be chosen only if the partner process has already committed. Furthermore, there may exist several guards with I/O commands in them. This is implemented by the process sending an activity check request to the monitor and going to sleep. This request asks the monitor to wake up the process if and when any of the process's partners either commit to an I/O operation with this process or terminate. Upon waking up, if the commitment affects any of the guards, then that guard is selected. Otherwise, the process issues another activity check request and goes to sleep.

A final interaction between the processes and the monitor occurs when a process, just before terminating, sends a message to that effect to the monitor. The monitor records that information which will be of use to the process's partners.

Guard selection and fairness

The implementation of the guarded commands is a compromise between polled (busy waiting) and interrupt driven processing. In the polled case, the process would constantly test the guards until one of them became true (as a result of a partner process making a commitment). This might be acceptable if processes were indeed allocated to distinct processors. On a uniprocessor, on the other hand, this is disastrous because this testing of the guards would itself use up some if not all of the time otherwise available to other processes waiting to do useful work (including the commitment necessary for the resumption of the waiting process).

In the interrupt-driven case, the process would go to sleep after asking to be woken up when any of the processes it is waiting on makes a commitment. It would only be awakened when a guard can be selected. This is more efficient of processor time than polling but is more complicated to implement.

In our implementation, the process goes to sleep and is awakened if any of its partners has an activity (commitment or termination). It is possible that the activity does not affect any of the guards, in which case the process goes back to sleep after testing all its guards. Thus the process is not always busy waiting, just sometimes. This implementation is much less complicated than the pure interrupt-driven case and is more efficient of processor time than pure polling.

For simplicity, our implementation tests the guards in purely sequential order. If the first guard is always true, therefore, the other guards will never be selected. Although this is unfair and could lead to starvation of certain processes, such scheduling policies are not ruled out by the language semantics. It is the responsibility of the programmer to write programs that do not rely on specific scheduling policies.

Extensions

Although the current version is adequate for our purpose of the evaluation of CSP as a programming language, we envision expanding the system capabilities a great deal. First, we intend to enhance CSP/80 for multiprocessor operation. Design efforts are underway for a two processor version to be inplemented on VAX 11/780 computers. With this implementation meaningful benchmarks can be run. In particular, we would like to measure how much time a process spends waiting for its partner to commit and how much of this time could be saved if non-blocked I/O were used. We would also like to enhance CSP/80 to:

- support more data types
- provide simple deadlock detection and handling,
- use a fairer guard selection algorithm.

CONCLUSIONS

We have described the design and implementation of CSP/80, an implementation of Hoare's communicating sequential processes. As far as we know, this is the first such implementation. The implementation is heavily based on and uses facilities provided by UNIX (and C) to minimize implementation time. Although this restricts our ability to exert control over process scheduling, it has resulted in a quick implementation.

BIBLIOGRAPHY

1. Hoare, C.A.R. Communicating Sequential Processes. Comm. ACM 21, 8 (August 1978), 666-677.

2. Dijkstra, E.W. A Discipline of Programming. Prentice-Hall, Englewood Cliffs, N.J., 1976.

3. Brinch Hansen, P. Distributed Processes: A Concurrent Programming Concept. Comm. ACM 21, 11 (November 1978), 934-941.

4. Ghezzi, C., and Jazayeri, M. More Comments on Communicating Sequential Processing. Dept. of Computer Science, Univ. North Carolina, Chapel Hill, N.C., 1979.

5. Kieburtz, R.B., and Silberschatz, A. Comments on Communicating Sequential Processes. ACM TOPLAS 1, 2 (October 1979), 218-225.

6. Ritchie, D.M., and Thompson, K. The UNIX Time-Sharing System. Bell System Tech. J. 57, 6 (July-August 1978), 1905-1929.

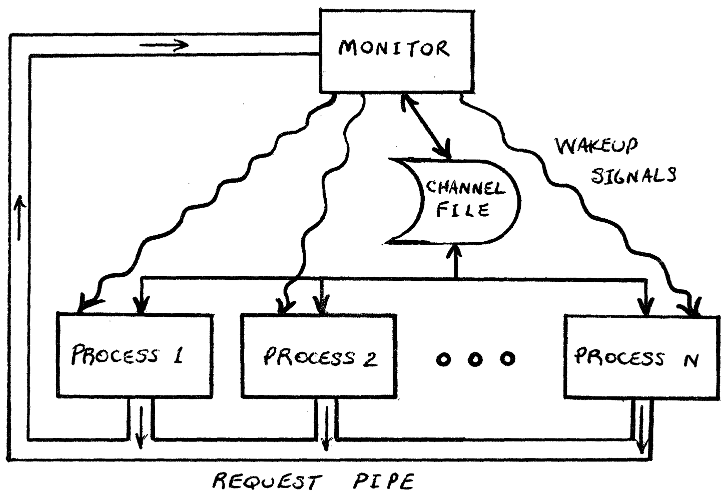


Figure 2. Run-time organization of CSP/80 processes

APPENDIX A

Below is the modified BNF description of CSP/80. What is shown has been extracted from the actual input to LEX, a lexical analyzer, and YACC, a parser generator, both available under UNIX. Nonterminals are in lower case; terminals are in upper-case. The metacharacters are:

: "produces"

| or

; end of a production

The meaning of the terminals is shown in the table following the productions.

```
process : PROCESS IDENT
          range
          DBLCOL portdec
          decls
          stmnts
          END PROCESS
        ;
range   : /* empty */
        | LPAREN IDENT
          COLON NUM
          RANGE NUM
          RPAREN
        ;
portdec : /* empty */
        | portdec guarded INPUT
          PORT TYPE
          dim   /* element size */
          IDENT
          dim   /* number of ports */
          SEMICOL
        ;
decls   : /* empty */
        | decls decl SEMICOL
        ;
decl    : TYPE
          IDENT
          dim
        ;
dim     : /* no bound => scalar */
        | LBRA
          NUM
          RBRA
        ;
guarded : GUARDED
        | /* empty */
        ;
stmnts  : command
        | stmnts
          command
        ;
```

```
command : SKIP SEMICOL
        |
          expn SEMICOL
        |
          io SEMICOL
        | alt
        | PASSTHROUGH
        | error
        ;
alt     : choice
          alterns
          RBRA
        ;
choice  : REP
        | LBRA
        ;
alterns : altern
        | alterns BOX altern
        ;
altern  :
          guard
          ARROW decls stmnts
        ;
guard   : bool
          decls
        | bool
          SEMICOL
          decls io
        |
          decls io
        ;
bool    :
          expn
        | bool SEMICOL
          expn
        ;
expn    : NUM
        | STRING
        | QUOTE
        | IDENT
          sub
        | LPAREN
          expn
          RPAREN
        | expn op expn
        | expn op
        | op expn
        ;
sub     : /* empty */
        | LBRA
          expn
          RBRA
        ;
op      : OP
        | EQUALS
        ;
io      : QUERY
          target
          EQUALS port
        | EXCLAM
          port
          EQUALS expn
        ;
target  : IDENT
          tsub
        ;
```

```
tsub    : /* empty */
        | LBRA
          expn
          RBRA
        ;
port    : IDENT
          psub
        ;
psub    : /* empty */
        | LPAREN
          expn
          RPAREN
        ;
```

ARROW	"->"
BOX	"|"
COLON	":"
DBLCOL	"::"
END	"end"
EQUALS	"="
EXCLAM	"!"
GUARDED	"guarded"
IDENT	C identifier
INPUT	"input"
LPAREN	"("
LBRA	"["
NUM	unsigned integer
OP	"~", "*", "/", "+", "%", "<", ">", "&", "^"
PASSTHROUGH	a line with "#" in column 1
PORT	"port"
QUERY	"?"
QUOTE	a single character delimited by single quotes
RANGE	".."
RBRA	"]"
REP	"*["
RPAREN	")"
SEMICOL	";"
SKIP	"skip"
STRING	a string delimited by double quotes
TYPE	"int"

A COMPREHENSIVE FRAMEWORK FOR EVALUATING DECENTRALIZED CONTROL

John A. Stankovic
Department of Electrical and Computer Engineering
University of Massachusetts
Amherst, Massachusetts 01003

Abstract -- Effective decentralized control algorithms will help achieve many of the potential advantages of highly cooperative distributed systems. Currently, there is no unified approach for developing and analyzing decentralized control algorithms. This paper describes a comprehensive framework that can serve such a purpose. Highlighted in the framework are the underlying principles of distributed systems and the need for effective evaluative techniques. A partial example of the application of the framework to a simple decentralized job scheduling algorithm is also presented.

1. Introduction

The dramatic reduction in computer costs coupled with the potential advantages of connecting computers in a network makes distributed processing systems inevitable. These potential advantages include increased resource sharing, better performance, higher reliability and easier extensibility than possible with uniprocessors. However, current distributed systems achieve these advantages in a very limited manner due to the multitude of "new" problems that distribution causes. Foremost among these problems are the high cost and critical nature of centralized control. These two issues must be resolved before the potential advantages of distributed processing can be realized to a large degree.

This paper describes a comprehensive framework for decentralized control. The framework aids the development and analysis of decentralized control algorithms. These algorithms can then be used to eliminate the high cost and critical nature of centralized control. Although there is a wide spectrum of distributed systems, the framework concentrates on one specific type of distributed processing system that is still in the early research stage. Specifically, it addresses distributed systems characterized by decentralized system-wide control of resources for the cooperative execution of application programs. By decentralized system-wide control we mean that overall executive control is exercised through the cooperation of decentralized system elements to form a single organism [21]. For the sake of brevity the term "system-wide" will be dropped when speaking of decentralized control in this paper. The proposed framework is applicable to the decentralized control of any function that must operate with incomplete or inconsistent data and under strict time requirements. Such functions might include routing, scheduling and resource allocation.

The framework is described in section 2. As an example, a new decentralized job scheduling algorithm is partially described and evaluated by means of the framework in section 3. Finally, the usefulness, potential and limitations of the framework are summarized in section 4.

2. A Framework for Decentralized Control

Currently, there is no unified approach for developing and analyzing decentralized control algorithms. In this section a comprehensive framework that can serve such a purpose is developed. First, the minimum requirements of the framework are stated, and then the framework itself is described. Since this field of research is in its infancy, the philosophy behind the development of the framework is to allow for easy extensibility and modifiability as new fundamental principles of distributed systems are discovered.

Requirements

The minimum requirements of the decentralized control framework are:

1) to address the central issues of decentralized control including,
   a) concurrency,
   b) operation in the presence of missing, incomplete or erroneous state information,
   c) uniqueness in time and space principle (see 2.2), and
   d) cost (overhead) of the algorithms,
2) to enable meaningful evaluation of the decentralized control algorithms,
3) to provide a convenient structure for the development and comparison of new algorithms,
4) to be generalizable to all functions, and
5) to allow for the incorporation of new research results.

The Framework

In the development of the framework, a distributed system is viewed as a collection of functions, f_i, where each function f_i of the system must be controlled by a decentralized system-wide control algorithm, X_i, which utilizes the set of state information $\{Y_i\}$ to achieve a set of goals $\{Z_i\}$. As an example, the function f_i might

CH1569-3/80/0000-0181$00.75 © 1980 IEEE

include routing, message communication, scheduling, resource allocation, data management, and distributed applications. Then for each f_i we develop algorithms X_{i1}, X_{i2}, X_{i3}, ..., where each X_{ij} has different sets of state information $\{Y_{ij}\}$. The set of goals (requirements) $\{Z_i\}$ is chosen by the designers depending on the function f_i and the application. A formal specification of the requirements is necessary to fully evaluate the solution. Since formal specifications is an active research area, the current framework permits an informal specification of the goals (requirements) as the set $\{Z_i\}$.

The intent of this approach is that for a given f_i, various decentralized control algorithms X_{ij}, can be developed and compared as a function of the state information and design goals (informally stated at this time). Important research questions include the amount of state information required to properly control function f_i, how that information is accessed, and how many distributed entities decide on the resultant control choices. The formulation of the problem in these terms is necessary to meet conditions 2, 3, and 4 of the framework, although not sufficient.

The next step in the development of the framework is to incorporate the central issues of decentralized control known at this time.

Concurrency. Each algorithm X_{ij} is implemented by multiple entities e_1, e_2,...,e_n running asynchronously and acting together but without central control or data.

Operation in the presence of missing, incomplete or erroneous state information. A fundamental characteristic of distributed systems is the long and unpredictable delays experienced in interprocess communication giving rise to missing or incomplete state information. This implies a great need for algorithms that can effectively operate under these conditions. In general, distributed systems will also experience greater probability of errors (hardware and software) than uniprocessors giving rise to the greater need for resilient algorithms.

Uniqueness in Time and Space Principle. One central principle confronting the design of decentralized control algorithms involves dealing with the absence of uniqueness both in time and space [26]. This characteristic of distributed system implies that the multiple, decentralized entities implementing the control algorithm get either a partial and coherent (i.e., observations are made at the same moment in the system--universal time) view, or a complete but incoherent view of the system. The consequences of this characteristic are not fully understood, but it is a critical distinction between centralized and decentralized systems.

The framework addresses the "uniqueness of time and space" issue, by viewing five dimensions of decentralized control.

1) Global Environment - this is the sum total of all the local environments at one instance of universal time. For decentralized control algorithms it will be impossible to know the global state accurately. Yet, some information about the global environment is necessary for system-wide control algorithms.

2) Local Environment - the state of the machine on which this entity is executing. Some subset of the information about the local environment will also be used by the algorithm. In most systems this information is assumed to be correct and timely. There are no special assumptions about the correctness of the local data in this framework.

3) Algorithm - The algorithm's logic obviously plays an important role in the effectiveness of the decentralized control. The algorithm's logic cannot assume that it knows or can construct the absolute chronological ordering of events, nor that the set of entities implementing the algorithm perceive identically the set of events in the system.

4) Data - This is the actual information about the global and local environment used by the algorithm. The data may be missing, incomplete or in error.

5) Time - There is no universal time reference.

In decomposing the "uniqueness of time and space" principle into these five components, we believe that the implications of this principle for decentralized control algorithms can be better understood and algorithms being developed will better address the important aspects of the problem. The example of the next section should help clarify this point.

Overhead of the Algorithms

In order to address the overhead of algorithms issue, the algorithms, X_{ij}, must meet the following conditions:

a) execute to meet strict time requirements,
b) be decentralized (i.e., X_{ij} will be implemented by multiple entities acting together but without central control or data),
c) be able to operate with uncertain, missing or erroneous data, and
d) require no more than a specified amount of memory.

The evaluation of decentralized control algorithms will consist of two parts. The first is an absolute evaluation that determines if the algorithm meets its requirements. The second is a comparative evaluation of different algorithms that meet the requirements. At a minimum the decentralized control framework requires that the following parameters be part of the evaluation:

- o performance (e.g. response time and throughput,
- o logical correctness (absence of deadlocks, cycles, etc.),
- o resiliency (capable of operating in the presence of failures as well as recovering from failures),
- o overhead (execution time, memory, and communication costs),
- o stability (presence of an anomaly should not have chaotic effects),
- o fairness,
- o extensibility (the algorithm should easily control additional resources of the same type),
- o cost and difficulty of initialization, and
- o understandability.

Although it is not possible (to date) to quantify many of these parameters, the choice of a practical algorithm should take all of these parameters into consideration. In general, the measurements of these parameters for decentralized control algorithms are open research questions. As part of the extensibility of the framework 1) new parameters may be added to this list, and 2) new techniques for evaluation of these parameters can replace techniques shown to be inferior. Currently, the appropriateness of different mathematical techniques (mathematical programming, dynamic programming, game theory, decision theory under uncertainty, and decentralized control theory) for use in the evaluation of the performance parameter is under investigation. Presently, decision theory under uncertainty utilizing a Bayesian decision strategy seems promising.

The elements of this decentralized control framework are meant to be general for all functions. Individual functions may have certain specialized characteristics that require extensions of this basic framework to deal with these characteristics.

3. Decentralized Control of Job Scheduling

This section provides an example of how the proposed framework for decentralized control might be applied to the development and evaluation of a new decentralized control algorithm for the function of job scheduling. For the sake of brevity some issues of the framework are summarily dismissed. In practice, every issue of the framework would be addressed in detail. A network of seven hosts configured as in Figure 1 is assumed for this discussion.

Using terminology of the framework, the function f_i to be addressed is job scheduling. There exist seven entities e_1, e_2, ..., e_7 that taken together implement f_i. These entities execute on hosts 1, 2, ..., 7 respectively. The local state information at host i is the length of the queue of jobs waiting to enter the system at host i. The global state information used at host i is host i's perception of the queue lengths at the other host locations. Note, that the framework allows iterative changes to these state information quantities for direct comparison. The primary goal of the algorithm is assumed to be a high throughput of jobs. In practice the requirements on throughput would be more precise and the requirements on all other parameters listed in section 2 would also be addressed.

Briefly, the intent of this decentralized control algorithm is to perform load balancing at the job level. Periodically, each entity, e_i, updates its Workload Table, sends load estimates to its neighbors, and performs scheduling which might include movement of jobs to other hosts[(a)]. These factors are part of the overhead costs of the algorithm.

Figure 2, conceptually illustrates the Workload Tables that are maintained at each host (in actuality not all columns need be retained between updates implying a low memory overhead). Table i exists at host i. The first column of each table i is host i's view of the system. The additional columns in table i correspond to the nearest neighbors (defined as having a direct physical interconnection) of host i, e.g. in Figure 1 host 1 has nearest neighbors 2 and 4. Hence TABLE-1 of Figure 2 has 3 columns labelled 1, 2 and 4. Conceptually, these additional columns of the table are host i's perception of its nearest neighbors view of the system.

The actual values in the table are workload estimates calculated based on the <u>state information</u> chosen as part of the algorithm. In this example, this is simply the number of jobs in a queue. In general, host i can determine precisely the number of jobs in its own queue (accurate local data) and therefore will believe his own estimate rather than his neighbors perception of his workload. These values are the boxes marked with vertical lines in Figure 2. Since nearest neighbors of host i are only 1 step away, their estimates of their workload as passed to host i will be only slightly our of date and in general be a better estimate than estimates other nodes have of them. Therefore, host i will assign a <u>higher probability</u> of correctness to nearest neighbors estimate of themselves (boxes marked with horizontal lines). All other estimates are grouped into a third probability category. If the precise configuration of the network is known, weights could be assigned to the estimates proportional to the distance from host i. In the third probability category, host i determines the workload by computing an average of the columns of nearest neighbors. Using the average is an arbitrary choice at this time. Only after a

(a) Not addressed in this paper are implementation issues of job movement, such as data translation if non-homogeneous hosts are involved.

proper evaluative comparison will the choice of an average be substantiated.

Each table is periodically updated by a host using messages from its nearest neighbors. For example, host 1 receives messages from host 2 and host 4 containing their view of the system (i.e., their column vectors). This is global data. Host 1 then recalculates column 1. To do this host 1 looks at its job queue to obtain the number of jobs waiting at host 1 and places this number in the first column, first row. It then takes host 2's view of 2 and places this number in the first column, second row, then host 4's view of 4 is placed in the first column, 4th row. All other entries in the first column are calculated by taking an average of host 2 and 4's perception of other hosts. In general there may be more than two columns (see Table 2).

Periodically a scheduling decision must be made. If host j is substantially less busy than host i then some number of jobs will be moved to j from i. Both the substantial difference parameter and the number of jobs to move are important variables. In this algorithm a substantial difference is chosen to be 3 jobs and $\left\lfloor \frac{\#i + \#j - 3}{2} \right\rfloor$ jobs are moved assuming this calculation results in a positive number. The jobs moved are taken from the back of the queue to account for some degree of fairness. At this point the algorithm is completely ad hoc. A substantial evaluation is required before we attest to the usefulness of this algorithm.

In this example, the variables that must be varied and evaluated are:

- o the substantial difference variable,
- o using an average or should other weighting schemes be used,
- o the period of update,
- o the probability assigned to the nearest neighbor view,
- o the state information used, and
- o the number of jobs to move.

Note that during the entire development of the algorithm, the five dimensions of control are constantly kept in mind. The global environment, the local environment, the algorithm's logic performed by multiple entities which do not perceive the identical set of events in the system, the missing, incomplete or erroneous state of the data used by the algorithm, and the fact that there is no universal time reference are all incorporated into the algorithm either explicitly or implicitly.

Finally, in addition to the variables just mentioned the algorithm must also be evaluated according to the 9 major evaluation parameters of the framework. Performance can be evaluated by any of the standard techniques; analytical models, simulations or implementations and measurement. This is, of course, easier said than done. In many cases closed form solutions are not possible and simulation studies will have to be used. However, we are actively pursuing the use of decision theory under uncertainty as a mathematical treatment of the evaluation of performance.

The other evaluation parameters (logical correctness, resiliency, overhead, stability, fairness, extensibility, cost and difficulty of initialization, and understandability) will not be discussed in this paper. However, the reader might notice that the algorithm presented has many of the same problems as the original ARPA routing algorithm (e.g. ping-ponging).

4. Conclusions

The main ideas behind the development of this framework are 1) to provide a structure in which to think, develop and analyze decentralized control algorithms, 2) to provide a convenient mechanism for a more meaningful comparison of proposed algorithms (meaningful in the sense that the assumptions, strengths and weaknesses of the algorithms are addressed), and 3) to encourage the development of more mathematical techniques for the evaluation aspects of the framework.

Currently, a major limitation of the framework is the scarcity of effective techniques for evaluating the parameters. A mathematical formulation of the problem is being sought. We are investigating the possibility of using decision theory under uncertainty, cooperative game theory, utility theory and mathematical programming. Other evaluation parameters like understandability might always be subjective but nevertheless should be addressed as best as possible. Finally, the framework described in this paper is merely the beginnings of a comprehensive framework.

Bibliography

[1] Austin, Donald M., editor, Proceedings of the Second Berkeley Workshop on Distributed Data Management and Computer Networks, Computer Science and Applied Math Dept. Lawrence Berkeley Laboratory, University of California May 1977.

[2] Bokhari, S.H., "Dual Processor Scheduling with Dynamic Reassignment," IEEE Transactions on Software Engineering, Vol. SE-5, No. 4, July 1979.

[3] Bokhari, S.H., "Optimal Assignments in Dual-Processor Distributed Systems Under Varying Load Conditions," ICASE Report No. 79-14, July 1979.

[4] Casey, L., and N. Shelness, "A Domain Structure for Distributed Computer Systems," Proceedings of the Sixth ACM Symposium on Operating Systems Principles, Nov. 1977, pp. 101-108.

[5] Chow, W., F. Ferrante, and M. Bolagangadhar, "Integrated Optimization of Distributed Processing Networks," Computer Studies, Technical Report, TR 77-01, North Carolina

State University, Raleigh, N.C., 1977.

[6] Chow, Yuan-Chieh, and Walter H. Kohler, "Dynamic Load Balancing in Homogeneous Two-Processor Distributed Systems," in Computer Performance, K.M. Chandy and M. Reiser, Eds., New York: North Holland, 1977.

[7] Chow, Yuan-Chieh, and Walter H. Kohler, "Models for Dynamic Load Balancing in a Heterogeneous Multiple Processor System," IEEE Transactions in Computers, Vol. C-28, No. 5, May 1979.

[8] Conway, Richard W., William L. Maxwell, and Louis W. Miller, Theory of Scheduling, Addison-Wesley Publishing Co., Reading, Mass. 1967.

[9] Denning, P.J., "Fault-Tolerant Operating Systems," ACM Computing Surveys, Vol. 8, No. 4, December 1976, pp. 359-389.

[10] Farber, David J., and Kenneth C. Larsen, "The Structure of a Distributed Processing System - Software," Proceedings Symposium on Computer Communications Networks and Teletraffic, Microwave Research Institute of Polytechnical Institute of Brooklyn, April 1972.

[11] Farber,David J., et al., "The Distributed Computer System," Proceesings 7th Annual IEEE Computer Society International Conference, February 1973.

[12] Feiler, Peter, "Implementation Issues of a Distributed Operating System, Notes on the Cm* Operating System," presented at the Brown University Workshop, August 1976.

[13] Forsdick, Harry C., Richard E. Schantz, and Robert H. Thomas, "Operating Systems for Computer Networks," IEEE Computer, Vol. II, No. 1, January 1978.

[14] Foschini, G.J., "On Heavy Traffic Diffusion Analysis and Dynamic Routing in Packet Switched Networks," Computer Performance, K.M. Chandy and M. Reiser (eds.), North Holland Publishing Co., 1977, pp. 499-513.

[15] Gallager, R.G., "A Minimum Delay Routing Algorithm Using Distributed Computation," IEEE Transactions on Communications, Vol. COM-25, No. 1, January 1977, pp. 73-85.

[16] Gerla, Mario, and Leonard Kleinrock, "On The Topological Design of Distributed Computer Network," IEEE Transactions on Communications, Vol. COM-25, No. 1, January 1977, pp. 48-60.

[17] Gonzalez, M.J.,Jr., "Deterministic Processor Scheduling," ACM Computing Surveys, Vol. 9, No. 3, Sept. 1977, pp. 173-204.

[18] Graham, G.S., editor, "Special Issue: Queueing Network Models of Computer System Performance," ACM Computing Surveys, Vol. 10, No. 3., Sept. 1978.

[19] Hamilton, Jim, "Functional Specification for the WEB Kernel," Digital Equipment Corporation, R & D Group, Maynard, Mass., Nov. 1978.

[20] Hewitt, C., et al., "Parallelism and Synchronization in Actor Systems," 1977 Conference on Principles of Programming Languages, Los Angeles, California, January 1977, pp. 267-280.

[21] Jensen, Douglas, "The Honeywell Experimental Distributed Processor--An Overview of its Objectives, Philosophy and Architectural Facilities," IEEE Computer, Vol. 11, No. 1, January 1978.

[22] Jones, A.K., R.J. Chansler, I. Durham, P. Feiler, K. Schwans, "Software Management of Cm*, a Distributed Multiprocessor," AFIPS Conference, Vol. 46, NCC, 1977.

[23] Kleinrock, Leonard, Queueing Systems: Volume 2: Computer Applications, John Wiley & Sons, New York, 1976.

[24] Kleinrock, Leonard, and Holger Opderbeck, "Throughput in the ARPANET--Protocols and Measurement," IEEE Transactions on Communications, Vol. COM-25, January 1977, pp. 95-104.

[25] Lamport, L., "Time Clocks and the Ordering of Events in a Distributed System," Massachusetts Computer Associates,Wakefield Mass., March 1976.

[26] Le Lann, G., "Distributed systems--towards a formal approach," Proceedings IFIP Congress, Toronto, North Holland Pub., Aug. 1977, pp. 155-160.

[27] Le Lann, G., "Algorithms for distributed data-sharing systems which use tickets," Proc. 3rd Berkeley Workshop on Distributed Data Management and Computer Networks, Aug. 1978, pp. 259-272.

[28] Lesser, Victor, and Daniel D. Corkill, "Functionally-Accurate Cooperative Distributed Systems," Computer and Information Sciences Technical Report, Univ. of Mass., February 1979.

[29] Lipsky, L., and J.D. Church, "Applications of a Queueing Network Model for a Computer System," ACM Computing Surveys, Vol. 9, No. 3, Spet. 1977, pp. 205-221.

[30] McQuillan, J.M., "Throughput in the ARPA Network--Analysis and Measurement," BBN Report 2491, January 1973.

[31] McQuillan, J.M., "Adaptive Routing Algorithms for Distributed Computer Networks," BBN Report No. 2831, May 1974.

[32] McQuillan, J.M., and D.C. Walden, "The ARPA Network Design Decisions," Computer Networks, The International Journal of Distributed Informatique, Vol. 1, No. 5, August 1977, pp. 243-289.

[33] Menasce, Daniel A., and Richard R. Muntz, "Locking and Deadlock Detection in Distributed Data Bases," IEEE Transactions on Software Engineering, Vol. SE-5, No. 3, May 1979.

[34] Michel, J. and Andries van Dam, "Experience with Distributed Processing on Host/Satellite Graphics Systems," in Proceedings SIGGRAPH, 1976.

[35] Mishkin, Eli, and Ludwig Braun, Jr., Adaptive Control Systems, McGraw Hill, Inc., New York, 1961.

[36] Myers, Glenford J., Composite Structured Design, Van Nostrand Reinhold Co., N.Y., N.Y., 1978.

[37] Narendra, K.S., editor, Proceedings of the Workshop on Applications of Adaptive Control Yale University, August 1979.

[38] Rosenkrantz, Daniel, Richard E. Stearns, and Philip M. Lewis II, "System Level Concurrency Control for Distributed Database Systems," ACM Transactions on Database Systems Vol. 3, No. 2, June 1978, pp. 178-198.

[39] Schwartz, Mischa, Computer Communication Network Design and Analysis, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1977.

[40] Segall, A., "The Modeling of Adaptive Routing in Data-Communication Networks," IEEE Transactions on Communications, Vol. COM-25, No. 1, January 1977, pp. 85-95.

[41] Stabler, G.M., "A System for Interconnected Processing," Ph.D. Dissertation, Brown Univ., Providence, R.I.; Oct. 1974.

[42] Stankovic, John A., and Andries van Dam, "The Distributed Processing Workshop," Brown University Technical Report CS-32, available from the IEEE Computer Society Repository R77-373, August 1977.

[43] Stankovic, John A., et al., "Issues in Distributed Processing," IEEE Computer, Vol. 11, No. 1, January 1978.

[44] Stankovic, John A., "Structured Systems and Their Performance Improvement Through Vertical Migration," Ph.D. Thesis, Brown University, CS Technical Report CS-41, May 1979.

[45] Stankovic, John A., "Communications Mechanisms: Procedure Calls Versus Messages," submitted to IEEE Computer, August 1979.

[46] Stankovic, John A., and Andries van Dam, "Research Directions in (Cooperative) Distributed Processing," Research Directions in Software Technology, MIT Press, Cambridge, Mass., 1979.

[47] Stevens, W.P., G.J. Myers, and L.L. Constantine, "Structured Design," IBM Systems Journal, Vol. 13, No. 2, 1974, pp. 115-139.

[48] Stone, H.S., "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," IEEE Transactions on Software Engineering, Vol. SE-3, January 1977.

[49] Stone, H.S., "Critical Load Factors in Distributed Computer Systems," IEEE Transactions on Software Engineering, Vol. SE-4, May 1978.

[50] Sunshine, Carl, "Formal Techniques for Protocol Specification and Verification," IEEE Computer, Vol. 12, No. 9, September 1979.

[51] Swan, R.J., S.H. Fuller, and D.P. Siewiorek, "Cm*: a Modular, Multi-Multiprocessor," AFIPS Conference, Vol. 46, NCC, 1977.

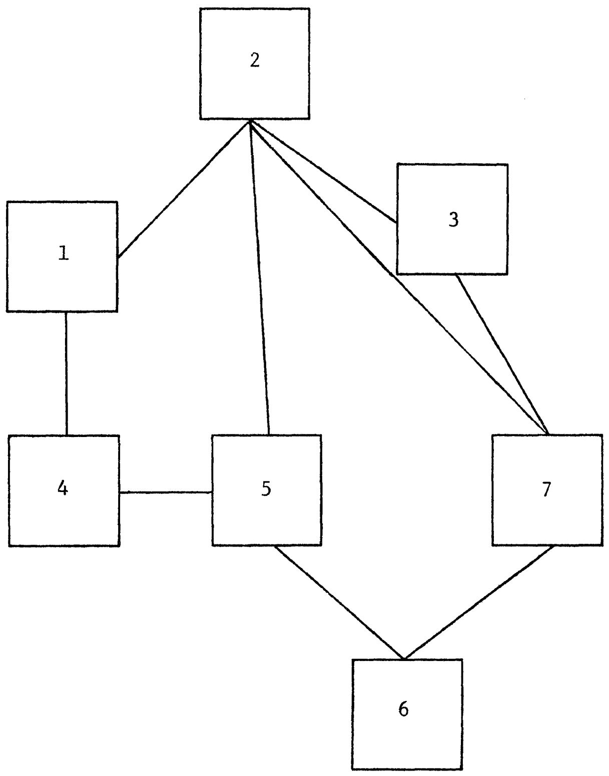


Figure 1: Arbitrary Distributed Network

TABLE - 1

	1	2	4
1	▥		
2		▤	
3			
4			▤
5			
6			
7			

TABLE - 2

	2	1	5	7	3
1		▤			
2	▥				
3					▤
4					
5			▤		
6					
7				▤	

TABLE - 3

	3	2	7
1			
2		▤	
3	▥		
4			
5			
6			
7			▤

TABLE - 4

	4	1	5
1		▤	
2			
3			
4	▥		
5			▤
6			
7			

TABLE - 5

	5	4	2	6
1				
2			▤	
3				
4		▤		
5	▥			
6				▤
7				

TABLE - 6

	6	7	5
1			
2			
3			
4			
5			▤
6	▥		
7		▤	

TABLE - 7

	7	2	3	6
1				
2		▤		
3			▤	
4				
5				
6				▤
7	▥			

Figure 2: "Workload" Tables

DIRECTIONS FOR USER DEFINED COMMUNICATION FOR DISTRIBUTED SOFTWARE

Robert B. Kolstad & Roy H. Campbell
Department of Computer Science
University of Illinois
Urbana, Illinois 61801

Summary

Advances in hardware technology have decreased costs of processors and memory, thus permitting collections of processors to be coupled cost-effectively into distributed computing systems. The distribution of computing resources may facilitate access speed, physical control, and contention reduction (though load sharing may become more difficult) [24]. The interconnection of computer resources to allow processes to communicate is a difficult task, though success has been achieved in closely coupled environments [21], [4], [1], [19]. Loosely coupled environments such as networks do not yield elegant solutions so quickly. Much research proceeds on low level network protocols and a few researchers have enhanced the users' ability to communicate between processes in a nonhomogeneous environment [14], [26], [24], [15], [13], [22].

Our criteria for implementing software for connected systems are evolving with gains of knowledge about and experience with these systems. We believe that a modular specification technique which allows separation and static description of synchronization, concurrency, and data access is necessary and that it is important to have the ability to develop software for connected systems (utilizing user specified communication) in a uniform, top-down manner. The ability to specify and implement true concurrency along with freedom of concern (during specification, design, and coding) of actual physical embodiment of a process's execution are fundamental. Desirable solutions meeting these criteria will include few extensions to current thinking and hide implementation details from the programmer. Certain qualities enhance general programming languages: conciseness, strong typing and user specifiable types, a direct relation between an algorithm's complexity and the complexity of its representation, separation and orthogonality of available language constructs (constructs should not overlap in function), and the ability to make static declarations (instead of data-dependent ones). These properties enhance modifiability, reliability, portability, readability, maintainability, and promote higher productivity. Unfortunately, few high level languages offer facilities to exploit abstraction of synchronization, concurrency, and communication.

Other researchers have proposed different methods for interprocess and interprocessor communication. Explicit send/receive of messages (sometimes with automatic message encoding) has been proposed by [20], [15], [13], [16], [19], [18], [24], [28], and [25]. Signal/wait is similar to send/receive and has been proposed by [2], [3], and [16]. Many schemes rely on shared memory [4], [5], [2], [3], [22]. Those schemes which allow communication in a loosely coupled environment usually require fixed configurations [4], [5], [15], and [10]. All schemes so far (except [16]) use dynamic synchronization methods or simple mutual exclusion. The schemes of [10] and [16] provide facilities similar to our proposal below but at the expense of extending language syntax.

Path Pascal was developed by augmenting Pascal [17] with a small number of orthogonal constructs to specify concurrency, encapsulation, and synchronization. The Path Pascal object encapsulates a set of data, a set of services to operate on the data, initialization for the data, and a specification of the synchronization for the services. Path Pascal [8], [21], [6], [7] contains path expressions [9] for synchronization and the process declaration for concurrency. Although Path Pascal has only been used in a closely coupled environment (with shared memory), we believe its object construct models the desired behavior of a (possibly remote) service.

Path Pascal objects are normally used in a multiprocessor environment with shared memory (or a multiplexed uniprocessor environment). We propose that their implementation be extended to encompass not only tightly coupled systems but also loosely coupled ones. In this new methodology, objects exist on any of the connected processors and communication between them is restricted to the invocation of objects' operations and return of var parameters. This invocation represents a transfer of control from the invoking process to the (possibly remote) object. Flow of control is always explicitly and deterministically controlled by the user: invocations of processes create new flows; terminations of processes destroy old ones. This control methodology resembles that of [12], strongly resembles [10], and is different from [11] which is the basis of several previous schemes.

This networking of objects is achieved by compiling invocations of "foreign" objects to calls on special communication routines which encode the parameters of the invocation (including references to other objects) [29], transmit them to the foreign object's host, await their return, decode return arguments, and return control to the invoking process. This represents only a change in the scoping and compilation of objects and is a dual of message passing systems [23].

The advantages of objects hold: encapsulation exists for each object (each object is represented on one machine) synchronization specifications are maintained, and data can be manipulated by all processes (local or remote) possessing the capability [12] for invoking the data's object's operations. Network objects extend the convenient manipulation of shared data to loosely coupled systems and require no changes or extensions to Path Pascal's syntax (though separate compilation becomes desirable). The link editor deduces the

CH1569-3/80/0000-0188$00.75 © 1980 IEEE

location of objects and imposes performance penalties only on true foreign object references.

Instantiation, naming, and compiling of new objects on different hosts requires special care [27]. The communications template of an object will be portable and is distributed to remote processors which require communication facilities. Binding of Path Pascal processes and objects to individual processors can be performed any time before execution of the process or object begins.

References

[1] Brinch Hansen, P., Operating System Princ., Prentice Hall, Englewood Cliffs, N. J., 1973.

[2] Brinch Hansen, Per, "The Programming Language Concurrent Pascal," IEEE Trans. on Software Engineering, Vol. SE-1, No. 2, June, 1975.

[3] Brinch Hansen, Per, "Experience with Modular Concurrent Programming," IEEE Trans. on Software Engineering, Vol. SE-3, No. 2, March, 1977.

[4] Brinch Hansen, P., "Multiprocessor Architectures for Concurrent Programs," Proc. of 1978 Nat'l ACM Conf., pp. 317-323, 1978.

[5] Brinch Hansen, Per, "Distributed Processes: A Concurrent Programming Concept", Comm. ACM, Vol. 21, No. 11, pp. 934-941, November, 1978.

[6] Campbell, R. H. and R. B. Kolstad, "Path Expressions in Pascal," Fourth Int'l Conf. on Software Eng., Munich, Sept. 17-19, 1979.

[7] Campbell, R. H. and R. B. Kolstad, "Practical Applications of Path Expressions to Systems Programming," ACM79, Detroit, 1979.

[8] Campbell, R. H. and R. B. Kolstad, "A Practical Implementation of Path Pascal," Technical Report, Dept. of Comp. Sci., Univ. of Ill. at Urbana-Champaign, UIUCDCS-R-80-1008, 1980.

[9] Campbell, R. H., "Path Expressions: A technique for specifying process synchronization," Ph.D. Thesis, U. of Newcastle upon Tyne, August, 1976; Also, DCS Tech. Report, Univ. of Ill. at Urbana-Champaign, UIUCDCS-R-77-863, May, 1977.

[10] Cook, Robert, "*MOD, A Language for Distributed Computing," First Int'l Conf. on Distr. Comp. Systems, Huntsville, Ala., pp. 233-241. Oct. 1-5, 1979.

[11] Dijkstra, E. W., "Guarded Commands, Non-Determinancy and Formal Derivations of Programs," CACM, Vol. 18, No. 8, pp. 453-457, August, 1975.

[12] Fabry, R. S., "Preliminary Description of a Supervisor for a Machine Oriented Around Capabilities," ICR Quarterly Report 18, Univ. of Chicago, COO-614-64, Aug., 1968.

[13] Feldman, Jerome A., "High Level Programming for Distributed Computing," Comm. ACM, pp. 353-367, Vol. 22, No. 6, June, 1979.

[14] Geller, Dennis F., "The National Software Works: Access to Distributed Files and Tools," Proc. of 1977 Nat'l ACM Conf., pp. 39-43, 1977.

[15] Hoare, C. A. R., "Communicating Sequential Processes," CACM, Vol. 21, No. 8, pp. 666-677, August, 1978.

[16] Ichbiah, J. D., et al., "Rationale for the Design of ADA..." & "Preliminary ADA Reference Manual", SIGPLAN, Vol. 14, No. 6, June, 1979.

[17] Jensen, K. and N. Wirth, Pascal User Manual and Report: Second Ed., Springer-Verlag, 1974.

[18] Jones, A. and K. Schwans, "TASK Forces: Distributed Software for Solving Problems of Substantial Size," Fourth Int'l Conf. on Software Eng., Munich, pp. 315-330, Sept. 17-19, 1979.

[19] Jones, A., et al., "StarOS, a Multiprocessor Operating System for the Support of Task Forces," Proc. of 7th Symp. on OS Princ., pp. 117-127, Dec. 10-12, 1978.

[20] Kahn, G., "The Semantics of a Simple Language for Parallel Programming," Proc. of 1974 IFIPS, pp. 471-475, North Holland Pub. Co., 1974.

[21] Kolstad, R. B. and R. H. Campbell, "Path Pascal User Manual," Dept. of Comp. Sci., Univ. of Ill. at Urbana-Champaign, Tech. Rept. UIUCDCS-R-80-893, Feb., 1980.

[22] Lampson, Butler W. and David D. Redell, "Experience with processes and monitors in Mesa," Symp. on O/S Princ., pp. 43-44, 1979.

[23] Lauer, Hugh and R. Needham, "On the Duality of Operating System Structures," Second Int'l Symp. on Op. Systems, IRIA, Oct., 1978.

[24] Liskov, B., "Primitives for Distributed Computing," Proc. of 7th Symp. on OS Princ., pp. 33-42, Pacific Grove, CA, Dec. 10-12, 1979.

[25] Mao, Tsang and Raymond Yeh, "Communication Port: A Language Concept for Concurrent Programming," First Int'l Conf. on Distr. Computing Systems, Huntsville, Ala., pp. 305-314, Oct. 1-5, 1979.

[26] Millstein, Robert E., "The National Software Works: A Distributed Processing System," Proc. of 1977 Nat'l ACM Conf., pp. 44-52, 1977.

[27] Peterson, James L., "Notes on a Workshop on Distributed Computing," Operating Systems Review (SIGOPS), Vol. 13, No. 3, pp. 18-27, July, 1979.

[28] Tarini, F., et al., "A Network System Language," First Int'l Conf. on Distr. Comp. Systems, Huntsville, Ala., pp. 305-314, Oct. 1-5, 1979.

[29] Wallis, Peter J. L., "External Representations of Objects of User-Defined Type," ACM Trans. on Programming Lang. and Systems, pp. 137-152, Vol. 2, No. 2, April, 1980.

SESSION 7: NUMERICAL ALGORITHMS AND APPLICATIONS

SIMD ALGORITHMS TO PERFORM LINEAR PREDICTIVE CODING FOR SPEECH PROCESSING APPLICATIONS

Leah J. Siegel, Howard Jay Siegel, Robert J. Safranek, Mark A. Yoder
Purdue University, School of Electrical Engineering
West Lafayette, Indiana 47907

ABSTRACT

The use of the SIMD (single instruction stream - multiple data stream) mode of parallelism to perform the speech analysis task of linear predictive coding is explored. Linear prediction represents one of the major analysis techniques for speech compression, transmission, and recognition applications. Parallel algorithms to perform linear prediction have been developed, and are evaluated in terms of the number of arithmetic operations and interprocessor data transfers needed. From the algorithms, architectural requirements such as machine size and interconnection network capability are analyzed.

I. INTRODUCTION

Because of the complexities involved in a general purpose parallel system, it is becoming apparent that one practical way to harness the power of large-scale parallel processing may be to consider its use as applied to a specific type or class of tasks. One area which appears to be well suited for such consideration is speech processing. Speech analysis, performed for either data compression or speech recognition purposes, involves substantial computation on vectors and arrays. SIMD (single instruction stream - multiple data stream [6]) parallelism may therefore be applicable to a number of speech processing tasks. Studies of how SIMD machines can be used to perform fast Fourier transforms [25] and pitch detection [1] have confirmed that speech processing operations can benefit from the SIMD mode of parallel processing.

In this paper, the use of SIMD machines to perform the speech analysis operation of linear predictive coding [2,11,12,16] is explored. Linear prediction is closely related to techniques in time series analysis [4], Kalman filtering [7], and Wiener filtering [28]. It is one of the principal analysis methods used for speech compression, transmission, and recognition [2,16,23], and is applicable to problems in neurophysics and seismic signal processing [11].

A general model of an SIMD machine is assumed for the development and analysis of parallel linear prediction algorithms. The SIMD machine model consists of a control unit, a set of $N=2^n$ processing elements (PEs), each a processor with its own memory, and an interconnection network [19]. The control unit broadcasts instructions to all PEs, and each active PE executes each instruction on the data in its own memory. The instruction is executed simultaneously in all active PEs.

This material is based upon work supported by the National Science Foundation under Grant ECS-7909016.

The interconnection network enables data to be transferred among the PEs. Each transfer is expressed in terms of an interconnection function, where interconnection function f is a bijection on the set of PEs which transfers a data item from PE i to PE f(i). The transfer occurs simultaneously for all i for which PE i is active [17].

Detailed SIMD algorithms to perform linear prediction analysis are given in [26]. In these algorithms, some known SIMD programming techniques have been extended [8,14] and new methods are introduced. In this paper, the relative complexities of corresponding serial and parallel algorithms are reported, and the machine size and interconnection network requirements of the algorithms are presented. The network requirements are expressed in terms of the interconnection functions which are executed. The ability of interconnection networks in the literature to perform the required transfers is discussed.

II. LINEAR PREDICTIVE CODING

Speech production is commonly modeled as a filter driven by an excitation component. The filter represents the configuration of the vocal tract - i.e., the positioning of the mouth, nose, and throat. The excitation represents the air flow from the lungs which has been either transformed into a periodic sequence of pulses by the vocal cords (the pitch in the production of pitched sounds), or set into rapid, "noise-like" motion by being forced past some constriction (e.g., the teeth against the lower lips in the production of an "f").

Linear predictive coding (LPC) analysis operates on a sampled signal {s}, where, if m is an integer variable, s(m) represents the m-th sampled value of a continuous-time speech signal. In the linear prediction model, it is assumed that each sample s(m) of the signal {s} can be expressed as the sum of two components, one a weighted sum of the previous p samples, and the other a residual component $\delta(m)$ which may differ for each s(m) [2,11,16]:

$$s(m) = \sum_{k=1}^{p} a(k)s(m-k) + \delta(m).$$

The weighted sum portion can be interpreted as the "predicted" value $\hat{s}(m)$ for speech sample s(m). If it is assumed that each s(m) can be approximated by $\hat{s}(m)$, then the predictor coefficients (a(k)'s) can be obtained by minimizing the total squared prediction error, defined as

$$E^2 = \sum_m [s(m)-\hat{s}(m)]^2 = \sum_m [s(m)-\sum_{k=1}^{p} a(k)s(m-k)]^2 \qquad (1)$$

The minimization of E^2 is performed by solving the set of equations

$$\frac{\partial E^2}{\partial a(k)} = 0 \qquad 1 \leq k \leq p. \tag{2}$$

By choosing the interval over which the linear prediction analysis is performed to correspond to an interval over which physiology precludes a significant change in the vocal tract configuration, the linear predictor will accurately model the vocal tract, but will not accurately model the excitation. For this reason, the linear prediction coefficients will describe the components of the speech due to the slow changing, "predictable" configuration of the vocal tract, while the error between $s(m)$ and $\hat{s}(m)$ will be primarily due to the less regular excitation component. Linear prediction is therefore used in speech analysis to obtain characterizations of the vocal tract and excitation components of speech.

The number of samples $s(m)$ used in obtaining a set of predictor coefficients will typically be between 100 and 400, corresponding to 10-20 milliseconds of speech, depending on the rate at which the original speech signal was sampled. The linear prediction analysis will therefore be performed between 50 and 100 times for a second of speech. Typical values of p, the number of terms used in the approximation of $s(m)$, will be between 6 and 25 [12].

Different assumptions about the range of m in (1) yield different formulations of linear prediction, on which different techniques to solve the system of equations in (2) can be used. The assumption that $s(m)$ is 0 outside the interval $0 \leq m < M$, for some M, results in the autocorrelation method [12,16]. This method possesses some desirable computational properties, but the assumption that $s(m) = 0$ outside the given interval is not, in general, true. The assumption that m is to range over a fixed interval $p \leq m < M$, but that the signal may be non-zero outside that interval (in particular, that $s(l)$, $0 \leq l < p$, need not be zero) results in the covariance method [2,12,16]. This is a more accurate model of speech, but the solution of the equations in (2) is more expensive than with the autocorrelation method. Both methods are widely used.

III. SIMD ALGORITHM ATTRIBUTES

Under the assumptions of the autocorrelation method, the predictor coefficients, ($a(k)$'s), can be obtained by solving the system of equations [11,16]:

$$\sum_{k=1}^{p} a(k)R(|i-k|) = R(i) \qquad 1 \leq i \leq p \tag{3}$$

where the $R(i)$'s are the short-time autocorrelation functions:

$$R(i) = \sum_{m=0}^{M-1-i} s(m)s(m+i) \qquad 0 \leq i \leq p$$

Equivalently, the predictor coefficients can be found by solving the matrix equation:

$$\mathcal{R}\underline{a} = \underline{R} \tag{4}$$

where $\underline{R}$ and $\underline{a}$ are the p-element column vectors of elements $R(i)$ and $a(i)$ respectively for $1 \leq i \leq p$, and $\mathcal{R}$ is the p by p matrix in which $\mathcal{R}(i,k) = R(|i-k|)$, $0 \leq i,k < p$. $\mathcal{R}$ is a Toeplitz matrix, i.e., it is symmetric, with all elements in each diagonal being identical.

Obtaining the predictor coefficients using the autocorrelation method consists of two steps: computation of the $R(i)$'s and solution of the system of equations in (3). SIMD algorithms to compute the $R(i)$'s in $N \geq M$ PEs were given in [24]. Durbin's method [16] is an iterative serial technique for solving a system of equations involving a Toeplitz matrix. A parallel algorithm based on Durbin's method for computing the $a(k)$'s given the $R(i)$'s is presented in [26]. The relative complexities of the serial and parallel algorithms are shown in Table 1.

The two algorithms to compute the $R(i)$'s use $N \geq M$ PEs. The interconnection functions required by the first algorithm are the Shift_{-1} function and the Cube interconnection functions. The Shift_{-1} is one of the uniform shift functions, where in general the $\text{Shift}_{\pm d}$ function is defined as the interconnection function which transfers data from PE i to PE $(i \pm d) \bmod N$, $0 \leq i < N$. The Cube functions [17] consist of n interconnection functions, defined for $0 \leq i < n$ as

$$\text{Cube}_i(p_{n-1} \ldots p_i \ldots p_0) = p_{n-1} \ldots \bar{p}_i \ldots p_0$$

where $p_{n-1} \ldots p_0$ is the binary representation of a PE address and $\bar{\ }$ denotes complement. The second $R(i)$ algorithm employs the n Cube functions and a Broadcast, defined to be the transfer of a data item from one PE to all PEs. The parallel algorithm based on Durbin's method uses $N \geq p$ PEs and requires the $\log_2 p$ Cube functions, the $\text{Shift}_{\pm d}$ functions for $1 \leq d < p/2$, and $p/2$ Exch_d functions, defined for $p/2 \leq d < p$ as

$$\text{Exch}_d(j) = \begin{cases} j+d & 0 \leq j < p-d \\ j & p-d \leq j < d \\ j-d & d \leq j < p \end{cases}$$

In Section IV, the ability of various interconnection networks to execute these interconnection functions is discussed.

Under the assumptions of the covariance method, the predictor coefficients can be obtained by solving the system of equations [2,12]:

$$\sum_{k=1}^{p} a(k)c(k,i) = -c(0,i) \qquad 1 \leq i \leq p \tag{5}$$

The $c(k,i)$'s represent a covariance matrix

$$c(i,j) = \sum_{m=p}^{M-1} s(m-i)s(m-j) \qquad 0 \leq i,j \leq p \tag{6}$$

where samples $s(0)$ through $s(M-1)$ are available in the speech segment. Solution of the system of equations in (6) is equivalent to solving the matrix equation:

$$\mathcal{C}\underline{a} = -\underline{c} \tag{7}$$

where $\underline{c}$ is the p-element column vector of elements $c(0,i)$, $1 \leq i \leq p$, and $\mathcal{C}$ is the p by p covariance matrix in which $\mathcal{C}(k,i) = c(k,i)$, $1 \leq i,k \leq p$. Solution for the a(k)'s in the covariance method consists of two steps: computation of the c(i,j)'s and solution of the system of equations in (5) or (7). Parallel algorithms to compute the c(i,j)'s, perform the matrix inversion, and compute the matrix-vector product needed to solve for the a(k)'s in equation (7) are given in [26]. Complexities of the serial and SIMD algorithms are shown in Table 1.

The algorithm to compute the c(i,j)'s uses $N \geq M$ PEs. The interconnection functions required are the Shift_{-1} and $\text{Shift}_{-(M-p)}$ functions, the Shift_{+i} functions for $1 \leq i \leq p$, and the set of n "Minus2^i" functions. The Minus2^i functions are defined for $0 \leq i < n$ as

$$\text{Minus2}^i\,(j) = (j - 2^i) \bmod N.$$

The matrix inversion algorithm uses $N \geq p$ PEs, and requires a Broadcast, the $\text{Shift}_{\pm d}$ functions for $1 \leq d < p$, and the set of $\log_2 p$ "Plus2^i" functions defined for $0 \leq i < n$ as

$$\text{Plus2}^i(j) = (j + 2^i) \bmod N.$$

The algorithm to compute the matrix-vector product uses $N \geq p$ PEs, and requires a Broadcast.

IV. MACHINE REQUIREMENTS

From the algorithms developed, it is possible to infer some characteristics of an SIMD machine designed to perform LPC analysis efficiently. The machine should have at least M PEs, needed for the fast computation of autocorrelation or covariance coefficients. A submachine of size p will be used to solve for the predictor coefficients. For the autocorrelation method, the Cube, Shift, and Exch functions must be performed. A Broadcast may also be needed. For the covariance method, the Plus2^i, Minus2^i, and Shift functions and a Broadcast are used. It can be shown [26] that each of these functions and the Broadcast can be performed in a single pass through a number of multistage networks, including the data manipulator [5], augmented data manipulator [18], generalized cube (with four-function interchange boxes) [22], and omega [9] networks. In one pass, the indirect binary n-cube [15] can perform each of the functions; the effect of a Broadcast can be achieved in at most n passes, using a transfer pattern similar to that of recursive doubling. The STARAN flip network [3] can perform the Cube, Plus2^i, and Minus2^i functions in a single pass, the Shift functions and Broadcast in at most n passes, and the Exch in at most 2n passes. A single-stage shuffle-exchange network [27] can perform each of these data transfers in at most n shuffles and n exchanges. More details are in [26].

V. CONCLUSIONS

Many large-scale multimicroprocessor systems which can operate in the SIMD mode of parallelism have been proposed [e.g., 10, 13, 15, 20, 21]. This paper explores the use of SIMD parallelism

Table 1. Summary of serial and SIMD algorithm complexities for linear prediction computations.

		multiplications	additions	inter-PE transfers	types of transfers
AUTOCORRELATION METHOD:					
R(i)'s	serial	$M(p+1)-p(p+1)/2$	$M(p+1)-p(p+1)/2$	-	-
	SIMD - M PEs*	$p+1$	$(p+1)\log_2 M$	$\log_2 M(p+1)+p$	Shift_{-1}, Cube
	or SIMD - M PEs*	$\log_2 M+1$ †	$2(\log_2 M+1)$ †	$\log_2 M$	Cube, Broadcast
a(k)'s	serial	p^2+2p	p^2+p	-	-
	SIMD - p PEs	$5p-2$	$p\log_2 p+3p+\log_2 p$	$p\log_2 p+p/2+\log_2 p-1$	Cube, Shift, Exch
COVARIANCE METHOD:					
c(i,j)'s	serial	$Mp+p^2-p$	$Mp+p^2-p$	-	-
	SIMD - M PEs	$p+1$	$(p+1)(\log_2 M+1)$	$\log_2 M(p+1)+3p+1$	Shift, Minus2^i
matrix inversion	serial	p^3+p-1	p^3-2p^2+p	-	-
	SIMD - p PEs	$2p^2$	$2p^2-p$	$3p^2-2p+2\log_2 p+1$	Broadcast, Shift, Plus2^i
matrix-vector product	serial	p^2	p^2	-	-
	SIMD - p PEs	p	p	p	Broadcast

* presented in [24] † complex arithmetic

for the applications area of speech processing by discussing parallel algorithms to perform linear predictive coding analysis. From the algorithms, design criteria for an SIMD machine for speech analysis applications can be derived.

The approach taken to studying the applicability of SIMD machines to linear predictive coding has been to develop and analyze parallel linear prediction algorithms. On one hand, these analyses provide direct information towards evaluating the usefulness of parallel computers for speech processing and related areas. At the same time, however, they contribute to the more general body of knowledge concerning parallel processing. By developing algorithms for a general model of a parallel system, insight can be gained into a number of aspects of parallel processing. The algorithms can be used to define specific architectural features, such as the number of processors needed/useful for a class of problems, the sizes of memories required, interconnection network capabilities needed, and the type of processing capability required in each processor. Thus, applications studies such as this provide information for both the speech processing and the parallel processing researcher.

REFERENCES

[1] A. V. Ashajayanthi, S. Rajaram, N. Viswanadham, "A parallel processor for real-time speech signal processing." 1979 IEEE Int. Conf. Acoust. Speech Signal Proc., Apr. 1979, pp. 868-871.

[2] B. S. Atal, S. L. Hanauer, "Speech analysis and synthesis by linear prediction of the speech wave," J. Acoust. Soc. Am., Vol. 50, Aug. 1971, pp. 637-655.

[3] K. E. Batcher, "The flip network in STARAN," 1976 Int. Conf. Parallel Proc., Aug. 1976, pp. 65-71.

[4] G. E. Box and G. M. Jenkins, Time Series Analysis Forecasting and Control. San Francisco: Holden-Day, 1970.

[5] T. Feng, "Data manipulating functions in parallel processors and their implementations," IEEE Trans. Comput., Vol. C-23, Mar. 1974, pp. 309-318.

[6] M. J. Flynn, "Very high-speed computing systems," Proc. IEEE, Vol. 54, Dec. 1966, pp. 1901-1909.

[7] R. E. Kalman, "A new approach to linear filtering and prediction problems," Trans. ASME, J. Basic Eng., Series D82, pp. 35-45, 1960.

[8] P. M. Kogge, H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," IEEE Trans. Comput., Vol. C-22, Aug. 1977, pp. 786-792.

[9] D. H. Lawrie, "Access and alignment of data in array processor," IEEE Trans. Comput., Vol. C-24, Dec. 1975, pp. 1145-1155.

[10] G. J. Lipovski, A. Tripathi, "A reconfigurable varistructure array processor," 1977 Int. Conf. Parallel Proc., Aug. 1977, pp. 165-174.

[11] J. Makhoul, "Linear prediction: a tutorial review," Proc. IEEE, Vol. 63, Apr. 1975, pp. 561-580.

[12] J. D. Markel, A. H. Gray, Jr., Linear Prediction of Speech. NY: Springer-Verlag, 1976.

[13] G. J. Nutt, "Microprocessor implementation of a parallel processor," 4th Symp. Comp. Arch., Mar. 1977, pp. 147-152.

[14] M. C. Pease, "Matrix inversion using parallel processing," JACM, Vol.14, Oct. 1967, pp.757-764.

[15] M. C. Pease, "The indirect binary n-cube microprocessor array," IEEE Trans. Comput., Vol. C-26, May 1977, pp. 458-473.

[16] L. R. Rabiner, R. W. Schafer, Digital Processing of Speech Signals. Englewood Cliffs, NJ: Prentice-Hall, 1978.

[17] H. J. Siegel, "Analysis techniques for SIMD machine interconnection networks and the effects of processor address masks," IEEE Trans. Comput., Vol. C-26, Feb. 1977, pp. 153-161.

[18] H. J. Siegel, "Interconnection networks for SIMD machines," Computer, Vol. 12, June 1979, pp. 57-65.

[19] H. J. Siegel, "A model of SIMD machines and a comparison of various interconnection networks," IEEE Trans. Comput., Vol. C-28, Dec. 1979, pp. 907-917.

[20] H. J. Siegel, P. T. Mueller, Jr., H. E. Smalley, Jr., "Control of a partitionable multimicroprocessor system," 1978 Int. Conf. Parallel Proc., Aug. 1978, pp. 9-17.

[21] H. J. Siegel, L. J. Siegel, R. J. McMillen, P. T. Mueller, Jr., S. D. Smith, "An SIMD/MIMD multimicroprocessor system for image processing and pattern recognition," IEEE Conf. Pattern Recog. Image Proc., Aug. 1979, pp. 214-224.

[22] H. J. Siegel, S. D. Smith, "Study of multistage SIMD interconnection networks," 5th Symp. Comp. Arch., Apr. 1978, pp. 223-229.

[23] L. J. Siegel, "A procedure for using pattern classification techniques to obtain a voiced/unvoiced classifier," IEEE Trans. Acoust. Speech Signal Proc., Vol. ASSP-27, Feb. 1979, pp. 83-89.

[24] L. J. Siegel, "Parallel processing algorithms for linear predictive coding," 1980 IEEE Int. Conf. Acoust. Speech Signal Proc., Apr. 1980, pp. 960-963.

[25] L. J. Siegel, P. T. Mueller, Jr., H. J. Siegel, "FFT algorithms for SIMD machines," 17th An. Allerton Conf. Communication, Control, Computing, Oct. 1979, pp. 1006-1015.

[26] L.J. Siegel, H.J. Siegel, R.J. Safranek, M.A. Yoder, "Linear Predictive Coding Algorithms for SIMD Machines," Purdue Univ., Elec.Engr. Tech. Rept., in preparation.

[27] H. S. Stone, "Parallel processing and the perfect shuffle," IEEE Trans. Comput., Vol. C-20, Feb. 1971, pp. 153-161.

[28] N. Wiener, Extrapolation, Interpolation and Smoothing of Stationary Time Series with Engineering Applications. Cambridge, MA: M.I.T. Press, 1949.

A Note on Pipelining a Mesh Connected Multiprocessor for Finite Element Problems by Nested Dissection.

Dennis Gannon[a]
Department of Computer Science
University of Illinois, Urbana-Champaign
Urbana, Illinois, 61801

Abstract -- A pipelined version of the parallel Givens Reduction algorithm of Sameh and Kuck is developed that runs on a quad-connected p^2 multiprocessor array. With the addition of a shuffle transformation, this permits the solution of the n by n system $Ax = f_i$ $i = 1,..,r$ in time $C(n/p)^2(n+p+r)$. When A has band width p, the time is $C(n/p)(p+r)$. This is used as the kernel for a pipelined nested dissection solver for the n^2 by n^2 algebraic systems that arise in finite element problems on n by n grids. With an n by n mesh-connected multiprocessor, the method runs in time $C(n+r\log(n))$.

INTRODUCTION

The method of Nested Dissection by George [1] has been shown by Liu [2] to be capable of solving the sparse $n^2 x n^2$ system of equations $Ax=f$ on an n by n finite element grid in time $O(n)$ with $O(n^2)$ processors. Like many of the now classical parallel algorithms, this theoretical result did not consider the problem of inter processor communication. Kung and Leiserson [3] have shown that, by a "wavefront" or "systolic" process, the solution of a dense n by n system of linear equations can be pipelined on a hex-connected array of $O(n^2)$ processors which carry out standard Gaussian elimination. The result is an explicit scheme of computation that includes inter processor communication within the standard O(n) time bound. More recently, Kuhn [4] has demonstrated a class of program transformations that map a great number of algorithms onto specific architectures with the property that the resulting execution time is of the same complexity as the classical asymptotic time bound.

Among the more stable dense system solvers is the method of Parallel Givens rotations by Sameh and Kuck [5]. As has been observed by Kung and Leiserson, this method can be easily be adapted to this type of pipelined array environment. The purpose of this note is to develop one such implementation of the Sameh-Kuck algorithm that runs on a quad-connected multiprocessor array, and to use this as the kernel for a nested dissection solver that can be pipelined on a mesh-connected multiprocessor such as the N.A.S.A. Langley Finite Element Machine [6].

The following paragraphs will first treat the Sameh-Kuck algorithm as well as several other useful operations that can be executed on a square array mesh-connected multiprocessor. These operations will constitute a set of primitives from which the nested dissection solver will be constructed in the third section. The last section will discuss the significance as well as the shortcomings of such an approach.

The Givens Rotation

Consider the problem of finding the solutions to the system $Ax = f_i$, for load vectors f_i, $i = 1..r$. The approach taken by Sameh and Kuck is to reduce the matrix A to upper triangular form by the application of a carefully chosen sequence of rotations. It is shown in [5] that this sequence of rotations can be blocked in such way that all rotations within a given block may be executed in parallel. Once A is in upper triangular form, an algorithm such as the column sweep method described in Kuck [7] can by used to complete the back solve process to obtain the solution vectors x.

The basic Givens rotation of two rows i,j of the matrix (A, f) is given by

$$\begin{bmatrix} a'_{i1}\cdot\cdot a'_{in} & f'_{i1}\cdot\cdot f'_{ir} \\ 0 \quad \cdot\cdot a'_{jn} & f'_{j1}\cdot\cdot f'_{jr} \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a_{i1}\cdot\cdot a_{in} f_{i1}\cdot\cdot f_{ir} \\ a_{j1}\cdot\cdot a_{jn} f_{j1}\cdot\cdot f_{jr} \end{bmatrix}$$

Assume that the i,j rows of the (A,f) array are made available to a pair of connected processors every C time steps. After receiving the first column, the processor pair must compute the rotation coefficients given by

$$d = (a_{i1}^2 + a_{j1}^2)^{1/2}, \quad c = \frac{a_{i1}}{d}, \quad s = \frac{a_{j1}}{d}$$

and then output $(d, 0)^t$. Assume the pair of processors can communicate one item of data in either direction in one unit of time. The algorithm for the pair of processors consists of the following sequences

---

[a] Currently at Department of Computer Science, Purdue University, West Lafayette, Indiana. Supported by the Department of Energy grant ENERGY/EY-76-S-02-2383 and an IBM Graduate Fellowship.

In processor P1

```
input $a_{i1}$
$d := a_{i1}^2$
send d to P2
recv c from p2
$d := d + c$
$d := sqrt(d)$
$c := a_{i1}/d$
send c to P2
recv s from P2
output d
```

In processor P2

```
input $a_{j1}$
$d := a_{j1}^2$
recv c from P1
send d to P1
$d := d + c$
$d := sqrt(d)$
$s := a_{j1}/d$
recv c from P1
send s to P1
output 0
```

At succeeding blocks of C time steps the pair computes and outputs

$$a'_{ik} = ca_{ik} + sa_{jk} \quad k = 2,..,n,$$

$$f'_{ik} = cf_{ik} + sf_{jk} \quad k = 1,..,r$$

in processor 1, and

$$a'_{jk} = -sa_{ik} + ca_{jk} \quad k = 2,..,n,$$

$$f'_{jk} = -sf_{ik} + cf_{jk} \quad k = 1,..,r$$

in processor 2. From the first stage of computation, it can be seen that C = 9 + one square root. At the expense of a more complex instruction sequence for each processor, the square root free method of Hammerling [8] can be implemented (In this case, in order to prevent underflow the processor pair must make a test, and possibly switch the computational roles of the two processors).

Figure 1 illustrates the overall structure for the pipelined algorithm. It is assumed the matrix and right hand side vectors (A, F) are stored by rows in a column of memory units along the right hand side of the array of processors. One column is accessed every C time units. The data marches, column by column, through the array of processors. The pattern of zeros introduced into the matrix is given by the numbered elements. When matrix element x reaches processor x the reduction sequence is started. The processor x and the processor marked by * above x carry out the programs described above for P1 and P2 respectively. The unmarked processors simply execute a receive, wait, and transmit operation on all data they receive. In order to keep the rows synchronized as they move through the array, the wait cycle should be equal to C. As in figure 1, an n by n matrix is reduced to upper triangular form after being passed through a triangle imbeded in a rectangular array of 2n-3 by n quad connected processors.

The total time to preform the reduction on all n columns of A plus the r columns f_i i = 1,..,r is C(3n+r-3). The reduction operation is equivalent to a left multiply by an orthogonal matrix Q. If we let the upper triangular matrix be represented by U = QA, and let $f'_i = Qf_i$ i = 1,..,r, the problem has been reduced to the solution of the system

$$Ux = f'_i \qquad i = 1,..,r.$$

A second pass through the processor array can be used to execute a column sweep back solve algorithm as illustrated in figure 2. The initial storage scheme is identical to the reduction sweep. First the entries of U are moved into the array with a simple pipelined broadcast along the rows of the array. In the computational phase of the algorithm the rows of the f_i vectors are advanced into the computational array. The movement is not by columns but is stagered so that the first entry of the last row is entered first, then the first entry of row n-1 and the second entry of row n. In figure 2, the numbers beside the arrows indicate the order of the data movement for the first column as it passes through the array. The activities of the processors fall into three categories. The processors containing matrix diagonal elements receive a data item from the right and divide by the diagonal element. The result is an x value and is transmitted both upward and to the left. The processors above the diagonal receive an x value from below which they

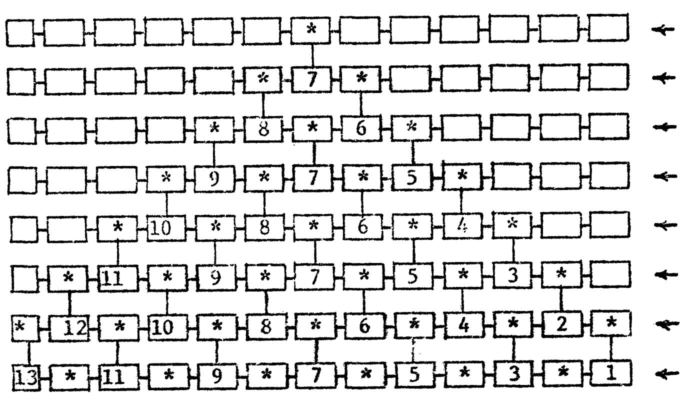
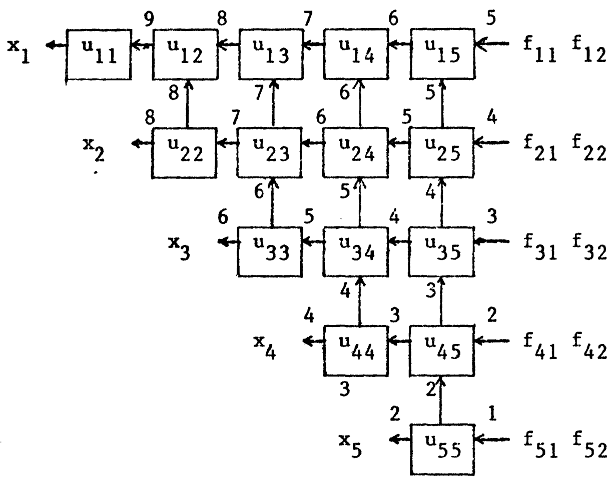


Figure 1. Givens Reduction Array.

$$\begin{bmatrix} . & . & . & . & . & . & . & . \\ 7 & . & . & . & . & . & . & . \\ 6 & 8 & . & . & . & . & . & . \\ 5 & 7 & 9 & . & . & . & . & . \\ 4 & 6 & 8 & 10 & . & . & . & . \\ 3 & 5 & 7 & 9 & 11 & . & . & . \\ 2 & 4 & 6 & 8 & 10 & 12 & . & . \\ 1 & 3 & 5 & 7 & 9 & 11 & 13 & . \end{bmatrix}$$

Input matrix.

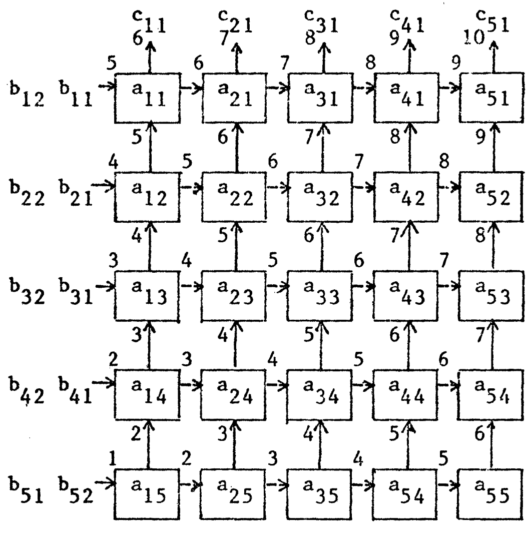
multiply by the contained matrix entry, and subtract from the value received from the right. The difference is passed to the left and the x value is sent on upward. Processors below the diagonal receive x values which they transmit to the left. If the vectors f_i i=1,..,r is viewed as an n by r matrix F, this process computes $U^{-1}F$ in time C(3n+r). In this case C, the maximum time for any processor to complete one cycle of its task, is about 4 instructions.



x_1 x_2 x_3 x_4 x_5
u_{11} u_{12} u_{13} u_{14} u_{15}
u_{22} u_{23} u_{24} u_{25}
u_{33} u_{34} u_{35}
u_{44} u_{45}
u_{55}
f_{11} f_{12} f_{21} f_{22} f_{31} f_{32} f_{41} f_{42} f_{51} f_{52}

Figure 2. Back solve.

In a similar manner, one can compute the matrix product AB for an n by n matrix A, and an n by r matrix B. This is illustrated in figure 3. In this case, matrix A is initially stored along the top row and the matrix B is stored along a perpendicular edge. During the first phase the matrix A is moved into the array. In the second phase, the matrix B is piped through the array.



c_{11} c_{21} c_{31} c_{41} c_{51}
b_{12} b_{11} b_{22} b_{21} b_{32} b_{31} b_{42} b_{41} b_{51} b_{52}
a_{11} a_{21} a_{31} a_{41} a_{51}
a_{12} a_{22} a_{32} a_{42} a_{52}
a_{13} a_{23} a_{33} a_{43} a_{53}
a_{14} a_{24} a_{34} a_{44} a_{54}
a_{15} a_{25} a_{35} a_{54} a_{55}

Figure 3. C = AB

The result of the inner product is accumulated as the data moves vertically, and the values of B are piped horizontally. The result of the product appears along the top row. It should be noted that the above two procedures are simple adaptations of ideas that are not at all new and fall under the general heading of 'column sweep' or 'wave front' algorithms.

For the purpose of a concise description of the dissection algorithm we shall adopt the following notation. Given an n by n array of processors connected vertically, horizontally, and diagonally to their neighbors, and an n by n matrix A and an n by r matrix B, define

GR(A; B) = the Givens reduction process of forming (U, QB).

BS(U; B) = the back solve process to form $U^{-1}B$.

MP(A; B) = the matrix multiplication process.

The notation (A, B) will describe the n by n+r matrix formed by the concatenation of the matrices A and B. If P is a row or column of processors the notation A_P shall mean that A is stored in the memories of P one row per processor. The above processes all share the property that one starts with data along one edge of the processor array and the results are produced along some other edge. The dissection algorithm shall have occasion to require the results of an operation to lie in the same processor memory set as those in which the data originated. For this reason define the process

$A_R := MV(A_P)$ = the pipelined copy of A from processor set P to processor set R.

Note that the Givens reduction, as defined above, does not exactly correspond to the algorithm described previously. The difference lies in the size of the processor array required. The basic reduction process requires a 2n by n array. Rather than treat the problem of reducing the requirement to an n by n array, consider the more general limited processor problem of reducing an n by n matrix with a p by p processor array with p < n. For simplicity, assume that p divides n. The more general case is not difficult to develop. The process of reducing a p by n matrix to upper triangular form by a height p processor array will be called p-reduction. Observe that by using p^2 processors divided first into two height p/2 triangles, one can reduce a pxn matrix to the form illustrated in figure 4. This is simply two reduced p/2 by n matrices stacked vertically. The resulting process will be called a 2(p/2)-reduction. The problem is then how to complete a 2(p/2)-reduction to a p-reduction. The solution lies in the use of a shuffle transformation to rearrange the rows of the matrix so that a second pass through the processor array will complete the task. For p even, the

$$\begin{bmatrix} a'_{11} & a'_{12} & a'_{13} & \cdots & a'_{1n} \\ 0 & a'_{22} & a'_{23} & \cdots & a'_{2n} \\ 0 & 0 & a'_{33} & \cdots & a'_{3n} \\ 0 & 0 & 0 & \cdots & a'_{4n} \\ a'_{51} & a'_{52} & a'_{53} & \cdots & a'_{5n} \\ 0 & a'_{62} & a'_{63} & \cdots & a'_{6n} \\ 0 & 0 & a'_{73} & \cdots & a'_{7n} \\ 0 & 0 & 0 & \cdots & a'_{8n} \end{bmatrix} \qquad \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ a_{41} & a_{42} & a_{43} & \cdots & a_{4n} \\ a_{51} & a_{52} & a_{53} & \cdots & a_{5n} \\ a_{61} & a_{62} & a_{63} & \cdots & a_{6n} \\ a_{71} & a_{72} & a_{73} & \cdots & a_{7n} \\ a_{81} & a_{82} & a_{83} & \cdots & a_{8n} \end{bmatrix}$$

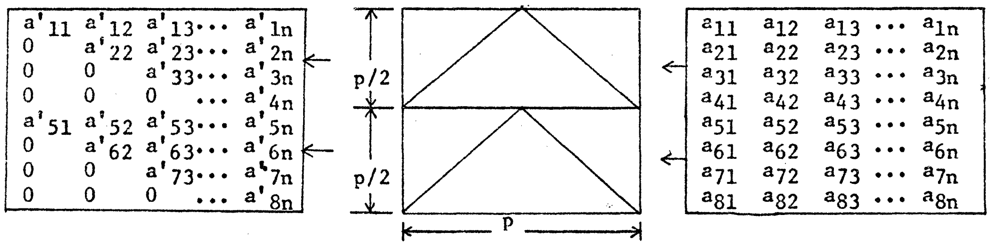


Figure 4. The 2(p/2)-reduction.

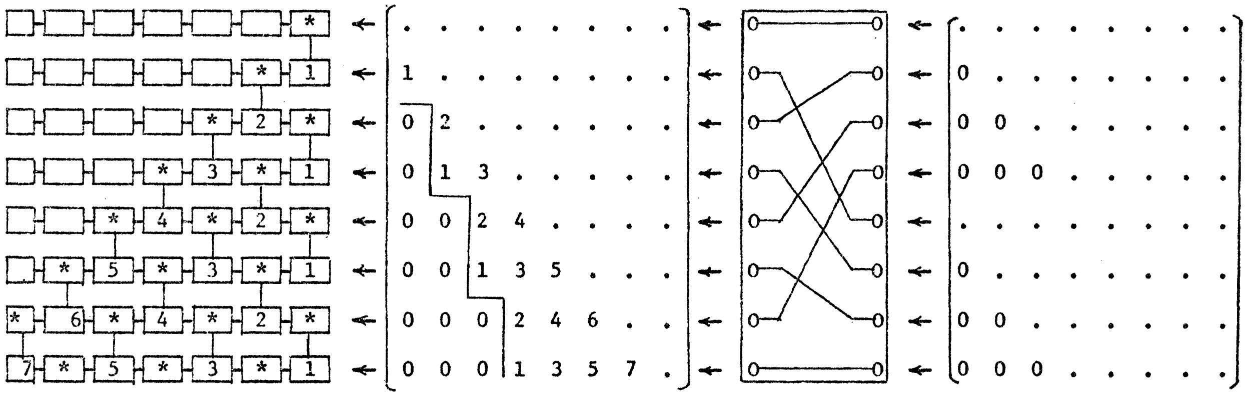


Figure 5.

shuffle permutation is given by

$$sh(i) = 2i \quad 0 \leq i < \frac{p}{2}$$

$$sh(i) = 2i-p+1 \quad p > i \geq \frac{p}{2}.$$

By applying a shuffle transformation, the two (p/2)-reduced arrays are merged to a matrix that is "half" reduced in the sense that the existing zeros correspond to those introduced by the right half of the 2p by p processor triangle studied above. Hence, the remainder of the reduction can be completed by using the left half of the height p processor triangle (see figure 5). Employing the block elimination scheme illustrated in figure 6 for the case p/2 = n/4, the n by n system can be n-reduced in time $2C\frac{n^2}{p^2}(p+n+r)$ using p^2

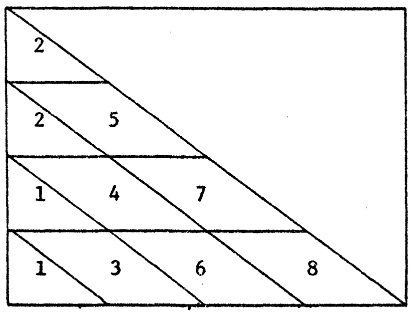


Figure 6. Block Reduction.

processors. The algorithm first completes n/p 2(p/2)-reductions and the remainder of the blocks are reduced by the shuffle half reduction scheme. The order of elimination of the various block is given by the numbering in figure 6. A similar analysis gives the case of a band width p system in time $Cn(3+\frac{2r}{p})$.

To complete the mapping of this process onto the p by p processor array, it suffices to observe that the shuffle permutation is easily pipelined. This is illustrated in figure 7 for the case of p = 8.

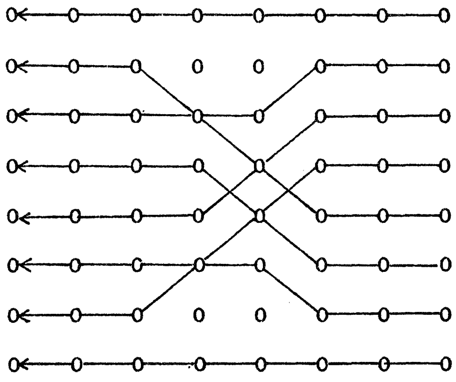


Figure 7. Pipelined Shuffle.

NESTED DISSECTION

The nested dissection algorithm is simply Gaussian elimination based on a very special ordering of the variables in the system. Initially, one starts with a finite difference or finite element problem defined on a n by n grid of nodes. The variables of the system correspond to the solution values at the nodes of the grid. The n^2 by n^2 matrix of coefficients in the system of equations corresponds to the pairwise interactions of the nodes on the n by n grid. The main idea behind the algorithm can be described as follows. First, from the system of equation, we eliminate all variables corresponding to the interior nodes of the square grid. What remains is a much smaller system involving only the nodes on the boundary. Once the smaller system has been solved, the boundary values obtained are used in a back solve process to obtain the values for the interior node variables. The important feature of the algorithm is the way in which we eliminate the interior node variables. Divide the n by n grid into 4 grids of size n/2 by n/2. Eliminate the interiors of each of these smaller grids (in parallel) and then eliminate the variables for the nodes on the cross that quartersected the grid. The elimination of the interior nodes for the subgrids is the same process, recursively applied. By repeatedly subdividing, all interior nodes are eventually seen to lie on some subdiving cross. In terms of the Gaussian elimination process, subdiagonal segments of the columns in the matrix corresponding to the smallest subdividers are eliminated in parallel first. Then the subdiagonal matrix elements corresponding the the next smallest are eliminated. The process continues until all subdiagonal elements for all interior nodes have been eliminated. The paragraphs that follow will give a more rigorous description of this process.

The dissection algorithm is one of the very large class of algorithms based on a recursive divide-and-conquer approach. These algorithms start with a problem of size 2^k and generate 2 problems of size 2^{k-1}, then 4 problems of size 2^{k-2}, and so on. Previous studies of parallel nested dissection have been to consider the application of the algorithm to a machine like the CRAY 1. The unique feature of a vector architecture is that only a few vectors can usually be processed in parallel, and one wants these to be as long as possible. Unless one defines the vectors used in the computation to be cross sections of the components of the subproblems, the recursive algorithms very quickly generate 2^k vectors of length 2. Calahan has proposed a method [9] based on "generalized vectors" to deal with this problem.

A second approach is to consider termination of the algorithm before the vectors get too short and then switch to a second method. The transition point depends upon various factors such as the pipe start-up time. George, Poole, and Voigt [10] have studied this trade off in great detail and have developed several very attractive algorithms.

An architecture of the MIMD class provides a very flexible environment for the parallel implementation of algorithms based on the recursive divide-and-conquer technique. In its simplest form, the N.A.S.A. Langley finite element machine is an array of n^2 microprocessors that correspond to the nodes in a finite difference grid and are interconnected in a manner that corresponds to a nine point finite difference operator, i.e. each processor is connected to its eight nearest neighbors. In the local memory of each processor we can easily generate the rows of the stiffness matrix A and the load vectors f that correspond to that node of the grid. For simplicity, let $n=2^k+1$ for some $k>0$. Number the nodes by (x,y) coordinate pairs on an integer lattice with (0,0) as the lower left hand corner. Let the processor at (x,y) be denoted by P(x,y). Let $a,b,s\geq 0$, and consider a (2^s+1) by (2^s+1) block with P(a,b) as the lower left hand corner. In order to describe the algorithm it is helpful to define certain sets of processors and variables associated with such a block. Define

$$PX_1^s(a,b) = \{p(a+2^{s-1},\ b+j) \mid 1\leq j\leq 2^s-1\}$$

$$PX_2^s(a,b) = \{p(a+j,\ b+2^{s-1}) \mid 1\leq j\leq 2^s-1\ \ j\neq 2^{s-1}\}$$

$$PY_1^s(a,b) = \{p(a+i,\ b+2^s) \mid 1\leq i\leq 2^s\}$$

$$PY_2^s(a,b) = \{p(a+2^s,\ b+i) \mid 0\leq i\leq 2^s-1\}$$

$$PY_3^s(a,b) = \{p(a+i,\ b) \mid 0\leq i\leq 2^s-1\}$$

$$PY_4^s(a,b) = \{p(a,b+i) \mid 1\leq i\leq 2^s\}.$$

Let PN(a,b) be the set of processors in this block not contained in the above sets. In short, the set PX_1^s is the set of processors on the vertical bisector. PX_2^s is the set along the horizontal bisector less the center processor, and PY_i^s $i=1,..,4$ is the set corresponding to the four outer edges of the block. This is illustrated in figure 8. Let N, X_i, Y_j with $i=1,2$ and $j=1,..,4$ be the set of indices of the corresponding variables where the superscript s and base address (a,b) are dropped when the context is clear. Let Z be the set of indices corresponding to the columns of f_i $i=1,..,r$ and define A(H;I) to be the subblock of the matrix (A, F) for rows H and columns I. Special blocks of interest will be

$B_{ij} = A(X_i, X_j) \quad 1\le i,j\le 2,$

$C_i \; = A(X_i;\ Y_j j=1,..,4,\ Z) \quad i=1,2,$

$D_{ij} = A(Y_i;\ X_j) \quad 1\le i\le 4,\ 1\le j\le 2,$

$E_i \; = A(Y_i; Y_j j=1,..,4,\ Z) \quad 1\le i\le 4.$

The main idea behind the implementation constructed here is to map the dissection algorithm onto the array of processors so that when the grid is recursively quartered, the processors in each quadrant are assigned the task of completing the computation associated with the variables in that quadrant subgrid.

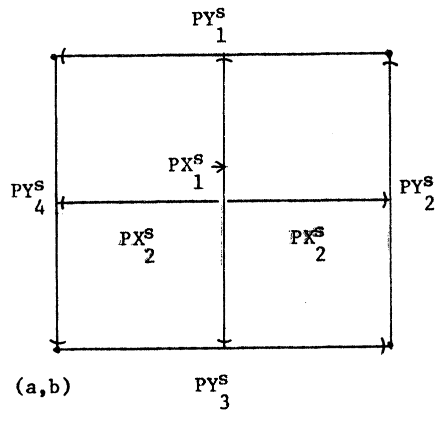
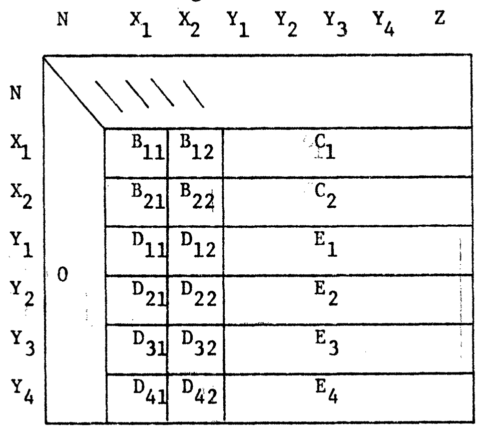


Figure 8. Quartered Grid.

The only problem to be solved is how to organize the computation of the recursive reduction process in a manner that avoids memory and processor contention for processors associated with the subdividing cross. In order to visualize the process, consider the structure of the matrix (A, F) in a block of size 2^s+1 by 2^s+1 after the elimination of all bisectors of size 2^{s-1} and smaller (corresponding to the elimination of all variables X^r_* for $r\le s-1$). The structure of the matrix is shown in figure 9.



(A, F) Block Decomposition.
Figure 9.

The algorithm must reduce the matrix B_{ij} and eliminate the subdiagonal blocks D_{ik} for $i,j=1,2$ and $k=1,..,4$. The nested dissection reduction of the interior variables of the grid can be described in the following algorithmic form.

```
Procedure NDR(s,a,b)
    Begin
      if s>0 pardo
          call NDR(s-1,a,b);
          call NDR(s-1,$a+2^{s-1}$,b);
          call NDR(s-1,a,$b+2^{s-1}$);
          call NDR(s-1,$a+2^{s-1}$,$b+2^{s-1}$);
          odpar;

      /* move the $B_{ij}$ rows to the outer
         edges of the array */
      $(B_{11},B_{12},C_1)_{PY^s_2(a,b)}$ :=MV$(B_{11},B_{12},C_1)$;
      $(B_{21},B_{22},C_2)_{PY^s_3(a,b)}$ :=MV$(B_{21},B_{22},C_2)$;

      /* reduce the $B_{11}$ matrix */
      $(U_{11},B_{12},C_1)$:=GR$(B_{11};B_{12},C_1)$;
      $(B'_{12},C'_1)_{PY^s_2(a,b)}$ :=BS$(U_{11};B_{12},C_1)$;
      $(B'_{12},C'_1)_{PY^s_1(a,b)}$ :=MV$(B'_{12},C'_1)$;

      /* if $X_2$ is nonempty eliminate
            $B_{21}$ and reduce $B_{22}$ */
      if $X_2$ <> 0 then
        $(B_{22},C_2)$:=MP$(B_{21};B'_{12},C'_1)+(B_{22},C_2)$;
        $(U_{22},C_2)$:=GR$(B_{22},C_2)$;
        $C'_2$:=BS$(U_{22};C_2)$
        $(C'_2)_{PY^s_2(a,b)}$ :=MV$(C'_2)$;
      fi;

      /* eliminate the $D_{ij}$ subblocks */
      for i=1 to 4 do
        $(D_{i1},E_i)_{PY^s_i(a,b)}$ :=MP$(D_{i1};B'_{12},C'_1)$
                                   +$(D_{i2},D_i)$;
        od;
      if $X_2$ <> 0 then
        for i = 1 to 4 do
          $(E_i)_{PY^s_i(a,b)}$ :=MP$(D_{i2};C'_2)$+$E_i$;
          od;
        fi;
    end;
```

The first block of the algorithm is the parallel recursive call, and hence constitutes the only possible point of processor or memory contention. In fact, each call to NDR must be seen as an allocation of a square block of processors to that called process, and the blocks defined here overlap along the common boundaries PY_i^s. The only possible way contention can occur is if some operation requires a pair of opposite outer edges. In this case, the same process executing on a neighboring block will demand access to the processors along the common boundary, and a conflict will develope. To see that no contention does develope, observe that at no point during the NDR procedure are there more than two perpendicular sets of edge processors involved in any one step of the process. The first set of moves are from the interior cross to an outer edge. The Givens reductions and the back solves start along an outer edge, but terminated short of the outer edge on the opposite side. The matrix multiplications employ a pair of perpendicular edges.

A time complexity estimate can be derived by observing that each operation in the procedure runs in time $O(2^s+r)$. Hence, if T_s is the time to complete a call to NDR(s,a,b) then there exists some constant C such that

$$T_s = T_{s-1}+C(2^s + r).$$

With $n = 2^k+1$, a call from the top level would be completed in time $2C(n+r\log(n))$.

A top level description of the algorithm would appear as

```
begin
  call NDR(k,0,0);

  call the limited processor
    Givens reduction
    to reduce the 4n by 4n block
    corresponding to the exterior
    variables $Y_i^k$ i=1,..4.

  call a limited processor
    back solve to obtain the 4nr
    variables $Y_i^k$.

  call NDBS(k,0,0);
end.
```

Once the subdiagonals associated with the interior nodes have been eliminated the reduction and solution of the problem corresponding to the boundary variables become a straight forward application of the limited processor techniques described earlier. The last call is to a nested dissection back solve procedure which is outlined as follows.

```
procedure NDBS(s,a,b)
  begin
    define $YV_i$, $XV_j$
      to be the $2^s$ by r solution
    values associated with $PY_i^s(a,b)$, $PX_j^s(a,b)$.
    define $F_i$ =$A(X_i;Z)$
           $C_{ij}$=$A(X_i;Y_j)$;
    for i=1 to 2 do
      for j=1 to 4 do
        $F_i$:=$F_i$-MP($C_{ij}$,$Y_j$)
      od;
    od;
    $XV_2$ := BS($U_{22}$;$F_2$);
    $F_1$ := $F_1$ - MP($B_{12}$,$XV_2$);
    $XV_1$ := BS($U_{11}$;$F_1$);
    $(XV_1)_{PX_1}$ := MV($XV_1$);
    $(XV_2)_{PX_2}$ := MV($XV_2$);
      if s > 0 then pardo
          call NDBS(s-1,a,b);

          call NDBS(s-1,a+$2^{s-1}$,b);
          call NDBS(s-1,a,b+$2^{s-1}$);
          call NDBS(s-1,a+$2^{s-1}$,b+$2^{s-1}$);
          odpar;
      end;
```

This procedure is of the same complexity as the reduction process and we see the time bound for the complete solution is of complexity $O(n+r\log(n))$.

CONCLUSION

The finite element machine was designed to solve finite element problems. While the most obvious application is to adapt iterative schemes, it has been show in the preceding sections that the architecture is rich enough to support the implementation of a direct solver that was also originally designed for finite element problems. The thrust of the preceding arguments has been to demonstrate that all interprocessor communication can be accounted for without changing the complexity of the time bound based on arithmetic operation counts. Furthermore we have shown that by pipelining the multiprocessor one can solve the system for several right hand sides at little additional cost ($Cr\log(n)$). But concerning the applicability of such a system, several important points should be raised.

While the algorithm is asymptotically very fast as a parallel direct solver applicable to a wide variety of problems, the size of the constant C may exceed the equivalent factor on a method like SOR or other iterative schemes by a factor of 10 or more. On the other hand, these other methods require more processing to determine factors such as relaxation parameters in order to run a their optimal rates. Furthermore, it is not clear how the iterative schemes can be effectively pipelined to solve a system for several right hand sides without simply multiplying the time bound by r.

A second interesting point concerns the numerical properties of the algorithm as it is presented here. The method is a blend of Givens reductions and direct elimination. While one would expect that this should work as well as direct elimination alone , more work is needed to verify this belief.

A third point of interest lies in the problem of generating code for a large multiprocessor. It is important to note that the procedures described above do not represent the code resident in any single processor. In fact, no two processors will execute the same code sequence. One approach to generating the code would be to generate calls to locally resident generic subroutines, such as the basic routines for the Givens reduction described earlier.

ACKNOWLEDGEMENTS.

The author would like to gratefully acknowledge many important suggestions and ideas of Ahmed Sameh, as well as several enlightening discussions with Robert Voigt.

REFERENCES

[1] A. George, "Nested Dissection of a Regular Finite Element Mesh", SIAM J. Numer. Anal. 10(1973), pp.345-363.

[2] J. W. H. Liu, The Solution of Mesh Equations on a Parallel Computer, Dept. of Computer Science, Univ. of Waterloo, Waterloo, Ontario, Report CS-78-19, 1978.

[3] H. T. Kung , C. E. Leiserson, "Algorithms for VLSI Processor Arrays", C. Mead and L. Conway, Introduction to VLSI Systems, Adison-Wesley, Reading, Ma., (1980). pp 271-292.

[4] R. Kuhn, Ph. D. Thesis, Dept. of Computer Science, University of Illinois, 1980.

[5] A. H. Sameh, D. J. Kuck, "On Stable Parallel Linear System Solvers", JACM 25 (Jan., 1978), pp. 81-91.

[6] H. Jordan, A Special Purpose Architecture for Finite Element Analysis , ICASE NASA Langley Research Center, Hampton, Virginia, ICASE Report no 78-9, (March, 1978).

[7] D. J. Kuck, The Structure of Computers and Computations, Vol. 1, John Wiley & Sons, Inc., New York, 1978.

[8] S. Hammerling, "A Note on Modifications to the Givens Plane Rotation", J. Inst. Math. Appl. 13(1974), pp. 215-218.

[9] D. A. Calahan, Complexity of Vectorized Solutions of Two-Dimensional Finite Element Grids , Systems Engineering Laboratory, Univ. of Michigan, Technical Report 91, (1975).

[10] A. George, W. G. Poole, Jr., R. G. Voigt, Analysis of Dissection Algorithms for Vector Computers, ICASE NASA Langley Research Center, Hampton, Virginia, ICASE Report no.76-17 (June, 1976), 58pp.

SOLVING LINEAR ALGEBRAIC EQUATIONS ON A MIMD COMPUTER

R. E. Lord, J. S. Kowalik, and S. P. Kumar
Department of Computer Science
Washington State University
Pullman, WA 99164

Abstract -- Two practical parallel algorithms for solving systems of dense linear equations are presented. They are based on Gaussian elimination and Givens transformations. The algorithms are numerically stable and have been tested on a MIMD computer.

Introduction

The problem of solving a set of linear algebraic equations is one of the central problems in computational mathematics and computer science. Excellent numerical methods solving this problem on uniprocessor systems have been developed, and many reliable and high quality codes are available for different cases of linear systems. On the other hand, the methods for solving linear equations on parallel computers are still in the conceptual stage, although some basic ideas have already emerged. The current state of the art in parallel numerical linear algebra is well described by Heller [3] and Sameh and Kuck [5].

Our investigation of methods for solving systems of dense linear equations on a MIMD computer includes Gaussian elimination with partial pivoting and Givens transformations. The first algorithm is commonly used to solve square systems of equations, the second produces orthognol decomposition used in several problems of numerical analysis including linear least squares problems. We focus our attention on the cases where the number of available processors is between 2 and $0(n)$, n being the number of linear equations. We take the view that it is not presently realistic to assume that $0(n^2)$ processors will be soon available to solve sizable sets of equations. To verify our analytic results we have used a parallel computer manufactured by Denelcor Co. [6]. This computer, called HEP (Heterogeneous Element Processor), is a MIMD machine of the shared resource type as defined by Flynn.

Gaussian Elimination

If we consider a step to be either a multiplication and a subtraction, or a compare and multiplication, then sequential programs for producing the LU decomposition of an $n \times n$ non-singular matrix requires $T_1 = (n^3/3) + 0(n^2)$ steps. The parallel method using $p = (n-1)^2$ processors and partial pivoting requires $T_p = 0(n \log n)$ steps. Thus the efficiency of such method for large n will be

$$E_p = \frac{T_1}{T_p \cdot p} = \frac{1}{0(\log n)} .$$

Even if the cost of each processor in a parallel system is substantially less than current processor costs, this method will be economically unfeasible for n sufficiently large. We further observe that parallel computers which are or soon will be available will not provide n^2 processors for reasonable values of n. Thus, we restrict our attention to the case where the number of processors is in the range from 2 to $0(n)$.

The algorithm which we present provides the LU decomposition of an $n \times n$ non-singular matrix A using from 1 to $\lceil \frac{n}{2} \rceil$ processors and has an efficiency of 2/3 when $P = \lceil \frac{n}{2} \rceil$.

Consider the sequential program for LU decomposition with partial pivoting given in Fig. 1.

Program LUDECOMP (A(n,n)).

```
For k ← 1 to n-1 do
    Find ℓ such that                                  ⎫
    |A(ℓ,k)| = max(|A(k,k)|,...,|A(n,k)|)             ⎪
    PIV(k) ← ℓ   {pivot row}                          ⎪
    A(PIV(k),k) ↔ A(k,k)                              ⎬ T_k^k
    c ← 1/A(k,k)                                      ⎪
    For i ← k+1 to n do                               ⎪
        A(i,k)   A(i,k)*c   {elements of L}           ⎭
    For j ← k+1 to n do
        A(PIV(k),j) ↔ A(k,j)                          ⎫
        For i = k+1 to n do                           ⎬ T_k^j, j > k
            A(i,j) ← A(i,j) - A(i,k)*A(k,j)           ⎭
```

Fig. 1: Program for LU decomposition with illustration of tasks.

In this program we shall consider a task to be that code segment which works on a particular column j for a particular value of k. We will denote these tasks by $J = \{T_k^j \mid 1 \le k \le j \le n,\ k \le n-1\}$.

The precedence constraints imposed by the sequential program are

$$<\cdot\ = \{(T_k^j, T_m^\ell) \mid j < \ell \text{ and } k=m, \text{ or } k<m\}.$$

Thus, $C = (J, <\cdot)$ is the task system which represents the sequential program (Coffman, Denning [1]). The range and domain of these tasks are:

CH1569-3/80/0000-0205$00.75 © 1980 IEEE

$$R(T_k^j) = \{A(i,j) \mid k \le i \le n\}$$

$$D(T_k^j) = \{A(i,j) \mid k \le i \le n\} \cup \{A(i,k) \mid k \le i \le n\}$$

and from this we can observe that, for example, $\{T_k^{k+1}, T_k^{k+2}, \dots, T_k^n\}$ are all mutually noninterfering tasks and could be executed in parallel. More specifically we observe that $C' = (J, <\cdot')$ where $<\cdot'$ is the transitive closure on the relation $X = \{(T_k^k, T_k^j) \mid k<j\le n\} \cup \{(T_k^j, T_{k+1}^j) \mid k<j\le n\}$ is a maximally parallel system equivalent to C. This system is illustrated in Fig. 2.

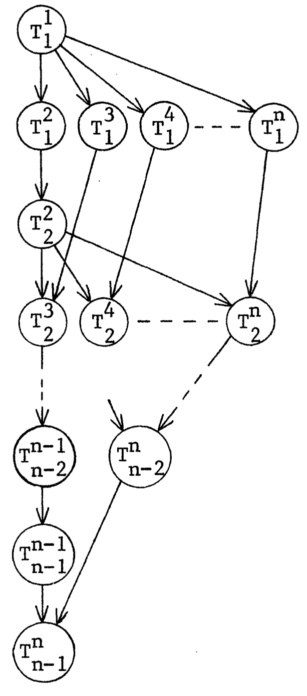


Fig. 2: Maximally Parallel Task System Equivalent to C.

Given the task system C' we now determine the execution time of the tasks and from that determine a schedule. We assume that one multiply and one subtract, or one multiply and one compare constitute a time step. Thus, neglecting any overhead for loop control, the execution time $W(T_k^j)$ for each of the tasks is given by:

$$W(T_k^j) = \begin{cases} n+1-k & \text{if } k = j \\ n-k & \text{if } k < j. \end{cases}$$

Treating the task system C' together with $W(T_k^j)$ as a weighted graph we observe that the longest path traverses the nodes: $T_1^1, T_1^2, T_2^2, T_2^3, T_3^3, ---, T_{n-1}^{n-1}, T_{n-1}^n$. We will denote this path as s_1 and the length of the path by $L(s_1)$.

$$L(s_1) = n+1 + 2\sum_{j=2}^{n-1} j = n^2-1$$

Since the problem cannot be solved in time shorter than this path length we developed a schedule where the tasks constituting s_1 are assigned to processor 1 and the remaining tasks are assigned to $\lceil \frac{n}{2} \rceil - 1$ additional processors. Processor 2 will execute the tasks $T_1^3, T_1^4, T_2^4, T_2^5, T_3^5, \dots, T_{n-2}^n$ and, more generally, processor j will execute the tasks $T_1^{2j-1}, T_1^{2j}, T_2^{2j}, T_2^{2j+1}, \dots, T_{n-2(j-1)}^n$ and we will denote this as s_j. Note that this is not a path through the graph. Since this schedule has length n^2-1, the length of the longest path, then this schedule is optimal for n/2 processors. Using this schedule we note that:

$$\lim_{n\to\infty} \frac{S_p}{p} = \lim_{n\to\infty} \frac{\frac{n^3}{3} + O(n^2)}{(n^2-1)n/2} = \frac{2}{3}$$

and this efficiency is achieved to within 2% for relatively small n ($n \ge 50$).

We now examine the question as to whether a schedule of length n^2-1 is achievable with $p \le n/2$ processors. From the task system C as illustrated in Fig. 2 we note that task T_1^1 is a predecessor to all tasks and has an execution time of n steps. Consequently, any schedule for this system will have only one processor doing work during the first n steps. Similarly, T_{n-1}^n is the successor of all tasks and thus during the last time step only one processor can be doing work. Task T_{n-2}^{n-1} has all tasks except $\{T_{n-1}^{n-1}\} \cup \{T_j^n \mid 1 \le j \le n-1\}$ as predecessors, task T_{n-1}^{n-1} is a successor task and for the tasks $\{T_j^n \mid 1 \le j \le n-1\}$ each is a successor or predecessor of all other tasks in the set. Thus, for any schedule from the time that T_{n-2}^{n-1} commences execution, no more than 2 processors can be doing work. By similar argument, once T_{n-j}^{n-j+1} commences execution no more than j processors can be doing work. From this, we define the "computational area" of any schedule C to be the product of the number of processors and the schedule length less the area where not all the processors can be doing work. Since $L(T_{n-j}^{n-j+1}) = j$ and during each time interval of length 2j at most j processors are working, we have:

$$CA = (n^2-1)p - (p-1)n - 2\sum_{j=2}^{p-1} (p-j)j - (p-1)$$

$$= (n^2-1)p - (p-1)(n-1) - (p^3-p)/3.$$

The total amount of work (sum of the task weights) for the task system C is $TW = (n^3/3) + (2n/3) - 1$. Thus, a lower bound on the number of processors required to achieve a schedule of length n^2-1 is the smallest p for which $CA \geq TW$. For small even values of n the minimum p values are:

$2 \leq n \leq 8$	$p = (n/2)$
$10 \leq n \leq 14$	$p = (n/2)-1$
$16 \leq n \leq 22$	$p = (n/2)-2$
$24 \leq n \leq 28$	$p = (n/2)-3$
$30 \leq n \leq 34$	$p = (n/2)-4$
$36 \leq n$	$p \leq (n/2)-5$

For large values of n let $p = \alpha n$ and determine α such that

$$\lim_{n\to\infty}(CA/TW) = 1$$

Thus, an α to satisfy the above limit is a solution to $3\alpha - \alpha^3 = 1$ and an approximate solution to this is $\alpha = 0.34729$. We note that this is only a lower bound and we do not know if it is achievable in general, however, for $n = 10$ we have found a schedule of length n^2-1 using $(n/2)-1$ processors and for $n = 16$ a schedule using $(n/2)-2$ processors.

Should this lower bound be achievable then the efficiency for large n and using αn processors would be

$$\lim_{n\to\infty} (S_p/p) = \frac{n^3/3}{(n^2-1)\alpha n} = \frac{1}{3\alpha} \simeq 0.9598.$$

Acutal Performance

The achievable schedules previously discussed were programmed using HEP FORTRAN and were executed on the HEP parallel computer. Although the program provided solutions to a set of linear equations, we present timing only for the LU decomposition part of the solution so that it may be compared with our predicted results. Due to memory limitations of the machine to which we had access, we could only run programs with $n \leq 35$ and $1 \leq p \leq 8$. Table 1 gives the achieved results together with a comparison of the predicted results.

Although the actual results are limited by the restriction on the maximum value for n, we feel that the agreement between actual and predicted performance is sufficiently good to give credibility to our model of the algorithm's performance and that the efficiencies are high enough to support the conclusion that parallel methods for solving linear equations are a viable alternative to sequential methods.

Fast Givens Transformations

To solve the square system of equations $A\underset{\sim}{x} = \underset{\sim}{b}$ using the fast Givens transformations, due to Gentleman [2], we proceed as follows:

(i) The matrix A is kept in the factorized form $A = D^{1/2}B$ where D is a diagonal matrix. Initially $D = I_{n\times n}$, B=A where n is the number of equations.

(ii) Triangularize the matrix A by applying Givens rotations to the augmented matrix $[A,\underset{\sim}{b}]$ and obtain the factors Q, $\hat{D}$, R and $\hat{\underset{\sim}{b}}$, such that

$$Q[A,\underset{\sim}{b}] = Q[D^{1/2}B,\underset{\sim}{b}] = \hat{D}^{1/2}[R,\hat{\underset{\sim}{b}}],$$

where R is upper triangular, Q is the product of the orthogonal transformations used in the triangularization, and $\hat{D}$ is diagonal.

(iii) Solve the upper triangular system $R\underset{\sim}{x} = \underset{\sim}{b}$ by back substitution.

The sequential method of orthogonal triangularization of A, eliminates the subdiagonal nonzero elements of A one at a time. The elimination process is performed sequentially by applying Givens plane rotation to A in such a way that the previously introduced zeros are not destroyed. For each column j of A, n-j rotations are required. This can be accomplished by algorithm 1:

```
for i ← 1 to n-1 do
    for j ← i+1 to n do
        GIVENS (i,j)
```

which reduces A to $\hat{D}^{1/2}R = QA$, where $Q = P_{1,2}, P_{1,3},\ldots,P_{1,n},\ldots,P_{n-1,n}$ is the product of the $n(n-1)/2$ Givens plane rotations. GIVENS (i,j) is a subroutine which constructs and applies the plane rotation $P_{i,j}$. The matrix $P_{i,j}$ rotates the rows i and j and annihilates the element in the (i,j)-th position. The entire process requires $\frac{4}{3}n^3 + \frac{7}{2}n^2 - \frac{29}{6}n$ arithmetic operations.

In a parallel implementation of the fast Givens method more than one plane rotation could be applied concurrently. Sameh and Kuck [5], and Kowalik et al. [4] describe details of such schemes which assume tnat $p = O(n^2)$ processors are available. The algorithm proposed in Kowalik et al. [4] produces the orthogonal matrix $Q = Q_{2n-3}, Q_{2n-4}\ldots Q_2Q_1$ where $Q_k = \{P_{i,j} | i < j = 1,2,\ldots,n, i+j = k+2\}$, $k = 1,2,\ldots,2n-3$, and $P_{i,j}$ are applied in parallel.

For the purpose of this analysis and implementation we assume that the number of available processors is $p = (n-1)/2$ where n is odd. We also assume that every Givens rotation is performed sequentially, however, more than one rotation could be performed in parallel.

We derive now a parallel scheme to triangularize A from the sequential method given in algorithm 1.

Let a task T_j^i in algorithm 1 be defined by T_j^i = GIVENS(i,j) where GIVENS(i,j) performs the following calculations:

1. $\alpha = -B(j,i)/B(i,i)$
2. $\beta = -(D(j)/D(i))*\alpha$
3. $\gamma = 1 - \alpha\beta$
4. $D(i) = (1/\gamma)D(i)$
5. $D(j) = (1/\gamma)D(j)$
6. $B(i,\ell) = B(i,\ell) + \beta B(j,\ell)$
7. $B(j,\ell) = B(j,\ell) + \alpha B(i,\ell)$ $\Big\}\ i \le \ell \le n$

Periodic rescaling of D and B to prevent underflows and overflows, and row interchanges for numerical stability are included in our implementation of the Givens routine. The precedence constraints on the set of these tasks

$$J = \{T_j^i \mid 1 \le i \le n-1,\ i < j \le n\}$$

imposed by algorithm 1 are given by

$$<\!\cdot\ = [\{(T_j^i, T_{j+1}^i) \mid 1 \le i \le n-2,\ i < j \le n-1\}$$

$$\cup\ \{(T_n^i, T_{i+2}^{i+1}) \mid 1 \le i \le n-2\}]^*$$

where * represents the transitive closure of the set. Thus the system C = (J,<•) is the task system representing the sequential program. The range and domain of these tasks are:

$$R(T_j^i) = (D(i), D(j), B(i,\ell), B(j,\ell) \mid i \le \ell \le n)$$

$$D(T_j^i) = (D(i), D(j), B(i,\ell), B(j,\ell)\ i \le \ell \le n).$$

From this we can see that the tasks $\{T_j^i \mid i < j \le n,\ 1 \le i \le n-1,\ i+j = k+2,\ k = 1,2,\ldots,2n-3\}$ are mutually noninterfering tasks and can be executed in parallel. Hence we obtain a maximally parallel task system C' = (J,<•'), where

$$<\!\cdot\,' = [(T_j^i, T_{j+1}^i) \cup (T_j^i, T_j^{i+1}) \mid 1 \le i \le n-2,\ i < j \le n-1]^*$$

is equivalent to C.

This maximally parallel task system C' is shown in Fig. 3. We now assume that one arithmetic operation constitutes a time step. Thus the length of T_j^i is $L(T_j^i) = 4(n-i+1) + 7$ steps. The longest path in this maximally parallel task system is $s_1 = \{T_2^1, T_3^1, \ldots, T_n^1, T_n^2, \ldots, T_n^{n-1}\}$, and the total length of s_1 is

$$L(s_1) = (4n+7)(n-1) + (4(n-1)+7) + (4(n-2)+7) + \ldots.\ (4\cdot 2+7)$$

$$= 6n^2 + 8n - 25 \text{ operations.}$$

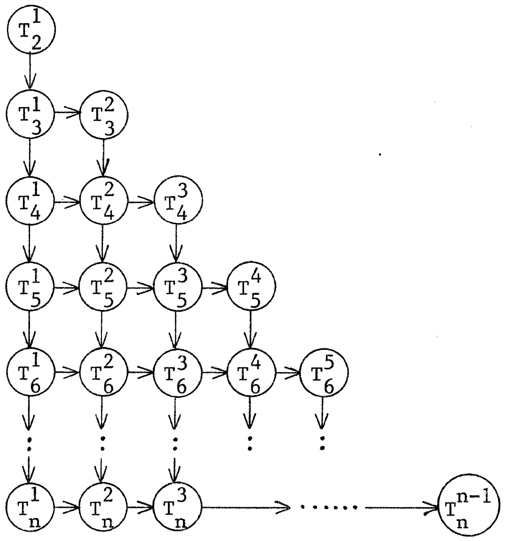

T_2^1
T_3^1 T_3^2
T_4^1 T_4^2 T_4^3
T_5^1 T_5^2 T_5^3 T_5^4
T_6^1 T_6^2 T_6^3 T_6^4 T_6^5
T_n^1 T_n^2 T_n^3 T_n^{n-1}

Fig. 3: Maximally Parallel Task System C'.

To execute our task system with p = (n-1)/2 processors we have selected a scheduling scheme called ZIGZAG, shown in Fig. 4. According to this scheme the processors p_k, k = 1,2,..,(n-1)/2 are assigned to the tasks as follows:

p_1 executes: $\{T_2^1, T_3^1, T_3^2, T_4^2, \ldots, T_{n-1}^{n-2}, T_n^{n-2}, T_n^{n-1}\}$

p_2 executes: $\{T_4^1, T_5^1, T_5^2, T_6^2, \ldots, T_{n-1}^{n-4}, T_n^{n-4}, T_n^{n-3}\}$

⋮

p_j executes: $\{T_{2j}^1, T_{2j+1}^1, T_{2j+1}^2, T_{2j+2}^2, \ldots, T_n^{n-2j+1}\}$

⋮

$p_{\frac{n-1}{2}}$ executes: $\{T_{n-1}^1, T_n^1, T_n^2\}$.

For this schedule the speedup and efficiency are:

$$S_p = \frac{T_1}{T_p} = \frac{\frac{4}{3}n^3 + O(n^2)}{6n^2 + O(n)} \simeq \frac{\frac{4}{3}n^3}{6n^2} = \frac{2n}{9}$$

$$E_p = \frac{S_p}{p} = \frac{2n}{9}\cdot\frac{2}{n-1} = \frac{4}{9}\cdot\frac{n}{n-1}$$

and for sufficiently large values of n, $E_p = 4/9 = 0.444\ldots$.

Computational Results

The ZIGZAG scheme for orthogonal triangularization shown in Fig. 4 was programmed and executed on the HEP parallel computer. Due to the present memory limitations the program was run for the values of n not exceeding n = 29. Since for this machine $1 \le p \le 8$, and we assumed that p = (n-1)/2, the obtained numerical results

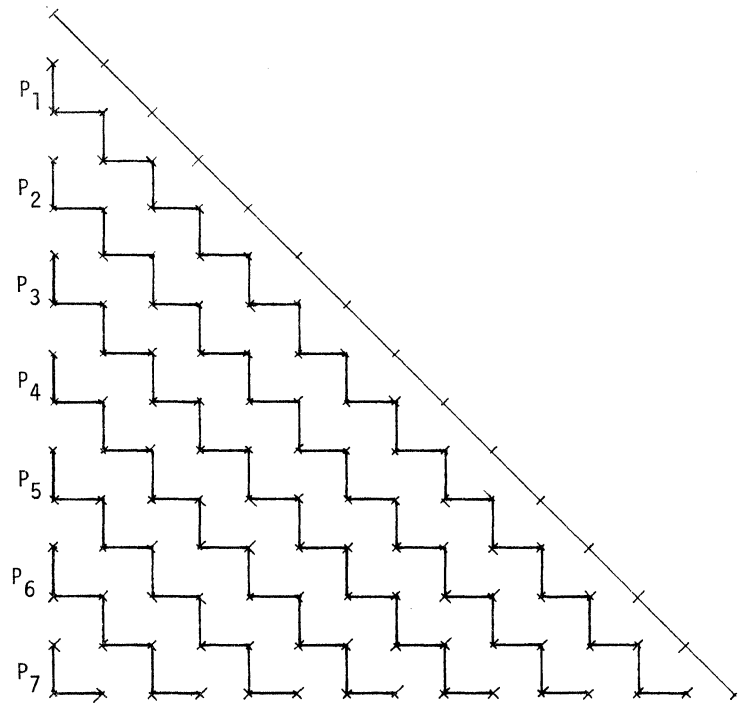


Fig. 8: Parallel Zigzag Scheme for $n = 15$, $p = \frac{n-1}{2} = 7$.

up to n = 17 are useful to compare. The actual and predicted speedups and efficiencies of the algorithm for different values of n are shown in Table 2. The differences between the predicted and actual values of S_p and E_p are due to several factors: machine overhead, approximate count of arithmetic operations involved in Givens rotations, and data dependent number of scaling operations in the GIVENS routine which are not included in the operations count.

References

[1] E.G. Coffman, Jr., and P.J. Denning, Operating Systems Theory, Prentice Hall, (1973), 331 pp.

[2] W.M. Gentleman, "Least Squares Computation by Givens Transformations without Square Roots," J. Inst. Math. Applic. (Aug. 1973), pp. 329-336.

[3] D. Heller, "A Survey of Parallel Algorithms in Numerical Linear Algebra," SIAM Review (Oct. 1978), pp. 740-777.

[4] J.S. Kowalik, S.P. Kumar, and E.R. Kamgnia, "An Implementation of the Fast Givens Transformations on a MIMD Computer," (Wash. State Univ., Computer Sci. Dept., Pullman, WA, 99164), (1980), unpublished manuscript.

[5] A.H. Sameh and D.J. Kuck, "On Stable Linear System Solvers," J. ACM (Jan. 1978), pp. 31-91.

[6] B.J. Smith, "A Pipelined, Share Resource MIMD Computer," International Conference on Parallel Processing, Wayne State Univ., IEEE and ACM (Aug. 1978).

	number of processors p							
	2	3	4	5	6	7	8	
n=10	.833 .852	.719 .739	.642 .678	.633 .685				A P
n=15	.888 .900	.794 .815	.740 .766	.651 .679	.618 .652	.625 .681	.581 .633	A P
n=20	.921 .931	.843 .863	.774 .798	.758 .789	.670 .703	.623 .656	.605 .640	A P
n=25	.934 .944	.878 .896	.830 .855	.763 .739	.755 .788	.692 .726	.642 .675	A P
n=30	.942 .949	.892 .911	.844 .863	.818 .843	.757 .783	.744 .777	.710 .745	A P
n=35	.948 .956	.901 .918	.862 .880	.819 .843	.790 .827	.747 .779	.741 .769	A P

Table 1: Actual and Predicted Efficiency.

n	P	T_1	T_p	S_p	E_p	
5	2	.0036	.0025	1.44 1.40	.72 .70	A P
7	3	.0087	.0045	1.93 1.83	.64 .61	A P
9	4	.0168	.0072	2.33 2.27	.58 .57	A P
11	5	.0286	.0105	2.72 2.72	.54 .54	A P
13	6	.0448	.0146	3.07 3.16	.51 .52	A P
15	7	.0660	.0194	3.34 3.61	.47 .51	A P
17	8	.0927	.0256	3.62 4.01	.45 .50	A P

Table 2: Actual and Predicted Speedup and Efficiency.
Time is measured in seconds.

OPTIMAL INTEGRATED-CIRCUIT IMPLEMENTATION OF TRIANGULAR MATRIX INVERSION [†]

Franco P. Preparata
Coordinated Science Laboratory
University of Illinois
URBANA, Illinois 61801
U.S.A.

Jean Vuillemin
Laboratoire de Recherche en Informatique
Bât 490, Université de Paris-Sud
91405 ORSAY
France

Abstract

We describe a class of integrated-circuit implementations of algorithms for inverting an n x n triangular matrix. These networks have area A and time T, with an area x time^2 product $AT^2 = O(n^4)$ for all values of T such that $O(\log^2 n) \le T \le O(n)$. Since there is a simple reduction of matrix multiplication to inversion of a triangular matrix, and Savage [6] has given an $AT^2 = \Omega(n^4)$ lower-bound for n x n matrix multiplication, the presented networks are asymptotically optimal in the VLSI model.

Keywords : VLSI, matrix inversion, triangular matrices, area-time complexity, pipeline computation, optimal networks.

1. Introduction

Increasing attention has been paid recently to the design of networks for the direct implementation of several interesting algorithms using the integrated-circuit technology (VLSI) ; particulary, combinatorial and numerical problems have been the target of these investigations [1-4].

Among numerical problems, several workers have directed their attention to matrix computations [1,2,5], and, as regards the design of networks, have found that the mesh interconnection of computing modules is particularly attuned to this class of problems, leading to optimal realizations [5,6] in the VLSI model [7,8].

[†] In this paper we consider the problem of designing VLSI networks for inverting a non singular triangular matrix. The design complies with specifications of the VLSI model of computation recently proposed by Mead, Conway, and Thompson [7,8], and further refined by Brent , Kung [3].

This work was partially supported by National Science Foundation Grandt MCS-78-13642, and by the Joint Services Electronics Program Contract N00014-79-C-0424, and by ERA 452 "Al Khowarizmi", Centre National de la Recherche Scientifique, France.

In this model, the network is a computation graph consisting of nodes (processing modules) and wires. Wires have unit width and are partitionable into two orthogonal sheaves. A data item takes a unit of time to propagate along a wire from node to node (processing time is thus absorbed into propagation time). A mathematically natural complexity metric is the area x time^2 product (AT^2), which embodies a trade-off between production cost (chip area A) and incremental cost (time T).

Within this model, Savage [6] has recently proved the following interesting result : any VLSI design for the multiplication of two n x n matrices, with chip area A and computation time T, must satisfy the bound $AT^2 \ge C n^4$, for some constant C. In [5] the authors demonstrate the existence of VLSI networks for multiplying n x n matrices with $AT^2 = O(n^4)$ for any computation time T in the range $\log_2 n \le T \le n$. Note that an $AT^2 \ge C'n^4$ bound also holds for the problem of inverting a non singular n x n triangular matrix, since matrix multiplication is reducible to it ; the straightfor ward reduction is based on the fact that the inverse of the 3n x 3n triangular matrix

$$\begin{bmatrix} I & A & 0 \\ 0 & I & B \\ 0 & 0 & I \end{bmatrix} \quad \text{is} \quad \begin{bmatrix} I & -A & AB \\ 0 & I & -B \\ 0 & 0 & I \end{bmatrix}$$

i.e., it contains an n x n block equal to the product AB.

This paper is organized as follows : In Section 2 we present a general scheme for inverting an n x n triangular matrix, and evaluate two network implementations, corresponding respectively to block-partitioning the matrix and choosing extreme values for the block size in the allowable range. These two inverters are referred to as "recursive" and "systolic" respectively ; with respect to the AT^2 measure, only the latter is optimal for $T = O(n)$. In Section 3 we show that the recursive and systolic inverters can be combined to build networks, called "mixted" inverters, which meet the optimal $AT^2=\Omega(n^4)$ bound for all values of T such that $O(\log^2 n) \le T \le O(n)$.

CH1569-3/80/0000-0211$00.75 © 1980 IEEE

2. The general scheme for inverting a nonsingular triangular matrix.

Let A be a nonsingular n x n triangular matrix(1) to be thought of as an n/s x n/s matrix whose elements are s x s blocks of the original entries (s is a parameter in the range $[1,n/2]$) ; let A_{ij} be the (i,j) block of A(i,j=1,2....,n/s) and let $A_{ij}^{(-1)}$ be the corresponding block of A^{-1}. It is straightforward to verify that

$$A_{ii}^{(-1)} = A_{ii}^{-1} \qquad\qquad A_{ij}$$

$$A_{ij}^{(-1)} = [A_{ii}^{(-1)} \ldots, A_{i,j-1}^{(-1)}] \cdot \begin{bmatrix} A_{i+1,j} \\ \cdot \\ \cdot \\ \cdot \\ A_{j-1,j} \end{bmatrix} \cdot A_{jj}^{(-1)} \qquad (1)$$

for $i \geq j$

This general formula will now be specialized to two interesting cases.

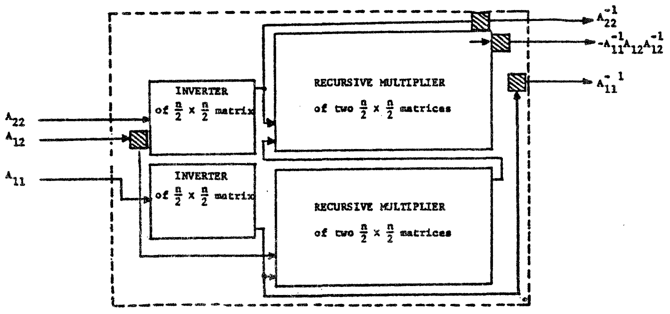
2.1 Recursive inversion

The standard scheme for the parallel inversion of a triangular matrix[9,10] corresponds to specializing the general scheme to s=n/2.
In this case the inverse of

$$\begin{bmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{bmatrix} \text{ is } \begin{bmatrix} A_{11}^{-1} & -A_{11}^{-1}A_{12}A_{22}^{-1} \\ 0 & A_{22}^{-1} \end{bmatrix} . \qquad (2)$$

This immediately suggests a recursively defined network, containing two inverters of n/2 x n/2 triangular matrices (to be used to compute A_{11}^{-1} and A_{22}^{-1} in parallel) and a network for the parallel multiplication of two n/2 x n/2 matrices(to be used to compute $(A_{11}^{-1}A_{12})A_{22}^{-1}$ in the order shown by the parenthesization). In figure 1, we show a possible layout for such a network. Each line shown carries $n^2/4$ operands in parallel and the shaded surfaces are buffers of area $(n^2/4)$; the core of the circuit are two multipliers of two (n/2) x (n/2) matrices, of a type described in [5], and called recursive multipliers. Each of these multipliers has height and width respectively proportional to n^2 and computes a matrix product in O(logn) time units. Due to the recursive definition of the inverter,

(1) The entries of all matrices considered in this paper are assumed to be drawn from a finite ring, so that an elementary finte chip can be used for multiplying and adding entries in constant area and time.



A_{22}
A_{12}
A_{11}
INVERTER of $\frac{n}{2} \times \frac{n}{2}$ matrix
INVERTER of $\frac{n}{2} \times \frac{n}{2}$ matrix
RECURSIVE MULTIPLIER of two $\frac{n}{2} \times \frac{n}{2}$ matrices
RECURSIVE MULTIPLIER of two $\frac{n}{2} \times \frac{n}{2}$ matrices
A_{22}^{-1}
$-A_{11}^{-1}A_{12}A_{12}^{-1}$
A_{11}^{-1}

Figure 1. Layout of the recursive matrix inverter ; shaded boxes are data buffers.

a simple argument shows(2) that its height and width are also respectively proportional to n^2 also, the computation time is $O((\log n)^2)$. Note therefore that for the matrix inverter being described called recursive inverter - we have the following properties :

(3)

Type	A	T	AT^2
Recursive inverter	$O(n^4)$	$O(\log^2 n)$	$O(n^4 \log^4 n)$

Note that AT^2 is short of the optimal $\Omega(n^4)$ by a small order factor $O(\log^4 n)$.

2.2. Systolic inversion

The next scheme to be described corresponds to the choice s=1 in the general method. The resulting network is a mesh of processors, each of which feeds data in and out, each time performing some computation, keeping a regular flow in the network. Such networks have been called systolic by Kung an Leiserson[11].

(2) Recurrences defining height and width are of the form $f(n) \leq f(n/2)+An^2$ for some constant A ; the solution satisfies $f(n) \leq \frac{4}{3} A n^2$.

With our choice of s, block A_{hk} in (1) becomes entry a_{hk} (and similarly $A_{hk}^{(-1)}$ becomes $a_{hk}^{(-1)}$). The form of (1) suggests a computation method on an n x n square mesh (figure 2). Only the upper-triangular positions in this mesh need contain processing modules (i.e., denoting by M_{ij} the module in position (i,j), M_{ij} is deployed only for j≤i). Modules are of two types with different computational capabilities : D-modules and M-modules, placed respectively in diagonal and off-diagonal positions.

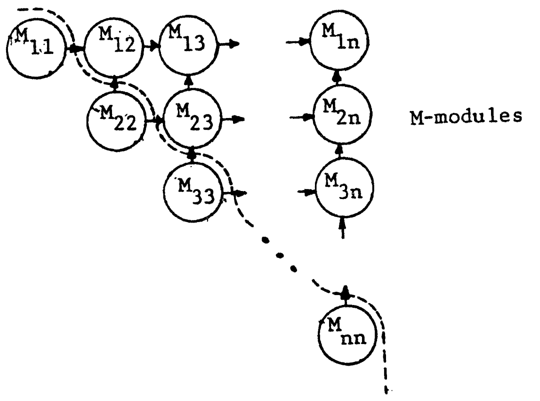
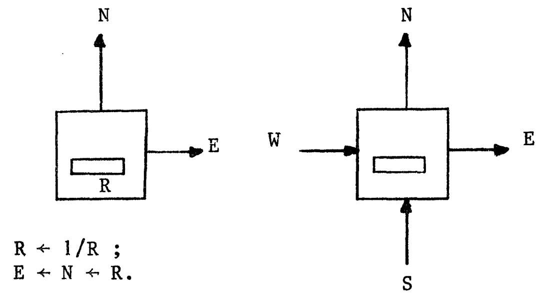


Figure 2. Generalstructure of the systolic matrix inverter (triangular mesh)

Each module contains an operand register R, and input/output ports referred to by means of the compass points N,E,S,and W; the instructions executable in either type of module are shown compactly in figure 3.



R ← 1/R ;
E ← N ← R.

initialisation step :
R ← W . R ; E ← W ;
general step :
R R + W . S; E ←W; N←S;
final step :
R← - R.S; E←R; N←S;

D-modules M-modules

Figure 3. Input/output structures and instruction sets of D-modules an M-modules.

Initially, each entry a_{ij} (i≤j) is read into register R of module M_{ij}.

The first module to be activated is M_{11}, which computes $a_{11}^{(-1)} = 1/a_{11}$ in R ,sends the result to M_{12} and activates M_{22}. All D-modules perform the same function upon activation : invert the a_{ii} entry, broadcast it eastward to $M_{i,i+1}$ and activate $M_{i+1,i+1}$.

As for off-diagonal modules, they accumulate the inner product $\sum_{i\leq k<j} a_{ik}^{(-1)} \cdot a_{kj}$ in their general step, and transform it to $a_{ij}^{(-1)} =$ $-\sum_{i\leq k<j} a_{ik}^{(-1)} a_{kj}) \times a_{jj}^{-1}$ in their final step. For the purpose of the general step, entries $a_{ik}^{(-1)}$ are transmitted eastward along horizontal lines, while entries a_{kj} are transmitted northward along verticals lines. A timing argument shows later that $a_{ik}^{(-1)}$ and a_{kj} meet in M_{ij} for k=i,...,j-1. Module M_{ij} is thus activated when it receives a_{ii}^{-1} on its west entry port and it proceeds with its initialization step : sending a_{ij} northward, passing a_{ii}^{-1} eastward,accumulating a_{ii}^{-1} a_{ij} his register R,and entering its general step. During the general step, M_{ij} receives a_{ik}^{-1} and a_{kj} on its W-and S-entry ports; it dutifully passes them on E-and N-ports accumulating $R \leftarrow R + a_{ik}^{-1} \cdot a_{kj}$. It enters the final step when it receives a_{jj}^{-1} on its S-entry port ; next the S input is passed northward, the result a_{ij}^{-1} kept in R and also transmitted eastward.

To ensure that timing is correct, we can verify that :

- Module M_{ij} is activated at time j ;
- M - modules M_{ij} are in their general step from time j+1 to 2j-i-1 ; their final step occurs at time 2j-i.
- entries a_{ip}^{-1} and a_{pj} reside in M_{ij} at the p - i+j step ;
- entry a_{jj}^{-1} arrives from S in M_{ij} at time 2j-i.

For clarity, in figure 4(a) we illustrate the timing of the computations : Each module is labelled with an integer which denotes the step at which computation in that module is completed. Also, in figure 4(b and c) we present snapshots of the data participating in the horizontal and vertical flow, respectively, at step 7. Clearly the calculation of A^{-1} is completed in 2n-1 steps.

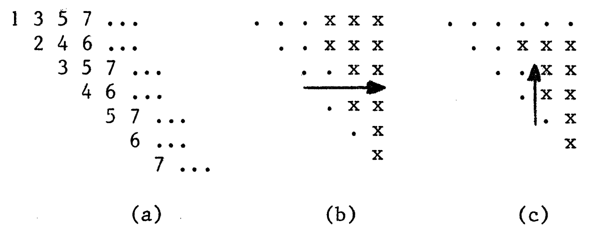


Figure 4(a): timing of completion of computation up to step 7.
(b): data (x) participating in horizontal flow at step 7.
(c): data (x) participating in vertical flow at step 7.

According to our original assumption that both the area of the processing modules and the time needed to execute any of the prescribed operations be bounded by a constant, we have the following :

Type	A	T	AT^2
systolic inverter	$O(n^2)$	$O(n)$	$O(n^4)$

,(4)

i.e., the network is optimal for the AT^2 measure. The optimal behavior, however, is achieved only for $T = O(n)$. An interesting question is whether it can be extended to a wider range of processing times. This question is addressed in the next section.

3. Mixed networks

We now describe how to combine the recursive and systolic inverters described in the preceding section in order to improve the AT^2 measure for a wide range of the time parameter T.

The resulting networds -to be called mixed- have the following general structure. A mixed network is a systolic scheme, as the one described in 2.2, where the "operands", rather than being elementary entries, are blocks of $s \times s$ such entries. In the corresponding $n/s \times n/s$ triangular mesh (see figure 2), the modules must now be designed to process $s \times s$ blocks. The layout of mixed networks is chosen as in figure 5, where the modules themselves have been conveniently assumed to have a rectangular shape on the chip (else, we consider the smallest rectangle with sides parallel to the coordinate axes which contains the module). From figure 5 it is clear that while the dimensions (width and height) of the M-module determine one dimension of the network -say, its width-, the other dimension -say, its height- is determined by the larger of the corresponding values for the D- and M-modules.

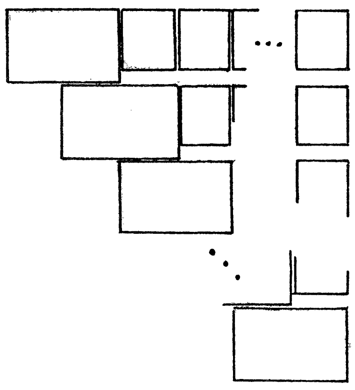


Figure 5 : General layout of mixed networks.

We can design a mixed inverter as follows : (Called Type-1 mixed inverter :

(1) D-modules are recursive inverters, as described in Section 2.1 ; they have width and height proportional to s^2, and computation time $O(\log^2 s)$.

(2) M-modules are recursive matrix multipliers of the type shown in [5], as already used to build the recursive inverter. They can be placed on the chip so that their width and height are both $O(s^2)$. Their computation time is $O(\log s)$. The following point must be noted : although this type of multiplier is completely pipelinable, i.e., it can complete an $s \times s$ matrix product at each step, we cannot take advantage of the property since the term $A^{(-1)}_{ij}$ is available on the E-port of M_{ij} only $O(\log s)$ time units after the eastward transmission of $A^{(-1)}_{i,j-1}$ (indeed $A^{(-1)}_{ij} = {}_{i<k<j} A^{(-1)}_{ik} \cdot A^{(-1)}_{ij}$, i.e., the multiplication $A^{(-1)}_{i,j-1} \cdot A_{i-1,j}$ must be completed before the final step of module M_{ij} may begin).

Since the heights of the D- and M-modules are of the same order, the height of the networks is $O(\frac{n}{s} \times s^2) = O(ns)$, and the same holds for the width of the network. Thus, $A = O(n^2 s^2)$, and the smallest containing rectangle is nearly a square with both sides $O(ns)$. As regards computation time, the blocks A^{-1}_{ii} $(i = 1,\ldots,n/s)$ are all computed in time $O(\log^2 s)$, and, after this, the mesh computation begins. We have shown in

Section 2.2 that the systolic-network completes its computation in $O(n/s)$ steps, whence the total computation time is $T = O(\log^2 s + \frac{n}{s} \log s)$. If the we bounded the parameter s by $s \leq n/\log n$ we obtain $T = O(\frac{n}{s} \log s)$, and the performance of Type-1 networks is summarized as follows :

Type	A	T	AT^2
Type-1 mixed inverter	$O(n^2 s^2)$	$O(\frac{n}{s} \log s)$	$O(n^4 \log^2 s)$

for cons. $\leq s \leq n/\log n$

The second kind of mixed networks (Type-2 mixed inverter) is constructed as follows :

(1) D-modules are type-1 mixed inverters (for $s \times s$ matrices). According to the preceding discussion, for any value of a parameter $r \leq s/\log s$, these modules have height and width both $O(sr)$ and computation time $O(\log^2 r + \frac{s}{r} \log r)$. choosing $r = s/\log s$ we obtain height $O(s^2/\log s)$, width $O(s^2/\log s)$, and time $O(\log^2 s)$.

(2) M-modules are <u>pipelined matrix multiplier</u>, as introduced by Preparata and Vuillemin [5]. It is shown in [5] that one such multiplier can be designed with height and width both $O(s^2/\log s)$ and computation time $O(\log s)$.

Again, the dimensions of both D-modules and M-modules are $O(s^2/\log s)$, whence :

$$A = O\left(\left(\frac{n}{s} \cdot \frac{s^2}{\log s}\right)^2\right) = O\left(\frac{n^2 s^2}{\log^2 s}\right).$$

With respect to computation time, we obtain the same conclusions as for type-1 mixed inverters, i.e.,

$$T = O(\log^2 s + \frac{n}{s} \log s).$$

Therefore we obtain

$$AT^2 = O\left(\frac{n^2 . s^2}{\log^2 s} \cdot (\log^2 s + \frac{n}{s} \log s)^2\right)$$

$$= O\left(\frac{n^2 s^2}{\log^2 s} \cdot \frac{n^2}{s^2} \log^2 s . (1 + \frac{s}{n} \log s)^2\right)$$

$$= O\left(n^4 . (1 + \frac{s}{n} \log s)^2\right).$$

Obviously, if $s \leq n/\log n$ we have $s \log s < n$, whence $AT^2 = O(n^4)$, and the performance of the Type-2 mixed inverter is so summarized :

Type	A	T	AT^2
Type-2 mixed inverter	$O\ \frac{n^2 s^2}{\log^2 s}$	$O(\frac{n}{s} \log s)$	$O(n^4)$

const. $\leq s < n/(\log n)^2$

Since as s varies from a small constant value to $n/(\log n)^2$ the computation time T varies from $O(n)$ to $O(\log^2 n)$, we can design networks meeting the $AT^2 = O(n^4)$ optimal bound for all T such that $O(\log^2 n) \leq T < O(n)$. Identically, even in totally unrestricted models of computation -as the shared-memory-machine [see, for example [10]]- $O(\log^2 n)$ is the smallest known running time for inverting a triangular matrix.

REFERENCES

1. H. T. Kung and C. E. Leiserson, "Algorithms for VLSI processor arrays," <u>Symposium on Sparse Matrix Computations</u>, Knoxville, Tenn., Nov. 1978.

2. L.J. Guibas, H.T. Kung, and C.D. Thompson, "Direct VLSI implementation of combinatorial algorithms," <u>Proc. Conference on VLSI Architecture, Design, Fabrication</u>, Calif. Inst. of Techn., January 1979.

3. R.P. Brent and H.T. Kung, "The area-time complexity of binary multiplication," Research Report, Department of Computer Science, Carnegie-Mellon University, Pittsburg, Penn., July 1979

4. C.D. Thompson, "Area-time complexity for VLSI," Proc. of the 11th Annual ACM Symposium on the Theory of Computing (SIGACT), pp. 81-88, May 1979.

5. F.P. Preparata and J. Vuillemin, "Area-time optimal VLSI networks for multiplying matrices", 14th Princeton Conference on Information Sciences and Systems, March 1980.

6. J.E. Savage, "Area-time tradeoffs for matrix multiplication and transitive closure in the VLSI model," Proc. of the 17th Annual Allerton Conference on Communications, Control, and Computing, October 1979.

7. C. Mead and L. Conway, Introduction to VLSI Systems, Addison-Wesley, Reading, Mass., 1979.

8. C.D. Thompson, "A complexity theory for VLSI" Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Penn., September 1979.

9. D. Heller, "A Survey of parallel algorithms in numerical linear algebra," Dept. of Comp. Sci. Carnegie-Mellon University, Pittsburgh, Pa., Feb. 1976.

10. L. Csanky, "Fast parallel matrix inversion algorithms, SIAM J. Computing 5, 1976, 618-623.

VLSI COMPUTING STRUCTURES FOR SOLVING LARGE-SCALE LINEAR SYSTEM OF EQUATIONS

Kai Hwang
School of Electrical Engineering
Purdue University
W. Lafayette, Indiana 47907

Yen-Heng Cheng
Department of Computer
Engineering and Science
Tsinghua University
Peking, China

Abstract -- Gaussian elimination for solving large-scale linear system of algebraic equations is realized with pipelined VLSI cellular arithmetic networks. VLSI arrays are proposed for L-U decomposition of a dense matrix with pivoting, for triangularizing a given linear system $\underline{A} \cdot \underline{x} = \underline{b}$ for pipelined solution, for obtaining the inverse of a triangular matrix, and for matrix multiplication used in solving a family of linear systems. Modular network realizations of the proposed VLSI computing structures are presented emphasizing practical packaging constraints. Structural complexity, expandability, speed analysis, memory and I/O requirements of the proposed VLSI architectures are also discussed.

1. INTRODUCTION

Finding fast, accurate, and cost-effective methods to solve a large scale Linear System of Equations (LSE), in the form $\underline{A} \cdot \underline{x} = \underline{b}$, has been highly demanded for centuries by scientists and engineers. Due to lengthy sequences of arithmetic computations, most large LSEs are solved on high-speed digital computers using well-developed software packages such as the ALGOL-60, FORTRAN, Extended ALGOL, and PL/1 programs described in Forsythe and Moler 4 . Two major difficulties arise in solving LSEs on general-purpose digital computers by software programs. (a) The main memory is not large enough to accommodate a very large system matrix $\underline{A}$. Henceforth, many time-consuming I/O transfers are needed in addition to the CPU computation time. (b) With fixed word length in digital computers, rounding errors in algebraic processes if not properly controlled may cause serious loss of accuracy leading to unreliable solutions.

In order to alleviate these problems presented by software means, the use of parallel computers (SIMD or MIMD machines) for solving LSEs has been studied by Csanky [3], Stone [20], Chen and Kuck [2], Orcutt [15], Sameh and Brent 18 , Sameh and Kuck [19], and Kant and Kimura [10]. The rapid advent of Very Large-Scale Integration (VLSI) technology has created a new architectural horizon in implementing parallel algorithms directly in hardware 13,14,21 . This possibility has created a new research front on VLSI computing structures, as reported in Rem and Mead [17], Kung and Leiserson [11], Kung [12], Foster and Kung [6], and Horowitz 5 . In particular, Kung and Leiserson have proposed the systolic arrays for L-U decomposition without pivoting [11]. Practical issues on packaging constraints, memory and I/O supports, and modular implementations are still open problems towards the eventual realization of VLSI computing structures.

In this paper, a new class of VLSI cellular arithmetic arrays is presented for solving LSE in a synchronous and pipelined fachion. The proposed VLSI architectures are structured differently from systolic arrays, even both using similar building cells. Listed below are numerical tasks to be realized with the proposed VLSI computing structures.

(1). L-U Decomposition of a Matrix with or without Pivoting.

(2). System Triangularization and Pipelined Solution of LSEs.

(3). Matrix Inversion and Matrix Multiplication.

The proposed VLSI arrays and networks can be applied to any dense matrices that are nonsingular. All the processing cells are kept busy all the time, Higher accuracy and system stability can be achieved with maximum column pivoting. The modularity of the proposed VLSI arrays offers better expandability, maintenance and application flexibilities. Computational procedures in Gaussian elimination with and without pivoting are described in section 2. Followed are various VLSI structures and their operational considerations. Finally, complexity, expandability, speed, accuracy, memory and I/O supports, and performances of the VLSI arithmetic devices are studied.

2. NUMERICAL COMPUTATIONS FOR SOLVING LINEAR ALGEBRAIC SYSTEMS

An LSE is characterized by a pair $(\underline{A}, \underline{b})$, where $\underline{A} = (a_{ij})$ is an $n \times n$ matrix, $\underline{b} = (b_1, b_2, \dots, b_n)^T$ is a column vector, and n is the order of the LSE. The problem of solving an LSE of order n is to find a vector $\underline{x} =$

CH1569-3/80/0000-0217$00.75 © 1980 IEEE

$(x_1, x_2, \ldots, x_n)^T$ which satisfies

$$\underline{A} \cdot \underline{x} = \underline{b} \qquad (1)$$

The solution $\underline{x}$ is unique, if and only if $\underline{A}$ is nonsingular. We shall consider only strongly nonsingular systems, in which all the diagonal submatrices of $\underline{A}$ are nonsingular.

For each nonsingular matrix $\underline{A} = (a_{ij})$ there exists an inverse matrix $\underline{A}^{-1} = (c_{ij})$ of $\underline{A}$ such that $\underline{A}^{-1} \cdot \underline{A} = \underline{A} \cdot \underline{A}^{-1} = I$, where $\underline{I}$ is the identity matrix. The solution vector $\underline{x}$ can be obtained by leftmultiplication both sides of Eq.1 by $\underline{A}^{-1}$

$$\underline{x} = \underline{A}^{-1} \cdot \underline{b} \qquad (2)$$

If $\underline{A}^{-1}$ is known, it requires n^2 multiplications and $n(n-1)$ additions to compute the n components of $\underline{x}$. However, to find the inverse $\underline{A}^{-1}$ is quite complicated and should be avoided if unnecessary 4 .

Using the Gaussian elimination method, one can systematically decompose $\underline{A}$ into two triangular matrices $\underline{L}$ and $\underline{U}$ such that

$$\underline{A} = \underline{L} \cdot \underline{U} \qquad (3)$$

where $\underline{L} = (\ell_{ij})$ is a lower triangular matrix with all diagonal elements equal to 1, and $\underline{U} = (u_{ij})$ is an upper triangular matrix with nonzero diagonal elements. Such an L-U decomposition is unique, if and only if $\underline{A}$ is strongly nonsingular. We shall show the calculations of elements ℓ_{ij} and u_{ij} along with the proposed VLSI arrays.

The sequence of Gaussian elimination operations transforms the dense system $\underline{A} \cdot \underline{x} = \underline{b}$ into an equivalent triangular LSE characterized by

$$\underline{U} \cdot \underline{x} = \underline{L}^{-1} \cdot \underline{b} = \underline{d} \qquad (4)$$

With this triangularized system, one can compute the solution vector $\underline{x}$ by

$$\underline{x} = \underline{U}^{-1} \cdot (\underline{L}^{-1} \cdot \underline{b}) = \underline{U}^{-1} \cdot \underline{d} \quad (5)$$

The inverse matrices $\underline{U}^{-1}$ and $\underline{L}^{-1}$ always exist, because $\underline{U}$ and $\underline{L}$ are nonsingular. We have expressed $\underline{L}^{-1} \cdot \underline{b} = \underline{d}$. Equation 5 can be also obtained from Eq.2 by the fact $\underline{A}^{-1} = (\underline{L} \cdot \underline{U})^{-1} = \underline{U}^{-1} \cdot \underline{L}^{-1}$.

The inversion of a triangular matrix requires much less computations than that of an arbitrary dense matrix. We shall show how to compute $\underline{d} = \underline{L}^{-1} \cdot \underline{b}$ and $\underline{x} = \underline{U}^{-1} \cdot \underline{d}$ directly with VLSI hardware. Gaussian elimination procedure automatically produces the new coefficient vector $\underline{d}$ without an explicit evaluation of $\underline{L}^{-1}$. In fact, if $\underline{U} = (u_{ij})$ is known, the solution given in Eq.5 involves the following recursive computations.

$$x_n = d_n/u_{nn}$$

$$x_i = (d_i - \sum_{j=i+1}^{n} u_{ij} \cdot x_j)/u_{ii}$$

$$\text{for } i = n-1, n-2 \cdots, 2, 1 \qquad (6)$$

This combined computation of elements (u_{ij}) and (d_j) can be done directly by the same ensemble of VLSI array. Gaussian elimination without pivoting (nature ordering of elimination) requires $n(n^2-1)/3$ operations to yield a triangular LSE. An operation here implies a multiplication-addition pair. It takes $n(n+1)/2$ operations to solve one triangular LSE using the above recursions. Frequently, one needs to compute a family of LSEs characterized by the same $\underline{A}$ matrix over different coefficient vectors $\underline{b}_k = (b_{1k}, b_{2k}, \ldots, b_{nk})^T$, for $k=1,2,\ldots,m$. The family of LSEs $\underline{A} \cdot \underline{x}_k = \underline{b}_k$ for $k=1,2,\ldots,m$ requires to repeat similar computations m times. We propose to solve such family of LSEs with a pipelined VLSI multiplication array in 2n operation cycles. In contrast, to solve m LSEs of order n on a uniprocessor system requires to perform $mn^2+(n^3-n)/3$ operations sequentially.

3. L-U DECOMPOSITION OF A DENSE MATRIX

Two VLSI processor arrays are proposed below to realize the Gaussian elimination method for L-U decomposition; one corresponds to Gaussian elimination with natural ordering (no pivoting), and the other corresponds to with maximal column pivoting. For clarity purpose, the decomposition procedure is presented by an example LSE of order n=4.

$$\underline{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{21} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \qquad (7)$$

The two triangular matrices, L and U, assume the following forms.

$$\underline{L} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \ell_{21} & 1 & 0 & 0 \\ \ell_{31} & \ell_{32} & 1 & 0 \\ \ell_{41} & \ell_{42} & \ell_{43} & 1 \end{bmatrix} \qquad (8.a)$$

$$\underline{U} = \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix} \qquad (8.b)$$

The following recursions are embedded in Gaussian elimination procedures without pivoting.

$$a'_{ij} = a_{ij} - (a_{i1}/a_{11}) \times a_{1j} \text{ for } i,j = 2,3,4$$

$$a''_{ij} = a'_{ij} - (a'_{i2}/a'_{22}) \times a'_{2j} \text{ for } i,j = 3,4 \qquad (9)$$

$$a'''_{ij} = a''_{ij} - (a''_{i3}/a''_{33}) \times a''_{3j} \text{ for } i,j = 4$$

The entries of L and U matrices are obtained in terms of these three sets of recursively generated coefficients.

$$\ell_{i1} = a_{i1}/a_{11} \quad \text{for } i = 2,3,4$$
$$\ell_{i2} = a'_{i2}/a'_{22} \quad \text{for } i = 3,4 \qquad (10.a)$$
$$\ell_{i3} = a''_{i3}/a''_{33} \quad \text{for } i = 4$$

$$u_{1j} = a_{1j} \quad \text{for } j = 1,2,3,4$$
$$u_{2j} = a'_{2j} \quad \text{for } j = 2,3,4 \qquad (10.b)$$
$$u_{3j} = a''_{3j} \quad \text{for } j = 3,4$$
$$u_{4j} = a'''_{4j} \quad \text{for } j = 4$$

The above recursions require to repeatedly perform Multiply, Divide and subtract operations in n=4 iterative steps. Two types of arithmetic cells as shown in Fig.1 are required to perform these basic computations. One is the M cell for additive multiplication specified by arithmetic equations d = c + a * b, a = a and b = b. The other type is called D cell for division specified by g = e/f and f = f. Note that a small circle at the input or output terminal of an arithmetic cell means an arithmetic negation, such as a two's complement operation. No registers are assumed within each cell. Instead, we use latch registers between segments of arithmetic cells.

A VLSI array for L-U decomposition without pivoting is shown in Fig.1. For a general LSE of order n, $(n-1)^2$ M cells and n-1 D cells are needed in the cellular array construction. In order to facilitate synchronous pipelined operations, fast interface latches are used between Segments (rows) of arithmetic cells.

Input operands are from the elements (a_{ij}) of matrix A, feeding in one column per each cycle as shown. The VLSI pipeline has two-way traffic flows. Data streams flow first upward. After reaching the top segment of division cells, data streams then flow downward. Due to this two-way pipelining, a number of dummy numeric zeros and ones are interleaved with the matrix elements (a_{ij}) in the input data streams. The array outputs are elements (u_{ij}), (ℓ_{ij}) of matrices U and L with also some dummy interspaced zeros. The proper timing of segment delays is marked at the side for each cycle of the pipeline. In general, 3n-2 pipeline cycles are needed to generate all the elements of L and U matrices. Between successive applications of the two-way pipeline, start-up delays of n-1 cycles are needed to drain the pipeline as shown by the first three cycles (t_1 through t_3) in Fig.1.

Gaussian elimination with natural order may result in serious accuracy loss problem. For an example, the division of a product terms by a very small number (Eqs. 9 and 10) may cause overflow beyond the precision limit of the machine. Therefore, we wish to choose the maximal divisors, called pivot elements, in the elimination process. The maximal pivots, selected among all remaining rows and columns of the matrix being triangularized, will cause least loss of accuracy. We implement the maximal columns pivoting, which searches for the maximal element only among each remaining column. This is especially convenient, because the elements of A are fed into the pipeline by column as demonstrated in Fig.1.

A VLSI array is proposed in Fig.2 for L-U decomposition with maximal column pivoting. This array is modified from the array in Fig.1 by adding additional pivot selection logic. The Pivot Indicator (PI) is a logic device which indicates the maximal among a column of matrix elements. The outputs of PI are Boolean values, "1" signaling the location of the maximal column pivot and "0" for the rest elements. The successive outputs of PI for j = 1,2, 3,4 are labeled by I_{ij} for segments i = 1,2,3,4. The Pivot Exchange (PE) unit has 8 inputs, four of which are the column elements and the other four are the corresponding Boolean indicators (I_{ij}) from the outputs of the PI on the right of the drawing. The PE will interchange at its output the indicated pivot element with the leftmost column element. The non-pivot input elements are passed to the corresponding output lines unchanged. In other words, the PE will always output the pivot element at its leftmost output line. When the original leftmost input is itself the indicated pivot, no exchange will be made and the PE will simply pass all its inputs unchanged to the corresponding outputs. Details of the pivoting logic can be found in Ref.[9]. The input/output of the array in Fig.2 is labeled in Table 1.

4. SYSTEM TRIANGULARIZATION AND SOLUTION PIPELINE

Substituting Eq.3 into Eq.1, we obtain $\underline{L} \cdot (\underline{U} \cdot \underline{x}) = \underline{b}$. This actually represents two triangular systems interlocking each other. The forward elimination corresponds to the lower triangular system.

$$\underline{L} \cdot \underline{d} = \underline{b} \qquad (11.a)$$

and the backward substitution corresponds to the upper triangular system

$$\underline{U} \cdot \underline{x} = \underline{d} \qquad (11.b)$$

The solutions of these two triangular systems will lead to the final solution vector $\underline{x}$. In this section, we wish not to compute the inverses $\underline{L}^{-1}$ and $\underline{U}^{-1}$ to obtain the solution vector $\underline{x}$.

Shown below is a VLSI array for triangularizing $\underline{A}$ into $\underline{U}$ and at the same time obtaining the new coefficient vector $\underline{d}$ and $\underline{L}$. Let us rewrite Eq.11.a for the example LSE of order 4.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ \ell_{21} & 1 & 0 & 0 \\ \ell_{31} & \ell_{32} & 1 & 0 \\ \ell_{41} & \ell_{42} & \ell_{43} & 1 \end{bmatrix} \cdot \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \qquad (12)$$

We relabel the column vector $(b_1, b_2, b_3, b_4)^T = (a_{15}, a_{25}, a_{35}, a_{45})$. Using the similar formulation of Eq.9, we can extend the column index to j=5 to obtain the following recursions with six additional terms.

$$\begin{aligned} a'_{ij} &= a_{ij} - (a_{i1}/a_{11}) \times a_{1j} \\ &\quad \text{for } i = 2,3,4 \text{ and } j = 2,3,4,5 \\ a''_{ij} &= a'_{ij} - (a'_{i2}/a'_{22}) \times a'_{2j} \\ &\quad \text{for } i = 3,4 \text{ and } j = 3,4,5 \\ a'''_{ij} &= a''_{ij} - (a''_{i3}/a''_{33}) \times a''_{3j} \\ &\quad \text{for } i = 4 \text{ and } j = 4,5 \end{aligned} \qquad (13)$$

The solutions of the forward triangular system (Eq.12) can be recursively computed below using Eq.10.a and Eq.13.

$$\begin{aligned} d_1 &= b_1 = a_{15} \\ d_2 &= (b_2 - \ell_{21} \cdot b_1) = a'_{25} \\ d_3 &= (b_3 - \ell_{31} \cdot b_1) - \ell_{32} \cdot (b_2 - \ell_{21} \cdot b_1) \\ &= a'_{35} - \ell_{32} \cdot a'_{25} = a''_{35} \\ d_4 &= a''_{45} - \ell_{43} \cdot a''_{35} = a'''_{45} \end{aligned} \qquad (14)$$

Note the similarity between the expressions in Eq.14 and those for (u_{ij}) in Eq.10.b. We can use an extended VLSI structure to compute Eq.14 modified from the previous L-U decomposition arrays (Fig.1). This array as detailed in Ref. [9] can simultaneously generate the elements of matrix $\underline{U}$ and of vector $\underline{d}$ without explicitly computing the elements of inverse matrix $\underline{L}^{-1}$. Such an array can directly convert the original LSE into an equivalent triangular system. The elements of matrix $\underline{A}$ and of vector $\underline{b}$ serve as the inputs and elements of $\underline{U}$ and vector $\underline{d}$ are the outputs.

The solution of the upper triangular system (Eq.11.b) has already recursively specified in Eq.6 in terms of (d_j) and (u_{ij}). The VLSI array shown in Fig.3 is specially designed to generate the solution vector $\underline{x}$ of the LSE. In order to connect the outputs of the triangularizing array directly to the inputs of this LSE solver, some precautions must be made to provide the necessary interface delays to match the speed of two data streams.

5. MATRIX INVERSION AND MULTIPLICATION

In application where repeated solutions of $\underline{A} \cdot \underline{x}_k = \underline{b}_k$ are needed over a set of coefficient vectors $\underline{b}_k$ for k=1,2,...,m, the use of the inverse matrix $\underline{A}^{-1}$ may become very attractive to generate the set of solution vectors via the sequence of computations $\underline{x}_k = \underline{A}^{-1} \cdot \underline{b}_k$ for k=1,2,...,m. In this sequence, the inverse $\underline{A}^{-1}$ need be computed only once. According to Eq.5, one can reduce the problem (after Gaussian elimination) to as follows:

$$\begin{aligned} \underline{x}_k &= \underline{A}^{-1} \cdot \underline{b}_k = (\underline{L} \cdot \underline{U})^{-1} \cdot \underline{b}_k \\ &= (\underline{U}^{-1} \cdot \underline{L}^{-1}) \cdot \underline{b}_k \text{ for } k=1,2,\ldots,m \end{aligned} \tag{15}$$

We present below VLSI pipelines for finding the inverse matrix $\underline{L}^{-1} = (m_{ij})$ $\underline{L} = (\ell_{ij})$ and the inverse matrix $\underline{U}^{-1} = (v_{ij})$ from $\underline{U} = (u_{ij})$. Based on the triangular forms of $\underline{L}$ and $\underline{U}$ specified in Eq. 8, $\underline{L}^{-1}$ and $\underline{U}^{-1}$ will be also triangular matrices.

$$\underline{L}^{-1} = (m_{ij}) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ m_{21} & 1 & 0 & 0 \\ m_{31} & m_{32} & 1 & 0 \\ m_{41} & m_{42} & m_{43} & 1 \end{bmatrix} \tag{16.a}$$

$$\underline{U}^{-1} = (v_{ij}) = \begin{bmatrix} v_{11} & v_{12} & v_{13} & v_{14} \\ 0 & v_{22} & v_{23} & v_{24} \\ 0 & 0 & v_{33} & v_{34} \\ 0 & 0 & 0 & v_{44} \end{bmatrix} \tag{16.b}$$

Matrix multiplication is performed to obtain the inverse matrix $\underline{A}^{-1} = \underline{U}^{-1} \cdot \underline{L}^{-1}$ from $\underline{U}^{-1}$ and $\underline{L}^{-1}$. Let $\underline{e}_k$ be the unit column vector whose components are all zero except the k-th component, which is one. The columns of $\underline{U}^{-1}$ are simply the respective solutions of the following n LSEs.

$$\underline{U} \cdot \underline{Y}_k = \underline{e}_k \text{ for } k=1,2,\ldots,n \tag{17}$$

One can write the n column vectors $\underline{Y}_k$ and $\underline{e}_k$ into a matrix form as $\underline{Y} = (\underline{Y}_1, \underline{Y}_2, \ldots, \underline{Y}_k)$ and $I = (\underline{e}_1, \underline{e}_2, \ldots, \underline{e}_n)$, where $\underline{I}$ is the identity matrix. Then the family of LSEs in Eq.17 can be rewritten into the compact form $\underline{U} \cdot \underline{Y} = \underline{I}$. This implies that $\underline{Y} = \underline{U}^{-1}$. The following recursive formula is used to compute the elements (v_{ij}) of the inverse U^{-1} from a given matrix $\underline{U} = (u_{ij})$.

$$\begin{aligned} v_{kk} &= \frac{1}{u_{kk}} \quad \text{for } k=1,2,\cdots,n \\ v_{ij} &= -\left[\sum_{k=i+1}^{j} u_{ik} \times v_{kj}\right] \cdot u_{ii}^{-1} \quad \text{for all } j > i \end{aligned} \tag{18}$$

For the LSE of order n=4, we have

$$\begin{aligned} v_{kk} &= 1/u_{kk} \quad \text{for } k=1,2,3,4 \\ v_{12} &= -(u_{12} \times v_{22})/u_{11} \\ v_{13} &= -(u_{12} \times v_{23} + u_{13} \times v_{33})/u_{11} \\ v_{14} &= -(u_{12} \times v_{24} + u_{13} \times v_{34} + u_{14} \times v_{44})/u_{11} \\ v_{23} &= -(u_{23} \times v_{33})/u_{22} \\ v_{24} &= -(u_{23} \times v_{34} + u_{24} \times v_{44})/u_{22} \\ v_{34} &= -(u_{34} \times v_{44})/u_{33} \end{aligned} \tag{19}$$

The VLSI array for finding the inverse matris $\underline{L}^{-1}$ is shown in Fig.4. Similar array can be obtained for finding $\underline{U}^{-1}$ from $\underline{U}$ as detailed in [9].

We present next a VLSI pipelined array of M cells for the multiplication of two arbitrary dense matrices. Obviously, this array can be used to compute the inverse matrix $\underline{A}^{-1}$ by performing the multiplication $\underline{U}^{-1} \cdot \underline{L}^{-1}$. The array structure is depicted by the multiplication of two 3 x 3 square matrices.

$$\underline{A} \times \underline{B} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}$$

$$= \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix} = \underline{C} \qquad (20)$$

, where the product coefficients $c_{ij} = \sum_{k=1}^{3} a_{ik} \cdot b_{kj}$ for all i and j.

The rectangular array design is shown in Fig.5. The elements of matrices $\underline{A}$ and $\underline{B}$ are fed from the lower and upper input lines in a pipelined fachion, one skewed row or one skewed column at a time. Some dummy zero inputs are interspaced with the matrix elements. In general, to multiply two n x n matrices requires n(2n - 1) multiply cells (M cells). The start-up delay corresponds to the longest path on this array, which equals 2n-1 clock periods. This array differs from the systolic array [11] in both interconnection structure and the way inputs are applied and outputs are retrieved. If one counts the start-up delays, the time required to produce the last product term c_{nn} (c_{33} in Fig.5) equals 4n-2 clock periods.

For triangular matrices, such as $\underline{U}^{-1}$ and $\underline{L}^{-1}$ specified in Eq.16, the full multiplication array must be used. This is due to the fact that their product matrix $\underline{A}^{-1} = \underline{U}^{-1} \cdot \underline{L}^{-1}$ is, in general, an arbitrary dense matrix.

The collection of m column-vector solutions $\underline{x}_k = (x_{1k}, x_{2k}, \ldots, x_{nk})^T$ for k=1,2,...,m specified in Eq.15 can be generated in pipelined fashion by carrying out the following matrix multiplication.

$$\underline{X} = \underline{A}^{-1} \cdot \underline{B} \qquad (21)$$

where

$$\underline{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1m} \\ x_{21} & x_{22} & \cdots & x_{2m} \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ x_{n1} & x_{n2} & \cdots & x_{nm} \end{bmatrix} \qquad (22)$$

$$\underline{B} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ b_{n1} & b_{n2} & \cdots & b_{nm} \end{bmatrix} \qquad (23)$$

and $\underline{A}^{-1} = (c_{ij})$ is the inverse matrix of $\underline{A} = (a_{ij})$, generated by the multiplication array of Fig.5. When m = n, one can simply reuse the array of Fig.5 to compute the solution matrix $\underline{X}$. When m > n, the multiplication array must be expended in one of the two dimensions, say adding more rows.

Since the elements (c_{ij}) of the inverse matrix $\underline{A}^{-1}$ will be repeatedly used, we have devised a special VLSI array for carrying out the multiplication specified in Eq.21. This alternate array being reported in [9] is singly pipelined only in the vertical direction. The (c_{ij}) entries are fed through a fan-in demultiplexer and distributed to all M cells. The column elements of the matrix $\underline{B}$ are fed through the vertical inputs, one skewed column at a time via a fan-in multiplexer. After the first solution $\underline{x}_1 = (x_{11}, x_{21}, \ldots, x_{n1})^T$ appears at the output end, one solution vector will appear at each additional cycle. The attractive part of this array is that it is applicable to any number of m LSEs in a pipelined fashion.

6. MODULAR NETWORKING OF VLSI COMPUTING STRUCTURES

VLSI devices must grow gradually. It is by far constrained by chip density, packaging area, and pin limitations. To built a "very large" LSE solver, say of order 10^3 or greater, on a monolithic chip depends on how these constraints can be overcome. Extensive development efforts are still needed to develop VLSI modules, which can be interconnected to form a network LSE solver. We propose below two concrete examples of such a networking approach. The first example shows the modularization of rectangular VLSI computing arrays and the second one for triangular VLSI arrays.

A general LSE of order n requires 4n-2 I/O ports in Fig.1, each of which has a width of w bits (equal to the operand length). For large n and w, this implies an exceedingly large number of external leads on the VLSI chip. Obviously, the projected IC packaging technology renders such VLSI arrays unrealistic. To alleviate this problem of constrained I/O leads, a fan-in demultiplexing and fan-out multiplexing scheme is depicted in Fig.6 for general VLSI computing structure with $k \times m$ parallel inputs and $r \times s$ parallel outputs. After the I/O multiplexing, reasonably small numbers of k inputs and r outputs are allowed at the I/O ends. In order to ensure proper serial-to-parallel and parallel-to-serial conversions as demonstrated, at least two clocks, C_1 and C_2, are needed per each VLSI device. The clock C_1 is used to control data in and out of the input and output registers respectively. The array clock C_2 has a period equal to km or rs times that of the period p of clock C_1. The multiplicity reflects the degree of multiway conversion logic used at the I/O ends. C_2 is the array clock controlling all the latches in the VLSI array. The timing relationships of the two clock signals are demonstrated in the lower half of Fig.6. The actual numbers, k and r, of inputs and outputs are determined by the chip packaging requirement.

The rectangular VLSI array in Fig.1 can be partitioned into two types of VLSI modules as shown in Fig.7. Using these two module types, one can construct an L-U decomposition networks of arbitrary high orders. The <u>multiply module</u>, $M(q \times q)$ corresponds to a q-by-q subarray of multiply cells in Fig.1. The <u>division Module</u>, $D(q)$, corresponds to the top row of division cells in Fig.1. Multiple number of division modules can be fabricated on the same chip, say q $D(q)$ modules on a chip, which would be comparable in complexity with one $M(q \times q)$ module.

In Fig.7, four $M(q \times q)$ modules and two $D(q)$ modules are used to construct an <u>L-U decomposition network</u> $LU(2q + 1)$ for an LSE of order $n = 2q + 1$. In general, an $LU(n)$ network of order $n = p \cdot q + 1$ requires p^2 $M(q \times q)$ modules and p $D(q)$ modules. One additional demultiplexer/multiplexer pair is needed between the network modules and the external memory system, where the operands and results suppose to reside.

The partition of triangular VLSI arrays into a network of VLSI modules is exemplified by a modular matrix inversion scheme. The interconnection of three triangular multiply modules $T(q)$ and three square multiply modules $S(q \times q)$ shown in Fig.8 produces a <u>matrix inversion network</u>, $MI(3q)$ for computing the inverse matrix $\underline{L}^{-1} = (m_{ij})$ of a lower triangular matris $\underline{L} = (\ell_{ij})$ of order $n = 3q$.

In general, a matrix inversion network $MI(n)$ of order $n = p \cdot q$ requires p $T(q)$ modules and $p \cdot (p - 1)/2$ $S(q \times q)$ modules. Datail designs of there VLSI modules can be found in [9].

7. STRUCTURAL COMPLEXITY AND PERFORMANCE ANALYSIS

It has been predicted that by the late 80's it will be possible to fabricate IC chips, each of which contains 10^7 or 10^8 indiviaual transistors [14,17]. New high-resolution lithographic techniques have already demonstrated the feasibility of achieving such VLSI devices with NMOS technology. The VLSI computing structures require not only large number of processing cells and latch memory, but also large number of conducting paths for communicating information throughout the integrated system. The length and organization of these communication paths set a lower bound on the chip area and time delay required for system operations [17]. Furthermore, the I/O and packaging constraints of monolithic VLSI chips has set limitations to the applicability of VLSI chips in digital system design.

The structural complexity of VLSI computing structures is estimated at the logical level in terms of the number of processing cells used in a schematic array layout or in terms of VLSI modules used in a network construction. The potential speed of a VLSI device is determined by the total clock periods needed for a specific computation sequence. We lump the path delays into the cell delays. The multiply cells (M cells) can each assume the global cellular structure of carry-save adders as in Hwang [7]. The division cells (D cells) can assume the cellular structures suggested by Cappa and Hamacher [1] and also those described in Hwang [8]. We firmly believe that the use of interface latches instead of registers in cells will better facilitate the control of pipelined operations.

With the same word length, the M cells and D cells should have about equal time delay, say Δ time units per cell of either type. It is now possible to achieve 24-bit-by-24-bit multiplication of division with LSI bipolar cellular arrays in less than 200 nanoseconds. The delay of the pivoting logic per each pipeline segment in Fig.2 is denoted by δ time units. The interface latch delay is negligible, when compared with Δ or δ. Therefore, the segment delay between two adjacent latches, equals $\Delta + \delta$ or Δ depending on whether pivoting is used or not. This means that the internal array clock (C_2 in Fig.2) of the pipeline may have a period p equal to Δ or $\Delta + \delta$.

Consider an LSE of order n. The numbers of arithmetic cells (either M or D cells) required in each of the presented VLSI arrays are summarized in Table 2. The numbers of I/O terminals are also shown. These I/O terminal counts refer to the parallel inputs and parallel outputs to or from the internal VLSI array before using the fan-in and fan-out conversion interfaces as demonstrated in Fig.6. The start-up delays for draining the array pipelines and the net compute time are expressed in terms of parameters n, Δ, and δ. The sum of the start-up delay and the compute time equals the total compute time required to complete the specified sequence of computations. In all cases, the total compute time of each of the proposed VLSI arithmetic pipelines is linearly proportional to the order n of the LSEs. This implies a speedup from $O(n^2)$ or $O(n^3)$ operations required in a serial computer to $O(n)$ steps using VLSI computing networks. For large values of n, the speedup is rather impressive.

The proposed VLSI arrays are expandable to allow modular growth. The L-U decomposition arrays and the system triangularization array can be each expanded by adding more rows of M cells at the bottom and extending the lengths of all rows to the right. Without pivoting logic, such extension can be done by using modules as demonstrated in Section 6. With pivoting, the array must be expanded into the third dimension in order to achieve modularization. Pivoting will increase the accuracy and stability of the solution to an LSEs. This is an improvement over the systolic arrays. Any dense system with a "strongly" nonsingular matrix $\underline{A}$ can be solved by the proposed VLSI networks.

The modular requirements for constructing VLSI computing networks demonstrate a tradeoff between module sizes q in M(q $\times$ q), D(q), and S(q $\times$ q), and the network sizes n in LU(n) or MI(n). The proper choice depends heavily on the VLSI technology and packaging capability. We have assumed the continuous supply of operands either from the main memory or from a cache memory. The operand supply rate may be slower than the array processing rate. For small matrix $\underline{A}$, this problem can be solved by using a large cache memory. However, large data buffer may increase the cost of the computer system significantly. The I/O interface structure matches the speed of VLSI devices and that of memories from which the matrix or vector elements are retrieved.

8. CONCLUSIONS

We have proposed a complete set of VLSI arithmetic array and network architectures for implementing Gaussian elimination method to solve LSEs. The L-U decomposition is realized in hardware with and without pivoting. The triangularization and solution of a dense linear system are realized directly with hardware arrays without explicitly finding the inverse matrix $\underline{A}^{-1}$. For solving a family of LSEs characterized by the same matrix $\underline{A}$, we have proposed the matrix inversion and multiplication arrays for generating the sequence of solutions in a pipelined fashion. Modular networking and efficient I/O structures are also presented for VLSI computing structures. Continued efforts are being exerted on the development of bit-slice VLSI computing structures.

REFERENCES

[1] Cappa, M. and Hamacher, V. C., "An Augmented Iterative Array for High-Speed Binary Division", IEEE Trans. Computers, Vol. C-22, Feb. 1973, pp. 172-175.

[2] Chen, S. C. and Kuck, D. J., "Time and Processor Bounds for Linear Recurrence Systems", IEEE Trans. Computers, C-24, 1975, pp. 701-717.

[3] Csanky, L., "Fast Parallel Matrix Inversion Algorithms", SIAM J. Computing, Vol. 5, 1976, pp. 618-623.

[4] Forsythe, G. and Moler, C. B., Computer Solution of Linear Algebraic Systems, Prentice-Hall, Inc., Englewood Cliffs, N.J. 1967.

[5] Horowitz, E., "VLSI Architecture for Matrix Multiplications", Proc. Int'l. Conf. on Parallel Processing, Aug. 1979, pp. 124-127.

[6] Foster, M. J. and Kung, H. T., "The Design of Special-purpose VLSI Chips", IEEE Computer Magazine, Jan. 1980, pp. 26-40.

[7] Hwang, K., "Global and Modular Two's Complement Array Multipliers", IEEE Trans. Computers, Vol. C-28, No. 4, April 1979, pp. 300-306.

[8] Hwang, K., Computer Arithmetic Principles, Architecture, and Design, John Wiley & Sons, Inc., New York, 1979, Chaps. 6 and 8.

[9] Hwang, K. and Y. H. Cheng, "VLSI Arithmetic Arrays and Modular Networks for Solving Large-Scale Linear System of Equations", TR-EE80-4, School of E.E., Purdue University, W. Lafayette, Indiana, March 1980.

[10] Kant, R. M. and Kimura, T., "Decentralized Parallel Algorithms for Matrix Computations", Proc. of the Fifth Annual Symp. on Computer Architecture, Palo Alto, CA, April 1978, pp. 96-100.

[11] Kung, H. T. and Leiserson, C. E., "Systolic Arrays (for VLSI)", in Spare Matrix Proc. 1978, Duff, I. S. and Stewart, G. W., Editors, Society for Indust. and Appl. Math., Philadelphia, Pa., 1979, pp. 256-282.

[12] Kung, H. T., "Let's Design Algorithms for VLSI Systems", Proc. Caltech. Conf. Very Large Scale Integration, Cal. Inst. of Tech., Pasadena, Calif., Jan. 1979, pp. 65-90.

[13] Kung, H. T., "The Structure of Parallel Algorithms", in Advance in Computers, Vol. 19, (M. C. Yovits, ed.), Academic Press, N. Y., 1980.

[14] Mead, C. and Conway, L., Introduction To VLSI Systems, Addison-Wesley Pub. Co., Reading, Mass., 1980, Chap. 8.

[15] Orcutt, S. E., "Parallel Solution Methods for Triangular Linear Systems of Equations", Tech. Rept. 77, Digital System Lab., Stanford Electronics Labs, Stanford University, 1972.

[16] Ramamoorthy, C. V. and Li, H. F., "Pipeline Architecture", ACM Computing Surveys, Vol. 9, No. 1, March 1977, pp. 61-102.

[17] Rem, M. and Mead, C. A., "Cost and Performance of VLSI Computing Structures", IEEE J. of Solid-State Circuits, Vol. SC-14, April 1979, pp. 455-462.

[18] Sameh, A. and Brent, R., "Solving Triangular Systems on a Parallel Computer", SIAM J. of Numerical Analysis, Vol. 14, No. 6, Dec. 1977, pp. 1101-1113.

[19] Sameh, A. and Kuck, D., "On Stable Parallel Linear System Solvers", J. of ACM, Vol. 25, No. 1, Jan. 1978, pp. 81-91.

[20] Stone, H. S., "An Efficient Parallel Algorithms for The Solution of a Tridiagonal Linear System of Equations", J. of ACM, Vol. 20, No. 1, 1973, pp. 27-38.

[21] Sutherland, I. E. and Mead, C. A., "Microelectronics and Computer Science", Scientific American, Vol. 237, No. 3, Sept. 1977, pp. 210-228.

[22] Wilkinson, J. H., Rounding Errors in Algebraic Processes, Englewood Cliffs, N.J., 1963.

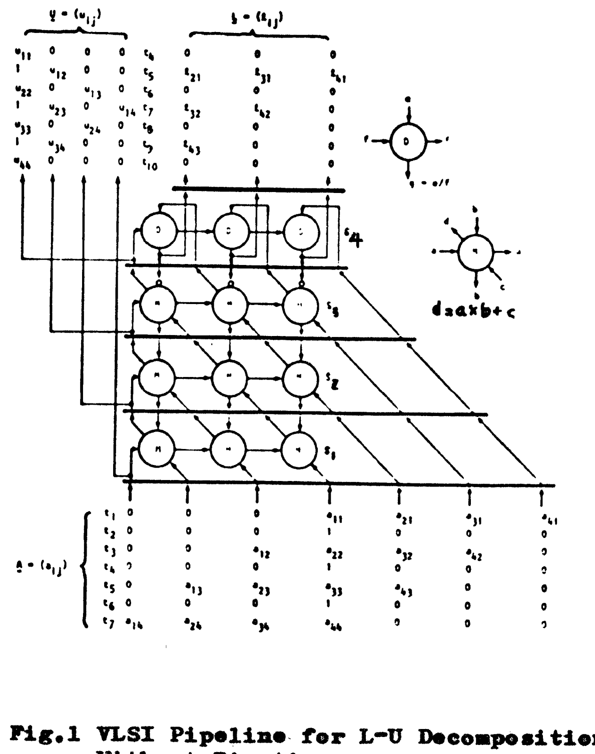
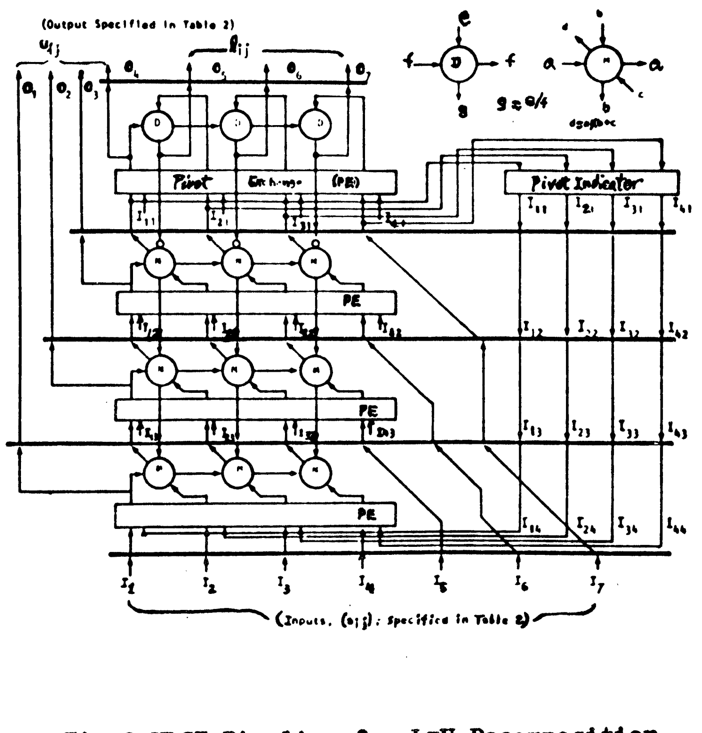


Fig.1 VLSI Pipeline for L-U Decomposition Without Piveting



(Output Specified in Table 2)
Pivot Exchange (PE)
Pivot Indicator
PE
g=e/f
d=a*b+c
(Inputs, (a_ij); Specified in Table 2)

Fig.2 VLSI Pipeline for L-U Decomposition With Column Pivoting

Table 1: I/O Labeling of the L-U Decomposition Pipeline in Fig.2

Time/Input	I_1	I_2	I_3	I_4	I_5	I_6	I_7
t_1 to t_6	0	0	0	1	0	0	0
t_7	0	0	0	a_{11}	a_{21}	a_{31}	a_{41}
t_8	0	0	0	1	0	0	0
t_9	0	0	a_{12}	a_{22}	a_{32}	a_{42}	0
t_{10}	0	0	0	1	0	0	0
t_{11}	0	a_{13}	a_{23}	a_{33}	a_{43}	0	0
t_{12}	0	0	0	1	0	0	0
t_{13}	a_{14}	a_{24}	a_{34}	a_{44}	0	0	0

Time/Output	O_1	O_2	O_3	O_4	O_5	O_6	O_7
T_{11}	0	0	0	u_{11}	ℓ_{21}	ℓ_{31}	ℓ_{41}
T_{12}	0	0	u_{12}	1	0	0	0
T_{13}	0	u_{13}	0	u_{22}	ℓ_{32}	ℓ_{42}	0
T_{14}	u_{14}	0	u_{23}	1	0	0	0
T_{15}	0	u_{24}	0	u_{33}	ℓ_{43}	0	0
T_{16}	0	0	u_{34}	1	0	0	0
T_{17}	0	0	0	u_{44}	0	0	0

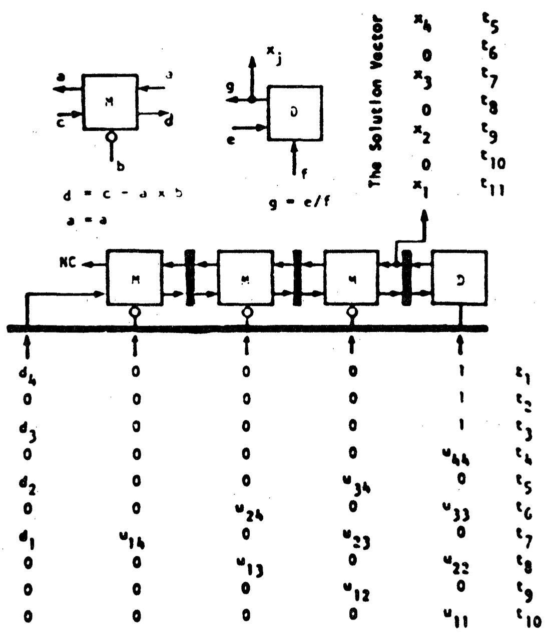


Fig.3 Pipelined VLSI Array for Solving a Triangularized LSE

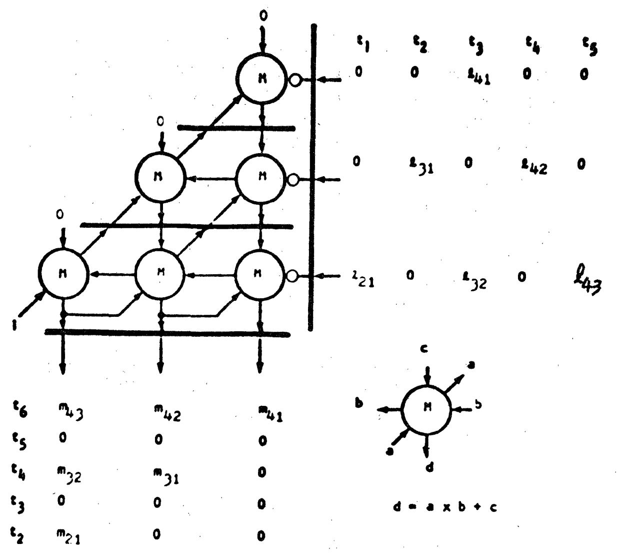


Fig.4 VLSI Pipeline For Matrix Inversion

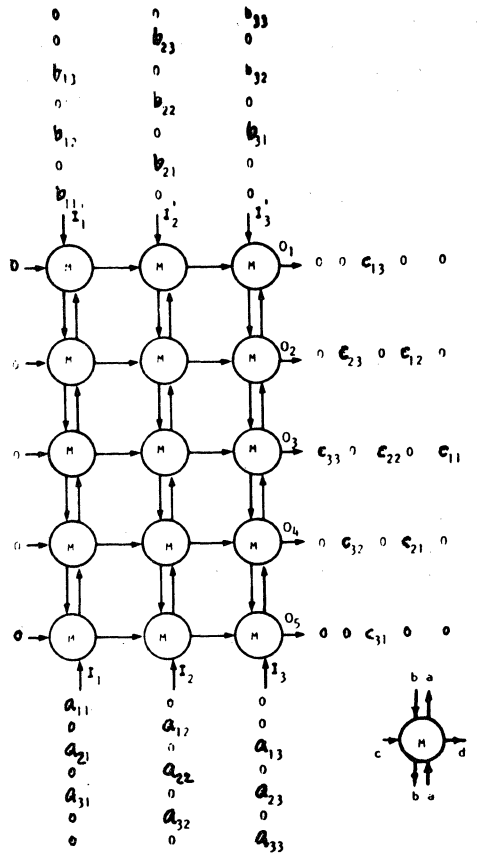


Fig.5 VLSI Array for Pipelined Multiplication of Two 3 3 Dense Matrices

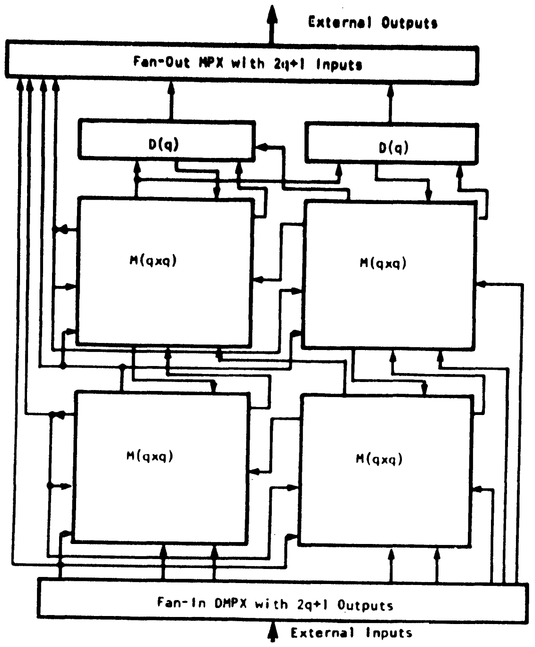


Fig.7 Interconnection of Four M(qxq) Modules and Two D(q) Modules to Form an L-U Decomposition Network LU(2q+1) of order 2q+1

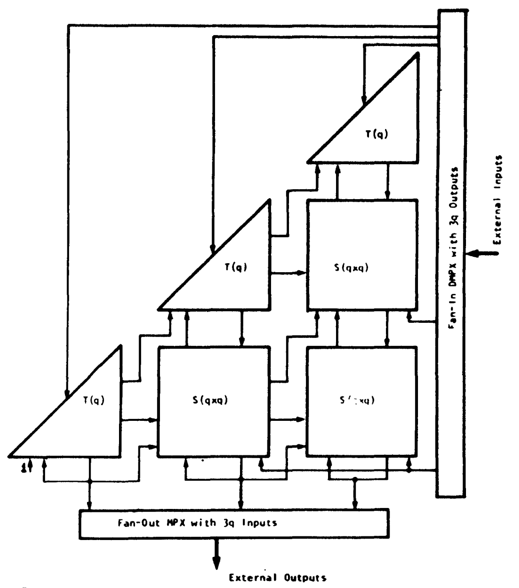


Fig.8 Interconnection of Three Square Multiply Modules S(q x q) and Three Triangular Modules T(q) to form A Matrix Inversion Network MI(3q) of order 3q

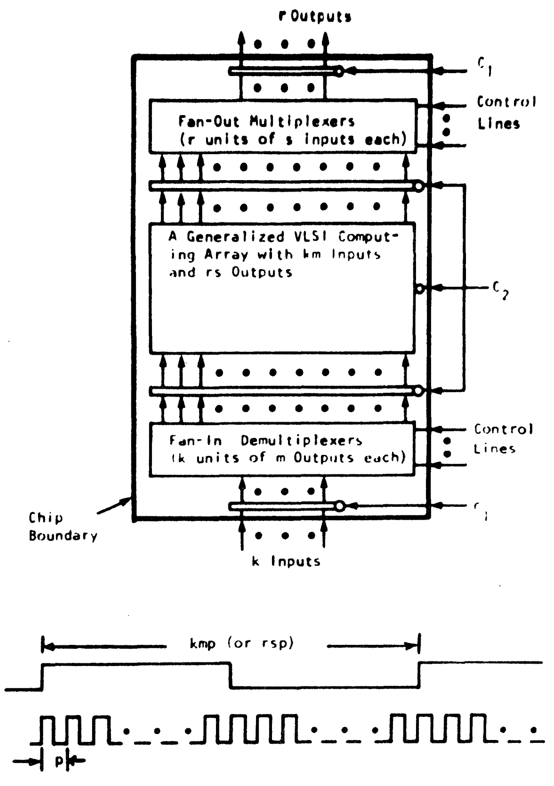

r Outputs
c_1
Control Lines
Fan-Out Multiplexers (r units of s inputs each)
A Generalized VLSI Computing Array with km Inputs and rs Outputs
c_2
Fan-In Demultiplexers (k units of m Outputs each)
Control Lines
c_1
Chip Boundary
k Inputs
kmp (or rsp)
c_2
c_1
p

Fig.6 I/O Multiplexing and Timing Control of VLSI Computing Structures

Table 2: Structural Complexity and Speed Performance of The VLSI Computing Structures

Type of VLSI Array	No. of Required Arithmetic Cells	No. of I/O Terminals	Start-Up Time Delay	Array Net Compute Time	Total Compute Time
L-U Decomposition Without Pivoting (Fig. 1)	$n^2 - n$	$4n - 2$	$n - 1$	$2n - 1$	$3n - 2$
L-U Decomposition With Pivoting (Fig. 2)	$n^2 - n$	$4n - 2$	$3n - 2$	$2n - 1$	$5n - 3$
Triangularization of LSEs	$n^2 - 1$	$4n$	n	$2n$	$3n$
Triangular Linear System Solver (Fig. 3)	n	$n + 2$	n	$2n - 1$	$3n - 1$
Inversion of Triangular Matrix $\underline{U}$	$\frac{n^2+n}{2}$	$2n$	1	$2n - 1$	$2n$
Inversion of Triangular Matrix $\underline{L}$ (Fig. 4)	$\frac{n^2-n}{2}$	$3n - 2$	1	$2n - 3$	$2n - 2$
Matrix Multiplication (Fig. 5)	$2n^2 - n$	$4n - 1$	$2n - 1$	$2n - 1$	$4n - 2$
Solving Family of LSEs Using Multiply Array	n^2	$n^2 + 2n$	n	n	$2n$

NOTE: Arithmetic cells include both M and D cells. All times are measured in terms of multiples of unit cell delay Δ, except in Fig. 3, where multiples of $\Delta + \delta$ are implied and δ is the delay of the pivoting logic.

SESSION 8: NONNUMERICAL ALGORITHMS AND APPLICATIONS

Simulation and Analysis in Deriving Time and Storage Requirements for a Parallel Alpha-Beta Algorithm

Selim G. Akl
David T. Barnard
Ralph J. Doran

Department of Computing and Information Science
Queen's University, Kingston, Ontario, Canada

Summary

Several recent papers have proposed parallel adaptations of the sequential alpha-beta algorithm [1, 2, 3, 6]. The present paper derives time and storage requirements for one such adaptation [1]. Alpha-beta search is fundamental to artificial intelligence research as many game playing programs employ it [4]. A simulation of a multiprocessor is used to derive timing requirements in terms of nodes visited, nodes scored, and elapsed time. The simulation environment includes hardware processors and software processes. Storage requirements for the algorithm are derived analytically. The tradeoff between time and storage "cost" in the algorithm is demonstrated.

The basis for our parallel implementation of the alpha-beta algorithm is the following: assuming that the tree to be searched is perfectly ordered, those nodes that must be scored are (concurrently) visited first. The algorithm is designed to minimize the run time of the search and to perform as many cutoffs as possible, thereby minimizing the cost of the search (total number of operations).

To achieve these goals a distinction is made among the sons of a node. The first son of a node is called the "left son". The subtree containing the left son is called the "left subtree" and the process that searches this subtree is the "left process". All other sons of a node are called "right sons" and are contained in "right subtrees" which are searched by "right processes".

The left subtree of a node is searched by a left process (which is spawned by the parent node) until a final value for the left son is backed up to the parent node. To obtain this final value, the left son's process spawns processes (lefts and rights) to search all of the left son's subtrees. Concurrently, a single, temporary value is obtained for each of the right sons of the parent node. These values are then compared to the final value of the left son and cutoffs are made where appropriate.

The temporary value for a right son is obtained by the right son's process spawning a process to search its left subtree. This new process searches the subtree, backs-up a value to the parent's right son, and then dies. If after a cutoff check the right subtree search continues, then a process is generated to search the second subtree of the right son. This procedure continues until either the subtree is exhaustively searched or the search is cut off.

It is clear that, by applying the above method, those nodes that must be examined by the alpha-beta algorithm will be visited first. This ensures that needless work is not done; a cutoff check is performed before processes are generated to search subtrees that may be cut off.

In a search with more processors than running processes it may be possible to minimize the runtime of the search by generating processes to search the sons of a right node concurrently using the idle processors. This brute force approach is not used since it conflicts with the other aim of our design, namely minimizing the cost of the search. The cost of any tree search consists mainly of the cost of updating the system in moving from parent to son and in the cost of evaluating or scoring a node. Therefore even though a processor (which could be doing concurrent work) is idle, the overall cost in operations is minimized by not searching subtrees which may not have to be searched.

There are seven main components of the parallel alpha-beta algorithm: Initialize, Handle, Score, Generate, GenerateMoves, Apply, and Update.

1) Initialize reads in the original board position (i.e., the configuration for the root node of the search tree) and the depth to which the tree will be searched. Handle is then invoked to create a process for the root.

2) Handle is a recursively-defined process. It searches a node in a game

CH1569-3/80/0000-0231$00.75 © 1980 IEEE

tree by calling either Score (for a leaf) or Generate (for a non-leaf) and then calling Update.

3) Score returns an integer representing the value of a given board configuration.

4) Generate searches a subtree that is not a leaf. It calls GenerateMoves to produce a list of moves from the current position. If the root of the subtree is a left node, then Handle is invoked once for each son. The processes thus created run concurrently, and Generate waits until they all terminate. If the root of the subtree to be searched is a right node, then the sons are searched in sequence by calling Handle for one of them, waiting for it to complete, and performing a cutoff check before searching the next son. Apply is used to produce board configurations for sons.

5) GenerateMoves produces all of the legal moves from a board configuration.

6) Apply produces the board configuration that results when a given move is made on a given board configuration.

7) Update waits until the parent's score table is free and then copies the value derived as a score for the current subtree into the table, if applicable.

Since we did not have a multiprocessor available on which to implement our algorithm, the simulation language GASP IV [5] was used to simulate physical parallelism. As our model of computation we use an MIMD computer. The machine we intend has a number of asynchronous processors with a communication facility provided by common memory or communication lines. A processor can initiate another processor, send a message to another processor, or wait for a message from another processor. Apart from these interactions, processors proceed independently.

The simulated environment provides multiple software processes and multiple hardware processors. A process is created for each node that is searched. The number of processors is a parameter of the program.

The implemented algorithm was experimented with to study the effects of parallelism on the cost of a tree search, this cost being expressed in: 1) run time of the tree search, 2) number of terminal nodes scored, and 3) total number of non-terminal and terminal nodes visited.

A uniform tree of a given depth and branch factor was generated and stored prior to the search. The terminal nodes of this tree were assigned scores chosen from a particular probability distribution. The principal continuation was sought and the three measures of cost recorded. Typical results of experiments are shown in Figs. 1 and 2.

The curves in Fig. 1 show that the run time decreases sharply with an increasing number of processors doing the search. As expected, the total number of nodes visited also increases with an increasing number of processors as can be seen in Fig. 2.

To analyse the storage requirements we first assume that an infinite number of processors is available to search the tree. During the first phase of the algorithm, knowledge about the behaviour of the sequential version is used to explore several paths concurrently and independently. During all the remaining phases several subtrees are searched in parallel, each subtree, however, being searched sequentially.

Fig. 3 shows a uniform tree whose depth and branch factor are both equal to three. The paths explored in parallel during the first phase are indicated by heavy lines. Nodes explored during the first phase are called "primary" nodes. Formally,

1) the root is a primary left son,
2) a primary left son at ply k is the left son of a primary left or right son at ply k - 1, and
3) a primary right son at ply k is a right son of a primary left son at ply k - 1.

Following the first phase the temporary score backed up at node l is compared with the ones at nodes i and j; if the former is smaller, then the subtrees of i and j need not be considered at all. Otherwise these two subtrees, shown circled in Fig. 4, are searched in parallel (each sequentially) during the second phase.

When these two subtrees have been fully searched the final score backed up at node l is compared with the temporary score at node m for a cutoff. If the former is larger, the cutoff check is successful and the unexplored subtrees of m need not be considered. Otherwise, more subtrees, shown circled in Fig. 5, are searched in parallel (each sequentially) during the third phase and so on.

At least one storage location is needed to hold the temporary score of each node being explored. When a node is discarded from further consideration its storage locations are reallocated to another unexplored node that the algorithm decides to examine. Therefore it is necessary to derive the maximum number of nodes simultaneously explored at any time during the search. This number is precisely the number of primary nodes.

To see this note that any tree searched sequentially during the subsequent phases is rooted at a node that was primary, that is to say explored during the first phase. This subtree is isomorphic to the leftmost subtree rooted at the same primary node. The leftmost subtree has at least as many primary nodes as a subtree searched in subsequent phases. Therefore the number of nodes searched in parallel during the second and later phases cannot exceed the number of primary nodes. Let

$l(k)$ = number of primary left sons at ply k, and

$r(k)$ = number of primary right sons at ply k.

In Fig. 3, $l(3)=5$ and $r(3)=6$. For a uniform tree:

$$l(k) = l(k-1) + r(k-1) \quad , k\geq 1$$
$$r(k) = l(k-1) * N \quad , k\geq 1$$
$$l(0) = 1 \text{ and } r(0) = 0 \quad ,$$

where N stands for the branch factor minus one. For a uniform tree of depth D, the total number of primary nodes is therefore given by

$$S = \sum_{k=0}^{D} l(k) + r(k)$$

and the storage requirements of the algorithm are of $O(S)$.

It is clear that our assumption about the availability of an unlimited number of processors can be relaxed. The maximum number of processors the algorithm will ever need to search a uniform tree of depth D is

$$P = l(D) + r(D) \ .$$

In Fig. 3, P=11.

Even though P establishes an upper bound it is still a very large number of order $N^{D/2}$, as one should have expected. In practice a small number of processors running in parallel is usually sufficient to achieve a substantial reduction in the running time of the sequential alpha-beta algorithm. In fact, it was observed that the run time cannot be decreased below a certain level no matter how many more processors are used in the search. The number of processors p* which first achieves this minimum run time was recognized as the "saturation point" of the algorithm.

These remarks lead us to reconsider our definition of primary nodes. The actual number of primary nodes is in fact determined by the number of processors available. If p processors are used to search a uniform tree of branch factor N + 1, then the actual number of primary nodes at level k is

$$\min \{l(k) + r(k), p\}$$

and the total number of primary nodes for a tree of depth D is

$$s(p) = \sum_{k=0}^{D} \min \{l(k) + r(k), p\} \quad .$$

Under these conditions the storage requirements of the algorithm are $O(s(p))$. Note that $S=s(P)$ and that for $p\leq N+1$, we have

$$s(p) = 1 + pD \quad .$$

Combining the experimental timing results with the analytical storage results and making a typical "time versus storage tradeoff" decision the optimum number of processors to be used can be determined. This is indicated by the graphs in Figs. 6 and 7. The curve in Fig. 6 is plotted empirically by varying the number of processors searching uniform trees of depth D and branch factor N + 1. The curve in Fig. 7 is obtained analytically using the expression for s(p). The two curves are used to determine p+, the optimum number of processors matching the available resources.

References

[1] S.G. Akl, D.T. Barnard, and R.J. Doran, Design, Analysis, and Implementation of a Parallel Alpha-Beta Algorithm, Department of Computing & Information Science, Queen's University, Kingston, Ontario, Canada, Technical Report 80-98, (April, 1980) 51 pp.

[2] G.M. Baudet, The design and analysis of algorithms for asynchronous multiprocessors, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Technical Report CMU-CS-78-116, (April, 1978) 182 pp.

[3] J.P. Fishburn, R.A. Finkel, and S.A. Lawless, Two papers on alpha-beta search, Department of Computer Sci., Univ. of Wisconsin-Madison, Technical Report 375, (Dec. 1979) 30 pp.

[4] N.J. Nilsson, Principles of Artificial Intelligence, Tioga, (1980) 476 pp.

[5] A.A.B. Pritsker, The GASP IV Simulation Language, John Wiley & Sons, (1974) 451 pp.

[6] G.C. Stockman, A minimax algorithm better than alpha-beta?, Artificial Intelligence, Elsevier, (1976) pp.179-196.

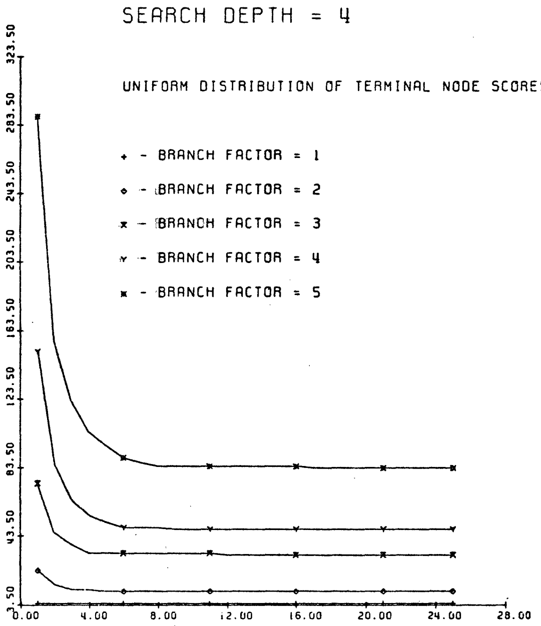


Fig.1

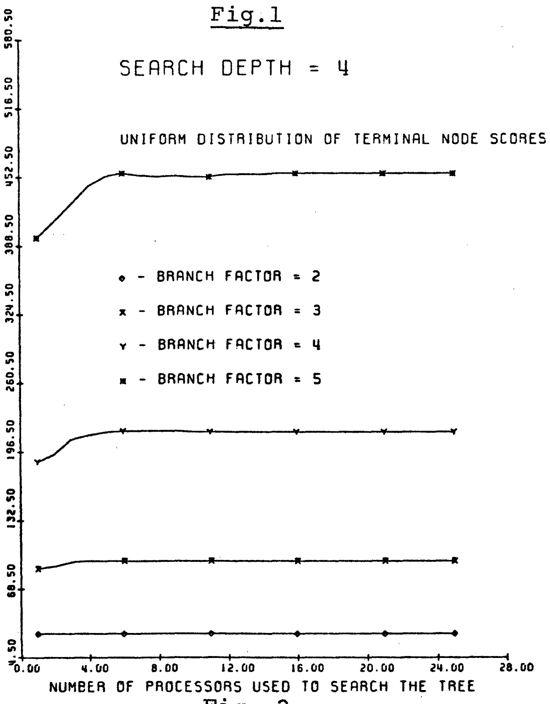


Fig. 2

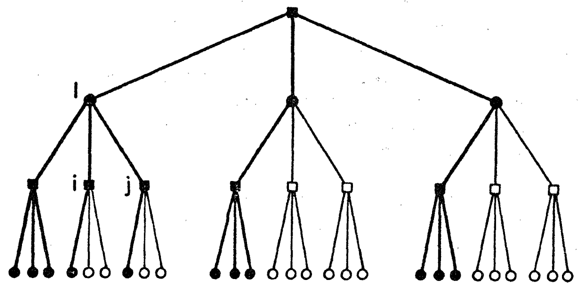


Fig. 3

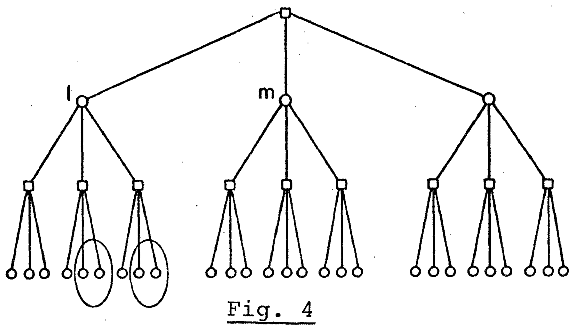


Fig. 4

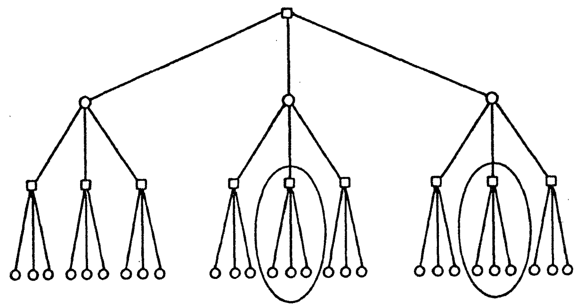


Fig. 5

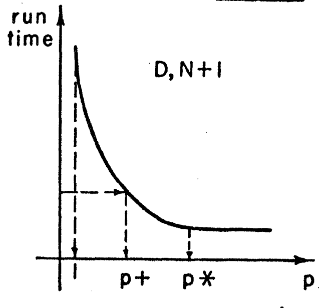


Fig.6

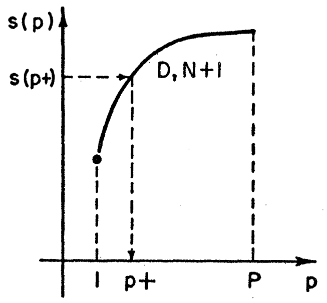


Fig. 7

PARALLEL ALPHA-BETA SEARCH ON ARACHNE

John P. Fishburn
Raphael A. Finkel
Sharon A. Lawless
Computer Sciences Department
University of Wisconsin-Madison
Madison, Wisconsin 53706

Abstract

We present a distributed algorithm for implementing α-β search on a tree of processors. Each processor is an independent computer with its own memory and is connected by communication lines to each of its nearest neighbors. Measurements of the algorithm's performance on the Arachne distributed operating system are presented. A theoretical model is developed that predicts speedup with arbitrarily many processors.

1. INTRODUCTION

The α-β search algorithm is central to most programs that play games like chess. It is now well-known [1] that an important component of the playing skill of such programs is the speed at which the search is conducted. For a given amount of computing time, a faster search allows the program to "see" farther into the future. In this paper we present and analyze a parallel adaptation of the α-β algorithm. This adaptation, which we will call the _tree-splitting algorithm_, speeds up the search of a large tree of potential continuations by dynamically assigning subtree searches for parallel execution.

In section 2, we summarize the α-β algorithm. Section 3 reviews a parallel implementation of the α-β algorithm suggested by Baudet [2]. Section 4 formally describes the tree-splitting algorithm. Section 5 presents performance measurements for this algorithm taken on a network of microprocessors. Section 6 discusses some possible optimizations and variations of the algorithm. Section 7 derives the obtainable speedup with k processors, as k tends towards ∞.

2. THE ALPHA-BETA ALGORITHM

Consider a board position from a game like chess or checkers. All possible sequences of moves from this position may be represented by a tree of positions called the _lookahead tree_. The nodes of the tree represent positions; the children of a node are moves from that node. The root node of the tree usually represents the current position. Since lookahead trees for most games are usually too large to be searched even by computer, they are usually truncated at a certain level. Since we will later be referring to a tree of processors, we reserve the following notation for nodes of lookahead trees: A node is often called a _position_. A node's child is its _successor_, and its parent is its _predecessor_. If each non-terminal node has n successors, we say that the tree has _degree n_. The _level_ of a node or subtree is its distance from the root.

The α-β algorithm is an optimization of the minimax algorithm, which we will review first. The two players are called _max_ and _min_; at the root node, it is max's turn to move. The minimax algorithm proceeds as follows: First, each leaf of the lookahead tree is assigned a _static value_ that reflects that position's desirability. (High values are desirable to max. In a game like chess, the main component of the value is usually the material balance between the two sides.)

The interior nodes of the lookahead tree may be given _minimax values_ recursively: If it is max's turn to move at node A, the value of A is the maximum of A's successors' values. (If the game were to proceed to node A, it would then be max's turn to move. Max, being rational, would choose the successor with the maximum value, say M. Therefore, the subtree rooted at A must have M as its value, because M is the value of the leaf node we would reach if the game reached A.) Similarly, if it is min's turn to move at a node, then the value of that node is the minimum of these values.

We will use a version of the minimax procedure called _negamax_: When it is max's turn to move at a terminal node, the node is assigned the same static value used in minimax. When it is min's turn to move, the static value assigned is the negative of what it would be in the minimax case. The value of a nonterminal node at any level is defined to be the maximum of the negatives of the values of its successors.

The negamax algorithm can be cast into an _ad hoc_ Pascal-like language. The following program is adapted from Knuth [3]:

```
function negamax(p:position):integer;
var m: integer;
  i,d : 1..MAXCHILD;
  succ : array[1..MAXCHILD] of position;
begin
  determine the successor positions
    succ[1],...,succ[d];
  if d = Ø then { terminal node }
    negamax := staticvalue(p)
  else
  begin { find maximum of child values }
    m := - ∞;
    for i := 1 to d do
      m := max(m,- negamax(succ[i]);
    negamax := m;
  end
end.
```

The α-β algorithm evaluates the lookahead tree without pursuing irrelevant branches. Suppose we are investigating the successors in a game of chess, and the first move we look at is a bishop move. After analyzing it, we decide that it will gain us a pawn. Next we consider a queen move. In considering our opponent's replies to the queen move, we discover one that can irrefutably capture the queen; she has moved to a dangerous spot. We need not investigate our opponent's remaining replies; in light of the worth of the bishop move, the queen move is already discredited.

The α-β search algorithm [3] formalizes this notion:

```
function alphabeta(p : position;
   α,β : integer) : integer;
label DONE;
var i,d : 1..MAXCHILD;
  succ : array[1..MAXCHILD] of position;
begin
  determine the successor positions
    succ[1],...,succ[d];
  if d = Ø then
    alphabeta := staticvalue(p)
  else
  begin
    for i := 1 to d do
    begin
      α := max(α, - alphabeta(succ[i],
          -β,-α));
      if α ≥ β then goto DONE { cutoff }
    end;
DONE: alphabeta := α
  end
end.
```

The function alphabeta obeys the _accuracy_ property: For a given position p, and for values of α and β such that $\alpha < \beta$,

if negamax(p) $\leq \alpha$,
then alphabeta(p,α,β) $\leq \alpha$

if negamax(p) $\geq \beta$,
then alphabeta(p,α,β) $\geq \beta$

if $\alpha <$ negamax(p) $< \beta$, then
alphabeta(p,α,β) = negamax(p)

The first and second cases above are called _failing low_ and _failing high_ respectively. In the third case, _success_, alphabeta accurately reports the negamax value of the tree. Success is assured if $\alpha = -\infty$ and $\beta = \infty$. The pair (α,β) is called the _window_ for the search.

To return to our example: When alphabeta is called with p representing the queen move, it is min's move. β is the cutoff value generated by the bishop move. The better the bishop move was for max, the lower is β. (Within the routine alphabeta, high values for α and β are good for the player whose move it is. A high value for α indicates that a good alternative for that player exists somewhere in the tree. A low value for β indicates that a good alternative exists for the other player somewhere else in the tree.) When the successor that captures the queen is evaluated, α becomes larger than β, and a cutoff occurs.

α-β pruning serves to reduce the _branching factor_, which is the ratio between the number of nodes searched in a tree of height N and one of height N-1, as N tends to ∞. Both theory [3], and practice [4] agree that with good move ordering (investigating best moves first), α-β pruning reduces the branching factor from the degree of the lookahead tree nearly to the square root of that degree. For a given amount of computing time, this reduction nearly doubles the depth of the lookahead tree.

When the algorithm is performed on a serial computer, the value of one successor can be used to save work in evaluating its siblings later on. Nevertheless, greater speed can be obtained by conducting α-β search in a parallel fashion. We define the _speedup_ of a parallel algorithm over a serial one to be the time required by the serial algorithm divided by the time for the parallel algorithm. We will restrict our attention to parallel computers built as a tree of serial computers. A node in this tree is a _processor_, a parent is a _master_, and a _child_ is a _slave_.

3. PARALLEL ASPIRATION SEARCH

In order to introduce parallelism, Baudet [2] rejects decomposition of the lookahead tree in favor of a parallel aspiration search, in which all slave processors search the entire lookahead tree, but with different initial α-β windows. These windows are disjoint, and in the simplest variant their union covers the range from $-\infty$ to $+\infty$. Since each window is considerably smaller than $(-\infty,+\infty)$, each processor can conduct its search more quickly. When the processor whose window contains the true minimax value of the tree finishes, it reports this value, and move selection is complete. Baudet analyzes several variants of this algorithm under the assumption of randomly distributed terminal values, and concludes that the obtainable speedup is limited by a constant independent of the number of processors available. This maximum is established to be approximately 5 or 6. Surprisingly, for k equal to 2 or 3, Baudet's method yields more than k-way speedup with k processors. Baudet infers that the serial α-β search algorithm is not optimal, and estimates that a 15 to 25 percent speedup may be gained by starting the search with a narrow window.

Since a narrow window does not speed up a successful search when moves are ordered best-first, Baudet's method yields no speedup under best-first move ordering.

4. THE TREE-SPLITTING ALGORITHM

Another natural way to implement the α-β algorithm on parallel processors divides the lookahead tree into its subtrees at the top level, and queues them for parallel assignment to a pool of slave processors. The master processor, as in the serial algorithm, maintains the variable α as the maximum of the negative of all subtree values. Each slave processor computes the value of its assigned subtree. The slave may use either serial α-β search or parallel α-β search if it has slaves of its own. When it finishes, it reports the value computed to its master. As the master receives responses from slaves, it narrows its window, and possibly tells working slaves about the improved window. When all subtrees have been evaluated, the master is able to compute the value of its position. A similar approach is discussed in [7].

4.1 The Slave Algorithm

The slave algorithm runs at terminal nodes of the processor tree. We will describe its interactions with its master by means of messages. The algorithm is equally easily expressed in a shared-memory or call-return form. The slave receives EVALUATE messages from its master, followed by any number of associated UPDATE messages that narrow its window. When an UPDATE message arrives, the slave adjusts its recursive values of α and β to what they would have been, had the search been started with the smaller window. When the slave has performed the search specified by the EVALUATE command, it sends a VALUE message back to its master, and then waits for another EVALUATE message.

The algorithm calls five functions:

Staticvalue(position)
: returns the static value of "position".

Send(message)
: sends the data in buffer "message" to process message.dest.

Receive(message)
: receives a message sent to this process, and places it in buffer "message".

Catch(kind,future message,catcher)
: arranges for all future messages with message.kind = "kind" to be immediately routed to buffer "message", bypassing any receive. Catch returns immediately, allowing the caller to proceed. Thereafter, when a message with the indicated kind arrives, the process is interrupted, and the routine "catcher" is called. When "catcher" returns, the process resumes. Slaves use catch to receive UPDATE messages without wasting time polling for them.

Alphabeta(p)
: was defined in section 2. The variables α and β are global arrays, not formal parameters, in order to facilitate updating their values in each recursive call of alphabeta when an UPDATE message arrives. The global variable "depth" represents the level of p.

```
The slave algorithm:
program slave();
label DONE;
var message,updatemessage :
    record
      pos : position;
      α,β,value : integer;
      kind : (EVALUATE,UPDATE,VALUE);
      dest : process;
    end;
  pos : position;
  α,β : array[1..MAXDEPTH] of integer;
  depth : 1..MAXDEPTH;
  tmp : integer;
  succ : array[1..MAXCHILD] of position;
  i,d : 1..MAXCHILD;
  mymaster : process;

procedure catcher;
{ called asynchronously by UPDATE }
  var scalα,scalβ,tmp : integer;
    k : 1..MAXDEPTH;
  begin
    scalα := updatemessage.α;
    scalβ := updatemessage.β;
    for k := 1 to MAXDEPTH do
    begin { update α,β arrays }
      α[k] := max(α[k],scalα);
      β[k] := min(β[k],scalβ);
      tmp := scalα;
      scalα := -scalβ;
      scalβ := -tmp;
    end
  end;
begin
  catch(UPDATE,updatemessage,catcher);
  while true do
  begin { 1 iteration per EVALUATE }
  receive(message); { receive EVALUATE }
    pos := message.pos;
    depth := 1;
    α[depth] := message.α;
    β[depth] := message.β;
    determine the children of pos
      succ[1],...,succ[d];
    if d = Ø then
      { evaluate terminal position }
      message.value := staticvalue(pos);
    else begin
      for i := 1 to d do
      begin { evaluate each successor }
        α[depth+1] := - β[depth];
        β[depth+1] := - α[depth];
        depth := depth+1;
        tmp := - alphabeta(succ[i]);
        depth := depth-1;
        if tmp > α[depth] then
          α[depth] := tmp;
        if α[depth] ≥ β[depth] then
        begin message.value := α[depth];
          goto DONE; { cutoff occurs }
        end
      end { for i := 1 to d do }
    end;
DONE: message.kind := VALUE;
    message.dest := mymaster;
    send(message);
  end { while TRUE do }
end. { program slave }
```

4.2 The Master Algorithm

The master algorithm runs on non-terminal nodes of the processor tree. It receives EVALUATE and UPDATE messages from its master and VALUE messages from its slave nodes. After an EVALUATE message is received, the master generates all successors of the position to be evaluated. Each slave is requested to EVALUATE one of these positions; the remaining positions are queued for service by slaves. Any UPDATE messages are relayed to active slaves.

The master may take various actions when it receives a VALUE message from a slave. First, if the VALUE message causes the current α value to increase, then -α is sent as an updated β value to all active slaves. Second, if α has been increased so that it becomes greater than or equal to β, then an α-β cutoff occurs. The nonpositive-width window is sent to all active slaves, quickly terminating them. Meanwhile, the master empties its queue of waiting successor positions. Third, if the queue of unevaluated successor positions is non-empty, the reporting slave is assigned the next position from the queue.

When all successors have been evaluated, the master sends a VALUE message to its master. In a game situation, the algorithm at the root node might serve as the user interface, and would remember which move has the maximum value.

Here is the master algorithm:

```
program master();
label INIT;
var message :
  record
    pos : position;
    α,β,value : integer;
    kind : (EVALUATE,UPDATE,VALUE);
    dest : process;
  end;
  pos : position;
  succ : array[1..MAXCHILD] of position;
  succstat : array[1..MAXCHILD] of
        (ASSIGNED,UNASSIGNED);
  i,d : 1..MAXCHILD;
  slave : array[1..MAXSLAVE] of process;
  slavestat : array[1..MAXSLAVE] of
        (BUSY,FREE);
  j : 1..MAXSLAVE;
  mymaster : process;
  α,β,tmp : integer;
begin
  while true do
  begin { 1 iteration per EVALUATE }
INIT: repeat { flush outdated UPDATEs }
        receive(message);
    until message.kind = EVALUATE;
    pos := message.pos;
    α := message.α;
    β := message.β;
    determine the successor positions
      succ[1],...,succ[d];
    if d = Ø then
    begin { terminal node }
      message.value := staticvalue(pos);
  message.kind := VALUE;
  message.dest := mymaster;
  send(message);
  goto INIT;
end;
for j:= 1 to MAXSLAVE do
  slavestat[j] := FREE;
for i := 1 to d do
  succstat[i] := UNASSIGNED;
while there exists a FREE slave j
  and an UNASSIGNED successor i do
begin { give initial assignments }
  message.pos := succ[i];
  message.$\alpha$ := $-\beta$;
  message.$\beta$ := $-\alpha$;
  message.kind := EVALUATE;
  message.dest := slave[j];
  send(message);
  slavestat[j] := BUSY;
  succstat[i] := ASSIGNED;
end;
while there exist BUSY slaves do
begin
  receive(message);
  if message.kind = UPDATE then
  begin { forward UPDATE message }
    if (message.$\alpha$ > $\alpha$) or
       (message.$\beta$ < $\beta$) then
    begin
      $\alpha$ := max($\alpha$,message.$\alpha$);
      $\beta$ := min($\beta$,message.$\beta$);
      message.$\alpha$ := $-\beta$;
      message.$\beta$ := $-\alpha$;
      message.kind := UPDATE;
      send(message) to all slaves;
    end
    if $\alpha \geq \beta$ then { cutoff }
      for i:=1 to d do
        succstat[i]
          := ASSIGNED;
  end
  else { message.kind = VALUE }
  begin
    j := answering slave;
    slavestat[j] := FREE;
    tmp := -message.value;
    if tmp > $\alpha$ then
    begin { send new $\alpha$-$\beta$ window }
      $\alpha$ := tmp;
      message.$\alpha$ := $-\beta$;
      message.$\beta$ := $-\alpha$;
      message.kind := UPDATE;
      send(message) to all slaves;
    end;
    if $\alpha \geq \beta$ then { cutoff }
      for i:=1 to d do
        succstat[i]
          := ASSIGNED;
    if there remains a successor,
      i, yet to be evaluated then
    begin { reassign slave }
      slavestat[j] := BUSY;
      succstat[i]
        := ASSIGNED;
      message.pos := succ[i];
      message.$\alpha$ := $-\beta$;
      message.$\beta$ := $-\alpha$;
      message.kind := EVALUATE;
      message.dest := slave[j];
      send(message);
    end
  end{ else message.kind = VALUE }
end; { while there are BUSY slaves }
message.value := $\alpha$;
message.kind := VALUE;
message.dest := mymaster;
send(message);
end{ while TRUE do }
end. { program master }
```

4.3 Alpha Raising

As an optimization of the master algorithm, the master running on the root node may send a special α-β window to a slave working on the last unevaluated successor. This window is $(-\alpha-1,-\alpha)$ instead of the usual $(-\beta,-\alpha)$. If that successor is not the best, then the slave's search will fail high as usual, but the minimal window speeds its search. If that successor is best, then the smaller window causes the search to fail low, again terminating faster. In either case, the root master determines which successor is the best move, even though its value may not be calculated. By speeding the search of the last successor, the idle time of the other slaves is reduced. (This narrow window given to the root's last subtree search can also be used in serial α-β search.)

We can generalize this technique in the following way, called alpha raising: Suppose that, among slaves evaluating successors of the root, slave$_1$'s current α value, α_1, is lower than any other, and that slave$_2$ has the second lowest α value, say α_2. Update α_1 to α_2-1, speeding up slave$_1$. If this update causes slave$_1$'s otherwise successful search to fail low, then the reported value is still lower than all others, and that move is still discovered to be best.

5. MEASUREMENTS OF THE ALGORITHM

Measurements of the performance of the tree-splitting algorithm have been taken on a network of LSI-11 microcomputers running under the Arachne operating system [5]*. The game of checkers was used to generate lookahead trees.

---

*We have been forced to change the name of the Roscoe distributed operating system, since Roscoe is a registered trademark of Applied Data Research, Incorporated. The new name we have chosen is Arachne; the operating system and research continue unchanged.

Static evaluation was based on the difference in a combination of material, central board position for kings and advancement for men. Moves were ordered best-first according to their static values. General α-raising was not employed, except for the special case for the last successor. A single LSI-11 machine searches lookahead trees at a rate of about 100 unpruned nodes per second. Inter-machine messages can be sent at a rate of about 70 per second.

Since only 5 processors are currently available in Arachne, it was not possible to test processor trees of depth greater than one directly. Instead, a depth-one processor tree was used to measure the speedup gained by replacing a terminal slave processor with a depth-one processor tree. When this slave is at level n, we call the measured speedup γ_n. γ_0 and γ_1 were measured. The procedure for measuring γ_1 made one simplifying assumption: Both a slave processor and a master processor below level zero can normally receive UPDATE messages from their masters. Due to the difficulty of duplicating the arrival times of these messages, they were not included in either the slave or the master-and-slaves case. (The master still gave its terminal slaves UPDATE messages.)

Ten board positions, $B_1, \ldots, B_{10}$, were chosen for use in these experiments. These positions actually arose during a human-machine game; they span the entire game. All lookahead trees from these positions were expanded to a depth of 8.

Two sets of experiments were performed. The two differed only in that the first set used one master and two slaves, while the second set used one master and three slaves. Within each experiment, γ_0 was measured directly for each B_i by evaluating the tree both serially and with the parallel algorithm running on a depth-one processor tree. Table 1 summarizes measurements of γ_0.

The ten board positions gave rise to 84 successors, so 84 EVALUATE commands were given to slaves while γ_0 was being measured. Times for both parallel and serial evaluation were measured for each command. The <u>aggregate speedup</u> for a group of commands is the total time required to execute them serially divided by the total time required to execute them in parallel. For each B_i, the aggregate speedup γ_1 for its subtree evaluations was computed. Table 2 summarizes measurements of γ_1.

Table 1: γ_0 for each B_i, i=1,...,10

	2 slaves	3 slaves
minimum	1.37	1.37
average	1.81	2.34
maximum	2.36	3.15
standard deviation	0.31	0.56

Table 2: γ_1 for each B_i, i=1,...,10

	2 slaves	3 slaves
minimum	1.03	1.38
average	1.46	1.96
maximum	1.77	2.60
standard deviation	0.22	0.38

Surprisingly, more than k-way speedup was occasionally achieved with k slaves: Three out of the ten B_i were sped up by more than 2 with 2 slaves, and two of those three were sped up by more than 3 with 3 slaves. Of the 84 subtrees of the B_is, 4 were sped up by more than 2 with 2 slaves, and 9 were sped up by more than 3 with 3 slaves; 2 of those achieved 6-way speedup. In each such case, subtree evaluations finished in a different order than they were assigned. While one large subtree was being evaluated by one slave, another smaller subtree was assigned and finished. The large subtree's evaluation then received an UPDATE message that sped it up or even terminated it. In fact, time-consuming searches are more likely than short ones to receive these messages. In particular, the search that receives the final $(-\alpha-1,-\alpha)$ window is likely to be larger than average.

6. <u>OPTIMIZATIONS</u>

Since the tree-splitting algorithm can be optimized in several ways, it should be considered the simplest variant of a family of tree-decomposing algorithms for α-β search. As a first optimization, since most of a master's time is spent waiting for messages, that time could be spent profitably doing subtree searches. However, only the deepest masters could hope to compete with their slaves in conducting searches. All other masters are by themselves slower than their slaves because their slaves have slaves below them to help. However, more than half of all masters control terminal slaves, and greater speedup should be achieved by running a slave algorithm along with these masters on the same processors. We might expect an additional 1.5-way speedup from this technique.

A second optimization groups several higher-level masters onto a single processor. For example, the 3 highest processors in a binary processor tree could be replaced by 3 processes running on a single processor.

Third, a master might evaluate a position by assigning that position's successor's successors to slaves, rather than that position's successors. Although this technique involves more message-passing, some advantage might result, because all of a master's slaves would work on finishing the position's first subtree before going on to the second. The evaluation of the second subtree would then receive the full benefit of the beta value generated by the first subtree. Furthermore, when slaves become idle as one subtree is finished, they can immediately be set to work on the next subtree.

Since most game-playing programs must make their move within a certain time limit, any speedup in tree search ability will generally be used to search a deeper lookahead tree. If we have an unlimited supply of processors to form into a binary tree, we can obtain an unlimited speedup only if the search is not limited in time. Otherwise we cannot, because we would eventually violate our premise that the lookahead tree is at least as deep as the processor tree. A new layer on the processor tree does not buy another full ply in the lookahead tree. For example, several speedups of 1.5 would be needed to search a 6-times larger chess lookahead tree, or about one additional ply. The depth of the processor tree would grow faster than the depth of the tree it searches and eventually would catch up. The only way to avoid this limit is to increase the fan-out of the processor tree. If the fan-out is high enough that no successor need ever be queued for evaluation by a slave, then the size of the maximum lookahead tree that can be evaluated within the time limit is limited only by the time required for EVALUATE commands to propagate from the root to the leaves. Long before this limitation is reached, we would run out of silicon for making the processors.

7. ANALYSIS OF SPEEDUP

We will now analyze the speedup that can be gained in searching large lookahead trees as the number of available processors grows without bound. For this purpose we introduce Palphabeta, a simplified version of the tree-splitting algorithm. This algorithm is less efficient than the version already discussed, but is more amenable to analysis. Much of the analysis in this section is a "parallelization" of results in [3]. Indeed, when $q = 0$ and $f = 1$, Theorem 1 and Corollary 1 reduce to results given by [3].

As before, the processors will be arranged in a uniform tree. Let $f \geq 1$ be the fan-out of the processor tree (uniform for all non-terminal nodes), and let $q \geq 1$ be its depth (uniform for all terminal nodes). Let $q + s$ be the depth of the lookahead tree, where $s \geq 1$. We assume that the lookahead tree has a uniform degree and that this degree, df, is a multiple of f, where d is ≥ 2.

The f function calls specified in the first line of the for-loop are intended to occur in parallel, activating functions existing on each of the f slaves. Unlike the tree-splitting algorithm, Palphabeta waits until all slaves finish before assigning additional tasks. Serial α-β search is activated on leaf slaves; Palphabeta is activated on all others. Here is the simpler parallel α-β algorithm.

```
function Palphabeta(p : position ;
   α, β : integer) : integer ;
var i : integer;
function g : integer;
begin
  determine the successors $p_1, \ldots ,p_{df}$.
  begin
    if depth($p_1$) < q then
      g := Palphabeta
    else  g := alphabeta;
    for i := 1 to d do
    begin
      $\alpha := \max(\alpha, \max_{(i-1)f<j\leq if} -g(p_j,-\beta,-\alpha))$;
      if $\alpha \geq \beta$ then go to DONE;
    end;
  DONE: Palphabeta:= α;
  end;
end;
```

7.1 Worst-first ordering

α-β search produces no cutoffs if, whenever the call alphabeta(p,α,β) is made, the following relation holds among the successors $p_1, \ldots ,p_d$:

$$\alpha < -\text{negamax}(p_1) < \ldots < -\text{negamax}(p_d) < \beta.$$

We call this ordering worst first. If no cutoffs occur, it is easy to calculate the time necessary for Palphabeta to finish. Assume that a processor can generate f successors, send messages to all of its f slaves and receive replies in time ρ. (This figure counts message overhead time but does not include computation time at the slaves.) Assume also that the serial α-β algorithm takes time n to search a lookahead tree with n terminal positions. Let a_n be the time necessary for a processor at distance n from the leaves to evaluate its assigned position. A leaf processor executes the serial algorithm to depth s. Thus we have $a_\emptyset = (df)^s$. An interior processor gives d batches of assignments to its slaves, and each batch takes time ρ plus the time for the slave processor to complete its calculation. Thus we have $a_{n+1} = d(\rho+a_n)$. The solution to this recurrence relation is

$$a_q = \rho\left(\frac{d^{q+1} - d}{d - 1}\right) + d^{q+s}f^s,$$

which is the total time for Palphabeta to complete. Since the time for the serial algorithm to examine the same tree is $(df)^{q+s}$, the speedup for large s is f^q. There are $(f^{q+1}-1)/(f-1)$ processors, roughly f^q, so when no pruning occurs, the parallel algorithm yields speedup that is roughly equal to the number of processors used.

7.2 Best-first ordering

We will now investigate what happens when the lookahead tree is ordered best-first. We omit the proofs of Theorems 1 and 2 in the interests of conciseness. Full details may be found in [6].

Definition: We will use the Dewey decimal system to name nodes in both processor trees and lookahead trees. The root is named by the null string. The j successors of a node whose name is $a_1 \ldots a_k$ are named by $a_1 \ldots a_k 1$ through $a_1 \ldots a_k j$.

Definition: We say that the successors of a position $a_1 \ldots a_n$ are in best-first order if

$$\text{negamax}(a_1 \ldots a_n) = -\text{negamax}(a_1 \ldots a_n 1).$$

Definition: We say a position $a_1 \ldots a_n$ in the lookahead tree is (q,f)-critical if a_i is (q,f)-restricted for all even values of i or for all odd values of i. An entry a_i is (q,f)-restricted if

$$1 \leq i \leq q \quad\text{and}\quad 1 \leq a_i \leq f$$

$$\text{or if } q < i \quad\text{and}\quad a_i = 1.$$

Theorem 1: Consider a lookahead tree for which the value of the root position is not $\pm\infty$ and for which the successors of every position are in best-first order. The parallel α-β procedure Palphabeta examines exactly the (q,f)-critical positions of this lookahead tree.

Corollary 1: If every position on levels $\emptyset,1,\ldots,q+s-1$ of a lookahead tree of depth q+s satisfying the conditions of Theorem 1 has exactly df successors, for d some fixed constant, and for f the constant appearing in Palphabeta, then the parallel procedure Palphabeta (along with alphabeta, which it calls), running on a processor tree of fan-out f and height q, examines exactly

$$f^{\lfloor q/2 \rfloor}(df)^{\lceil (q+s)/2 \rceil} +$$

$$f^{\lceil q/2 \rceil}(df)^{\lfloor (q+s)/2 \rfloor} -$$

f^q terminal positions.

Proof: There are $f^{\lfloor q/2 \rfloor}(df)^{\lceil (q+s)/2 \rceil}$ sequences $a_1 \ldots a_{q+s}$, with $1 \leq a_i \leq df$ for all i, such that a_i is (q,f)-restricted for all even values of i; there are $f^{\lceil q/2 \rceil}(df)^{\lfloor (q+s)/2 \rfloor}$ such sequences with a_i (q,f)-restricted for all odd values of i; and we subtract f^q for the sequences $\{1,\ldots,f\}^q 1^s$, that we counted twice.

Q.E.D.

Theorem 2: Under the conditions of Corollary 1, and assuming also that (1) serial α-β search is performed in time equal to the number of leaves visited, and (2) in ρ units of time, a processor can generate f successors of a position, send a message to each of its f slaves, and receive the f replies, then the total time for Palphabeta to complete is

$$(df)^{\lfloor s/2 \rfloor} + (df)^{\lceil s/2 \rceil} - 1 +$$
$$h(q)[d(3\rho+(df)^{\lfloor s/2 \rfloor}+(df)^{\lceil s/2 \rceil})+\rho$$
$$-(df)^{\lfloor s/2 \rfloor}-(df)^{\lceil s/2 \rceil}] - \rho q,$$
if q is even;

$$(df)^{\lfloor s/2 \rfloor} + (df)^{\lceil s/2 \rceil} - 1 +$$
$$h(q-1)[d(3\rho+(df)^{\lfloor s/2 \rfloor}+(df)^{\lceil s/2 \rceil})+\rho$$
$$-(df)^{\lfloor s/2 \rfloor}-(df)^{\lceil s/2 \rceil}] - \rho q$$
$$+d^{(q-1)/2}[d(\rho+(df)^{\lfloor s/2 \rfloor})+\rho-(df)^{\lfloor s/2 \rfloor}],$$
if q is odd;

where the function h is defined by

$$h(q) = (d^{q/2} - 1)/(d - 1).$$

Under conditions of best-first search, the parallel α-β algorithm gives $O(\sqrt{k}\,)$ speedup with k processors for searching large lookahead trees. Theorem 3 formalizes this result:

Theorem 3: Suppose that Palphabeta runs on a processor tree of depth $q \geq 1$ and fan-out $f > 1$. Suppose that the lookahead tree to be searched is arranged in best-first order and is of degree df and depth q+s, where $d \geq 1$. Denote by R the time for alphabeta to search this tree, and by P the time for Palphabeta to search the tree. Then

$$\lim_{s \to \infty} R/P = f^{q/2}.$$

Proof: The time for the serial algorithm is

$$(df)^{\lfloor (s+q)/2 \rfloor} + (df)^{\lceil (s+q)/2 \rceil} - 1,$$

from Corollary 1. If we divide this quantity by the expression given by Theorem 2 for P, and take the limit as s goes to ∞, we obtain the desired result.

Q.E.D.

7.3 Discussion

The measurements presented in section 5 fall within the range bounded by the theoretically-predicted best-first and worst-first speedups. If we take $\gamma_0\gamma_1$ to be the speedup that would be given by a processor tree of depth two, then the measured speedup for two, three, four, and nine terminal processors is 1.81, 2.34, 2.64, and 4.59 respectively. Theory predicts speedup equal to the number of terminal processors for worst-first ordering. Best-first speedup is predicted to be the square root of the number of terminal processors, or 1.41, 1.73, 2, and 3 respectively.

8. ACKNOWLEDGMENTS

The authors gratefully acknowledge the help and ideas offered by Karl Anderson, Will Leland, Marvin Solomon, and Larry Travis.

9. REFERENCES

[1] Berliner, H.J., "A Chronology of Computer Chess and its Literature," Artificial Intelligence, Vol. 10, 1978, (April, 1978), pp. 201-214.

[2] Baudet, G.M., The Design and Analysis of Algorithms for Asynchronous Multiprocessors, Department of Computer Science, Carnegie-Mellon University Technical Report, (April, 1978), 182 pp.

[3] Knuth, D.E., and Moore, R.W., "An Analysis of Alpha-Beta Pruning," Artificial Intelligence, Vol. 6, No. 4, (Winter, 1975), pp. 293-326.

[4] Samuel, A.L., "Some Studies in Machine Learning Using the Game of Checkers, II - Recent Progress," IBM Journal of Research and Development, (November, 1967), pp. 601-617.

[5] Solomon, M. H., Finkel, R. A., "The Roscoe Distributed Operating System," Seventh ACM Symposium on Operating Systems Principles, (Dec. 1979).

[6] Fishburn, J.P., Finkel, R.A., and Lawless, S.A., Two Papers on Alpha-Beta Pruning, (Revised) Department of Computer Science, University of Wisconsin-Madison Technical Report, (June 1980), 33 pp.

[7] Akl, S.G., Barnard, D.T., and Doran, R.J., Searching Game Trees in Parallel, Department of Computing and Information Science, Queen's University Technical Report, (Nov. 1979), 36 pp.

TWO PARALLEL ALGORITHMS FOR SHORTEST PATH PROBLEMS

Narsingh Deo
C. Y. Pang
R. E. Lord

Computer Science Department
Washington State University
Pullman, Washington 99164

ABSTRACT

After examining several dozen serial algorithms and their variations for various shortest-path problems, two algorithms were selected as good candidates for parallelization on an MIMD-type processor. These are: (1) Pape-D'Esopo version of the Moore's algorithm for finding shortest paths from one node to all others, and (2) Warshall-Floyd algorithm for finding shortest paths between all pairs of nodes. The techniques used in designing the two parallel algorithms are fundamentally different--one involves parallel processing with a queue and is suited for sparse networks while the other employs matrix methods and is suited for dense networks. The correctness of these algorithms is proved. Execution times are analyzed and compared with actual execution times on the HEP computer (an MIMD machine).

1. INTRODUCTION

Shortest-path problems are by far the most fundamental and also the most commonly encountered problems in the study of transportation and communication networks. Often the repeated determination of shortest paths and distances form the core (inner loop) in many transporation planning and utilization packages. Therefore, the search for faster and faster shortest-path procedures continues. After reviewing over 200 papers on shortest-path algorithms and after classifying and analyzing several dozen existing algorithms [5], two points became evident to us (among other things): (1) the shortest-path problems have almost reached their theoretical bounds of speed if conventional serial computers are to be used; and (2) certain algorithms (which may be most suited for serial mode) cannot be "parallelized" as readily as others. For example, Dijkstra's algorithm [4, 7, 18] for finding a shortest path between two nodes is not as well suited for parallelization as the Bellman-Moore [5, 14, 21] algorithm is.

We have selected two algorithms (for solving two different shortest-path problems), which appear to us as the best candidates for parallelization, for a detailed presentation in this paper. These are: (1) Pape-D'Esopo version of the Moore's algorithm for finding shortest paths from one node to all others [14, 15] and (2) Warshall-Floyd [4, 10, 18] algorithm for finding shortest paths between all pairs of nodes. The techniques used in designing the two parallel algorithms are fundamentally different--one involves parallel processing with a queue and is suited for sparse networks while the other employs matrix methods and is suited for dense networks.

We designed parallel versions of these two algorithms, suited for an MIMD (multiple instruction multiple data stream) [11] machine--keeping an eye, in fact, on the characteristics of the specific MIMD machine on which the designed parallel programs were actually to be executed. For example, on this machine the time required in creating a process is greater than the time needed to lock or unlock a resource.

In recent years, MIMD machines are not only being built experimentally in university laboratories, but they are being built in private industries. The Heterogeneous Element Processor (HEP) of DENELCOR Inc. [20], and the SMS 201 of Siemens AG [12] are two examples of commercial MIMD machines. Since the HEP was available to us, we coded and executed our programs on the HEP and performed the timing study on it.

Although a number of theoretical studies have been reported on parallel processing of graphs [1, 8, 9, 13, 17, 19], very few of them have considered the specific problems of shortest path problems and none have actually designed, coded and executed a parallel shortest-path algorithm on a real parallel computer (particularly on an MIMD computer) to the best of our knowledge. This study considers many of the real nuts-and-bolts issues of parallelization of existing algorithms, data structures, efficiencies and speed-gains over the serial implementations.

In Section 2, we will give definitions relevant to shortest paths on a network. In Section 3, we design a parallel algorithm for finding sortest paths from one specified node to all other nodes in a given network. The proof of correctness of the algorithm and the details of our model of computation are also given in Section 3. In Section 4, we present the second algorithm--for finding shortest paths between all pairs of nodes in a given network. The proof of its correctness and some empirical results on execution time are also presented in Section 4.

2. SOME DEFINITIONS

The following are the definitions of some of the important graph-theoretic terms used in this paper. Definitions for the rest of the terms can be found in

This work was supported by U.S. Department of Transportation contract no. DOT-RC-92042 and by NSF grant no. MCS78-25851.

CH1569-3/80/0000-0244$00.75 © 1980 IEEE

any textbook on graph algorithms or networks [4, 18]. A directed graph G = (V, E) is an ordered pair of finite sets: V of nodes, and E of arcs. We will use NODES to denote the number of nodes in V. We will also use {1, 2, . . . , NODES} to denote the elements of V. And arc a in E is an ordered pair, (u, v), of nodes. An arc a = (u, v) is said to start at u and end at v. A network is a directed graph, G, together with a real valued function, ℓ, on the set of arcs. For any arc a, $\ell(a)$ is the arc length of a. An arc length matrix has its $(u, v)^{th}$ entry as $\ell(u, v)$ if the arc (u, v) exists. The entry is ∞ if (u, v) does not exist. A path P is a finite sequence of arcs $P = (a_1, a_2, \ldots, a_k)$, such that a_i starts where a_{i-1} ends, for i = 2, . . . , k. The length d(P) of a path P is defined to be $d(P) = \ell(a_1 + \ldots + \ell(a_k)$. If $a_i = (u_{i-1}, u_i)$, we will, in addition, use $(u_0, u_1, \ldots, u_k)$ to denote P, and P is called a path from u_0 to u_k. A path that starts and ends at the same node is called a cycle. A cycle with negative path length is called a negative cycle. P is a shortest path from u to v if d(P) is minimum over the length of all paths from u to v; the shortest distance from u to v is then d(P). The one-to-all shortest path problem is the problem of finding the shortest paths from a given node, called the source, to all the other nodes, the destinations. The all-to-all shortest path problem is the problem of finding a shortest path for every pair of nodes in the network.

3. A PARALLEL ALGORITHM FOR THE ONE-TO-ALL SHORTEST-PATH PROBLEM

A modification of Moore's algorithm [14] by D'Esopo as reported in [16] was further developed by Pape [15] into two very efficient codes for finding shortest paths from a specified source node to all other nodes in the given network. This Pape-D'Esopo-Moore algorithm, which we will refer to as PDM algorithm, may be described in an Algol-like language as follows:

Algorithm PDM

```
 1 for all u ≠ SOURCE do
 2    D[u]  := ∞;
 3 D[SOURCE]  := 0;
 4 initialize Q to contain SOURCE only;
 5 while Q is not empty do
 6 begin
 7    delete Q's head node u;
 8    for each arc (u, v) that starts at u do
 9      if D[v] > D[u] + ℓ(u, v) then
10      begin
11        P[v]  := u;
12        D[v]  := D[u] + ℓ(u, v);
13        if v was never in Q then
14           insert v at the tail of Q;
15        if v was in Q, but not currently then
16           insert v at the head of Q
17      end
18 end
```

During the execution of Algorithm PDM, the label D[u] is always updated to be the currently known shortest distance from SOURCE to u, and P[u] is always updated to be the predecessor node of u on the currently known shortest path from SOURCE to u. Since each insertion of a node u into Q is preceded by a decrement of D[u], this algorithm is guaranteed to terminate provided the input network has no negative cycles.

To see that the D[u]'s do indeed converge to the shortest distances, we first note that at termination $D[v] \le D[u] + \ell(u, v)$ holds for every arc (u, v). Suppose the node sequence $(SOURCE \equiv u_0, u_1, \ldots, u_k \equiv u)$ is a path from SOURCE to u, then its path length is given by

$$\begin{aligned}&\ell(u_0, u_1) + \ldots + \ell(u_{k-1}, u_k)\\ &\ge (-D[u_0] + D[u_1]) + \ldots + (-D[u_{k-1}] + D[u_k])\\ &= -D[SOURCE] + D[u] = D[u].\end{aligned}$$

Thus, D[u] is the shortest distance from SOURCE to u, and the node sequence,

SOURCE ≡ P[... P[u] ...], ... , P[P[u]],P[u],u

is the shortest path from SOURCE to use as obtained by Algorithm PDM.

The experiments of Denardo and Fox [2], Dial, Glover, Karney and Klingman [3], Pape [8], and Vliet [11] show that on the average Algorithm PDM is faster than almost every other shortest-path algorithm, if the input network has a low arcs to nodes ratio. We will, therefore, base our parallel algorithm on Algorithm PDM.

Let us fix our model of parallel computation before developing parallel algorithms. We will assume that our computer can simultaneously execute up to K processes. The communication between the processes is done via a common memory. The computer supports the operations: create, lock, and unlock [pp. 77-78 of Ref. 2]. When a process P_1 executes the statement "create process P_2," P_2 will start execution and P_1 will continue. For a memory X, after process P_1 executes "lock X," any other process that attempts to read, write, or lock X will have to wait until P_1 executes an "unlock X." Our model of computation is a realistic one; for the HEP computer can simultaneously execute processes, it has a common memory for all the processes, and it supports the operations create, lock, and unlock efficiently.

For practical reasons, we will assume that create, lock, and unlock take non-zero units of time to execute. In designing our algorithm, we also assume that create requires a longer execution time than lock and unlock. This assumption is also realistic, because create in the HEP machine using the FORTRAN language is implemented with four instructions, whereas only one machine instruction is required for implementing lock or unlock.

An obvious way to utilize the concurrent processing in Algorithm PDM would be to execute the inner for loop (statements 8 to 17) simultaneously. But this approach is unprofitable because the overhead for a create is high compared to the execution of one pass of the loop. Moreover, in this approach the maximum number of concurrent processes utilized would be about four, if the input is a typical road network (with outdegree = 4). Therefore, we will avoid breaking the inner for loop into different processes; instead we will distribute the passes of the while loop (statements 5 to 18) to different processes. This will avoid excessive use of create's.

We will use only K-1 create's to obtain a total of K concurrent processes at the beginning of the algorithm, and use lock's and unlock's to take care of the rest of the synchronization. During the execution of the algorithm, the K processes--one called MASTER and the others called WORKERs--share the computation load, as long as there are known tasks to be performed. Each process takes approximately 1/K of the work load in the initialization step. In the path-finding step, each process repeatedly deletes a node, u, from Q, and updates P[v]'s and D[v]'s for the successors, v's, of u. In addition to a WORKER's tasks, the MASTER is responsible for finishing the initialization step and for synchronizing the initiation and termination of the path-finding step. Our parallel algorithm, which we will refer to as PPDM, is as follows:

Algorithm PPDM (Parallel Pape-D'Esopo-Moore)

Process MASTER

```
 1      MSYN  := "yes"; WAIT := 0; DONE  := 0;
 2      for i  := 2 step 1 until K do
 3        Create process WORKER(i);
 4      for u  := 1 step K until NODES do
 5        D[u]  := ∞;
 6 L1:  if WAIT < K - 1 then goto L1;
 7      D[SOURCE]  := 0;
 8      initialize Q to contain SOURCE only;
 9 L2:  lock Q;
10      if Q is empty then goto L3;
11      delete Q's head node u;
12      unlock Q;
13      MSYN  := "no";
14      reach successor nodes of u (Block B);
15      MSYN  := "yes";
16      goto L2;
17 L3:  if WAIT = K - 1 then goto L4;
18      unlock Q;
19      goto L2;
20 L4:  DONE  := 1;
21      unlock Q;
22 L5:  if DONE < K then goto L5
```

Process WORKER(i)

```
 1      for u  := i step K until NODES do
 2        D[u]  := ∞;
 3 L1:  if MSYN  := "yes" then goto L3;
 4      lock Q;
 5      If Q in empty then goto L2;
 6      delete Q's head node u;
 7      unlock Q;
 8      reach successor nodes of u (Block B);
 9      goto L1;
10 L2:  unlock Q;
11      goto L1;
12 L3:  lock WAIT; WAIT:=WAIT+1; unlock WAIT;
13 L4:  if DONE > 0 then goto L5;
14      if MSYN = "yes" then goto L4:
15      lock WAIT; WAIT:=WAIT-1; unlock WAIT;
16      goto L1;
17 L5:  lock DONE; DONE:=DONE+1; unlock DONE
```

Block B

```
 1      for each arc (u, v) that starts at u do
 2      begin
 3        newdv  := D[u] + ℓ(u, v);
 4        lock D[v];
 5        if D[v] ≤ newdv then
 6          unlock D[v]
 7        else begin
 8          P[v]  := u;
 9          D[v]  := newdv;
10          unlock D[v];
11          lock Q;
12          if v was never in Q then
13            insert v at the tail of Q;
14          if v was in Q, but is not currently then
15            insert v at the head of Q;
16          unlock Q
17        end
18      end
```

Note: For Block B of the MASTER process, statement 11 should be changed to:

```
11      MSYN  := "yes"; lock Q; MYSN  := "no";
```

In Algorithm PPDM, the local variables are written in lower case letters, they are i, u, v, and newdv. The variables MSYN, WAIT, and DONE are the communication links between the MASTER and the WORKERs. MSYN = "yes" signals the WORKERs to let the MASTER check the Q first. WAIT is the number of WORKERs waiting for further command from the MASTER (i.e. WAIT is the number of WORKER processes which are executing statements 13 and 14). DONE is used by the MASTER to broadcast the termination signal. This algorithm requires the processes to keep on processing Block B until Q is empty. Block B is equivalent to statements 8 to 17 of Algorithm PDM. The locking and unlocking of D[v] and Q are added in Block B to ensure that Algorithm PPDM computes correctly.

Proof of correctness

We will now informally prove the correctness of this algorithm. It is easy to see that the initialization step is correct. For the path-finding step, we will first state and prove six remarks to show that the algorithm terminates for all networks which have no negative cycles.

Remark 1: For any node v, D[v] is nonincreasing with time.

Remark 2: Each finite D[v] represents the length of a path from SOURCE to v.

Remark 3: Only a finite number of insertions are made into Q.

Remark 4: Every execution of Block B always terminates.

Remark 5: There exists a time, t_1, such that the MASTER process will not execute Block B and MSYN = "yes" for all time after t_1.

Remark 6: Algorithm PPDM terminates.

To see that D[v] is nonincreasing, one simply observes that D[v] only changes when it is locked, and the changes are always decrements. To see that each finite entry D[v] represents a path length, we use induction on the time sequence of the change on the array D[•]. Let t_1 be the time immediately after D[SOURCE] is initialized to zero, and let t_{i+1} be the time immediately after the first change (or changes) in D[•] after t_i, for i = 1, 2, . . . At time t_1, D[SOURCE] = 0 is the only entry of D[•] with a finite value, and 0 is the path length of the null path from SOURCE to SOURCE. Suppose for all time $t \leq t_i$, each finite D[v] represents a path length from source to v, and suppose D[v] is changed immediately before t_{i+1}. Assume that the change in D[v] is caused as we fan out from u, and that the value of D[u], at the time of its reading statement 3 in Block B, is the path length of (SOURCE $\equiv u_0$, u_1, . . . , $u_j \equiv u$). At time t_{i+1}, D[v] is the path length of (u_0, u_1, . . . , u_j, v). Thus, Remark 2 follows by induction.

To see that Remark 3 holds, we first notice that each D[v] is bounded from below, because the D[v]'s represent path lengths and the input network has no negative cycles. Secondly, we notice that there are only finitely many decrements to the D[v]'s, because each decrement decreases a D[v] by at least the minimum length difference between two loopless paths. Thus Remark 3 follows, since each insertion into Q implies a previous decrement of a D[v].

We will prove Remarks 4 and 5 together. To prove Remark 4, it suffices to show that no indefinite waits occur at Block B's statements 3, 4, and 11. By Remark 3, we see that Block B can be executed for only finitely many times. Thus every waiting at statements 3 and 4 takes a finite time. Because Q can be locked outside Block B, more arguments are needed to show that no indefinite wait occurs at Block B's "lock Q" statement (statement 11). We will prove a stronger result that no indefinite wait can occur at any "lock Q" statement in Algorithm PPDM. The MASTER always sets MSYN to "yes" before it executes "lock Q", and when MSYN is "yes" all WORKERs will be blocked from entering statements 4 to 11 and Block B. Thus the MASTER has no indefinite wait at "lock Q", and that its executions of Block B take finite time. Before we prove similar results for the WORKERs, we first prove Remark 5. It is easy to see that the loop of the MASTER's statements 9 to 16 has no indefinite wait. We claim that the loop of statements 9, 10, 17, 18, and 19 has no indefinite wait also, for if the MASTER is waiting at statement 17, then MSYN would have the value "yes", and consequently, only finitely many short lockings of WAIT can occur at the WORKERs' statement 12. Since indefinite wait does not occur at the MASTER process, and there are only finitely many insertions into Q, we conclude that eventually the MASTER will never enter Block B. We have just proved Remark 5. To finish the proof of Remark 4, we assert that the WORKERs have finite waiting time for executing the "lock Q" statements. Suppose the converse is true, and j WORKERs are waiting indefinitely at the "lock Q" statements (i.e. WORKER's statement 4 or Block B's statement 11). By Remark 5, the MASTER will eventually be looping at statements 9, 10, 17, 18, and 19. Each time the MASTER executes "unlock Q", statement 18, one of the j waiting WORKERs is allowed to finish executing "lock Q", which is a contradiction.

To prove Remark 6, we first recall that every execution of "lock Q" takes a finite waiting time. From Remark 3, we see that Q will eventually be empty and WORKER will not execute statements 6 to 9. By Remark 5, MSYN eventually has the value "yes", therefore all WORKERs are directed to the loop of statements 14 and 15. Consequently, Algorithm PPDM terminates.

Now we prove the correctness of the outputs, D[•], and P[•]. We use $D_t[u]$ and $P_t[u]$ to denote the values of D[u] and P[u] at time t, and use z to denote the termination time. We first claim that $D_z[v] \leq D_z[u] + \ell(u, v)$, for each arc (u, v). Suppose (u_1, v_1) is an arc of the input network. Let a be the time of the last deletion of u_1 from Q. Consequently, Block B is executed for u_1 after time a. The processing of the arc (u_1, v_1) includes the execution of either statements 5 and 6, or statements 5, 8, 9, and 10. Let b be the time of the execution of "unlock $D[v_1]$", at statement 6 or 10. Since the last deletion of u_1 occurs at a, it is easy to see that $D[u_1]$ stays constant after time a. Consequently, $D_z[v_1] \leq D_b[v_1] \leq D_z[u_1] + \ell(u_1,v_1)$. Having proved $D[v] \leq D[u] + \ell(u, v)$ for all arcs (u, v), we conclude that the D[u]'s are the shortest distances by the same argument that was used for the proof of correctness of Algorithm PDM.

To prove that for each u,

$$(\text{SOURCE} \equiv P_z[. . . P_z[u] . . .], . . . , P_z[u], u)$$

is a shortest path, it suffices to show that for each v_1, if $u_1 = P_z[v_1]$ then $D_z[v_1] = D_z[d_1] + \ell(u_1, v_1)$, for it says that a shortest path from SOURCE to u_1 concatenated with (u_1, v_1) forms a shortest path from SOURCE to v_1. Let time a and time b be defined as before. It is easy to see that $D[v_1]$ is decreased in that execution of Block B, and so $D_b[v_1] = D_z[u_1] + \ell(u_1, v_1)$. Finally, we see that

$D_z[v_1] = D_b[v_1]$, because any change of $D[v_1]$ after time b implies a change in $P_b[v_1] = u_1$. This completes the proof of correctness of Algorithm PPDM.

Algorithm PDM and Algorithm PPDM were coded to run on the HEP computer. The programs use linked queue, which is used in Pape [15], and Dial, Glover, Karner, and Klingman [6]. The input network is stored in a linked list structure called the forward star form, used also in [6]. Timing experiments were performed with randomly generated connected networks. Following the characteristics of the Eastern Washington Highway Network, the generated networks were assigned exponentially distributed arc lengths and have approximately 35% of nodes outdegree of one, 9% of nodes outdegree of two, 40% of nodes outdegree of three, and 16% of nodes an outdegree of four. Highway networks usually have all two-way roads, and so do generated networks. For each NODES = 10, 25, 50, 75, 100, we generated two networks. For each network, we picked five source nodes. Each of these 100 problems are solved with the sequential Algorithm PDM, and the parallel version, Algorithm PPDM, with the number of processors K = 1 to 8. Let T_S denote the solution time for the sequential algorithm, and T_K denote the solution time with the K-processor, parallel algorithm. For each problem, the speed-up $S_K = T_S/T_K$, and the efficiencies, $E_K = S_K/K$, are computed. For fixed NODES and K, the averages of S_K's and E_K's are plotted in Figure 1 and Figure 2, respectively. For NODES = 75 and 100, we see that a speed-up of approximately three is achieved with five processors, and thus an approximate efficiency of 60%. However, regardless of the number of processors used, we expect that Algorithm PPDM has a constant upper bound on its speed-up, because every process demands private use of the Q.

4. A PARALLEL ALGORITHM FOR THE ALL-TO-ALL SHORTEST PATH PROBLEM

The best known algorithm for determining shortest paths between all pairs of nodes is due to Floyd [10], which in turn is based on an earlier algorithm for transitive closure proposed by Warshall [4].

The basic idea of the algorithm may be expressed as follows:

Algorithm F

```
1  for k := 1 step 1 until NODES do
2    for i := 1 step 1 until NODES do
3      for j := 1 step 1 until NODES do
4        if D[i, j] > D[i, k] + D[k, j] then
5          D[i, j] := D[i, k] + D[k, j]
```

The matrix, D[•], is initialized to be the arc length matrix. If the input network contains no negative cycle element D[i, j] at the termination is the shortest distance from u to v; because at the end of the kth iteration, D[i, j] is updated to be the shortest distance from i to j via paths that have intermediate nodes which are contained in {1, 2, . . . , k}. We will show that the inner loops of Floyd's algorithm may be computed in parallel as follows:

Algorithm PF (Parallel Floyd)

```
1  for k := 1 step 1 until NODES do
2    for 1 ≤ i, j ≤ NODES do simultaneously
3      if D[i, j] > D[i, k] + D[k, j] then
4        D[i, j] := D[i, k] + D[k, j]
```

To prove that Algorithm PF is correct, we use the theory developed for controlling concurrent processes in operating systems. In particular, we use the definition and results in Chapter 2 of [2].

We first informally review some definitions. A task system $C = (\tau, \lessdot)$ is a set of tasks, $\tau = \{T_1, T_2, \ldots, T_n\}$, together with a precedence relation, $\lessdot$, where $T \lessdot T'$ means that T must be completed before T' begins. Any execution sequence of C must obey the precedence relation. Each task T is associated with two subsets, the domain D_T and the range R_T, of the memory cells. When T starts it reads values from its domain, and when T terminates it writes values into its range. T and T' are noninterfering if either $T \lessdot T'$, or $T' \lessdot T$, or $R_T \cap R_{T'} = R_T \cap D_{T'} = D_T \cap R_{T'} = \emptyset$. Tasks $\{T_1, \ldots, T_n\}$ are mutually noninterfering if every pair of tasks T_i and T_j $(i \neq j)$ are noninterfering. We will use the following theorem which is stated and proved in [2], pp. 39-40.

Theorem: Task systems consisting of mutually noninterfering tasks are determinate.

The definition of determinancy of task systems requires a long development, [2], pp. 35-38, which we will not review here. For the purpose of proving the correctness of the Algorithm PF, it suffices to note that determinancy of a task system implies that for the same initial memory state, any execution sequence of the task system will end up with the same final memory state. We will define a set of task systems, and prove that each of them contains mutually noninterfering tasks. Then, we will use the above theorem to conclude that Algorithm F and Algorithm PF compute identical results.

For each $1 \le i, j, k \le$ NODES, let T_{kij} denote the task

"for D[i, j] ≥ D[i, k] + D[k, j] then
D[i, j]:= D[i, k] + D[k, j]".

For each k = 1, . . . , NODES, define task system $C_k = (\tau_k, \emptyset)$, where task set $\tau_k = \{T_{kij} \mid 1 \le i, j \le \text{NODES}\}$ and $\emptyset$ is the null precedence relation, i.e. no task needs to precede any other task. We will now show that each C_k contains mutually noninterfering tasks, and thus conclude that every execution sequence of C_k produces the same result as Algorithm F's execution sequence does. We will use M_{ij} to denote the memory cell for the variable D[i, j]. $M_{ij} = M_{ab}$ if and only if i = a and j = b. We will use D_{kij} and R_{kij} to denote the domain and range of task T_{kij}.

Remark 7: (a) $D_{kij} = \{M_{ij}, M_{ik}, M_{kj}\}$

(b) $R_{kij} \subset \{M_{ij}\}$

(c) If the input network has no negative cycle, then $R_{kkj} = R_{kik} = \emptyset$.

Parts (a) and (b) follow immediately from the definitions of domain and range of a task. For part (c), T_{kkj} contains the test "D[k, j] > D[k, k] + D[k, j]". Since the network has no negative cycle, D[k, k] is nonnegative. Thus the test result is always false, and the content of M_{kj} will not be changed. $R_{kkj} = \emptyset$ follows. Similarly, $R_{kik} = \emptyset$ also follows.

Remark 8: If the input network has no negative cycle, then τ_k contains mutually noninterfering tasks.

Because there are no precedence constraints between tasks in τ_k, we need to prove that $R_{kij} \cap R_{kab} = R_{kij} \cap D_{kab} = D_{kij} \cap R_{kab} = \emptyset$, for all (i, j) ≠ (a, b). $R_{kij} \cap R_{kab} \subset \{M_{ij}\} \cap \{M_{ab}\} = \emptyset$, because (i, j) ≠ (a, b). $R_{kij} \cap D_{kab} \subset \{M_{ij}\} \cap \{M_{ab}, M_{ak}, M_{kb}\} = \emptyset$, for (i, j) ≠ (a, b), j ≠ k, and i ≠ k. Similarly $D_{kij} \cap R_{kab} = \emptyset$. It follows that τ_k contains mutually noninterfering tasks, for k = 1, . . . , NODES. As noted before, this implies that Algorithm PF is correct.

Algorithm PF is programmed to run on the HEP computer. The number of processes created is minimized in order to reduce the overhead (of the create operation). The logic of our program referred to as Algorithm HEPPF (HEP parallel Floyd) is as follows:

Algorithm HEPPF

Process MASTER

```
1  SYN := 0;
2  for ℓ := 1 step 1 until K-1 do
3    create WORKER(ℓ);
4  execute WORKER(K)
```

Process WORKER(ℓ)

```
1  for k := 1 step 1 until NODES do
2  begin
3    for i := step K until NODES do
4      if D[i, k] < ∞ then
5        for j := 1 step 1 until NODES do
6          execute $T_{kij}$;
7    lock SYN; SYN := SYN + 1; unlock SYN;
8 L1: if SYN < K * k then go to L1
9  end
```

Algorithm HEPPF was coded and run for the experimental timing study. Experiments used randomly generated 20-, 30-, and 40-node networks. NODES x NODES arc length matrices with different densities of non-infinity entries distributed uniformly from 0 to 99 were generated. The results of our timing study are shown in Table 1. Let T_K denote the experimental running time of the algorithm with K processors. Let S_K and E_K denote the speed-up, T_1/T_K, and efficiency, S_K/K, respectively. The efficiency of this algorithm for networks with 40, 30, and 20 nodes is plotted in Figures 3, 4, and 5. It is evident that the efficiency tends to be high when the number of nodes in the network is a multiple of K, the number of processors. For in such a case, each WORKER process does exactly the same amount of work, but in the case where K does not divide NODES exactly, all WORKERs do not do the same amount of processing. For example, for each K, WORKER(1) performs NODES/K executions of statements 4 to 6, but WORKER(K) performs NODES/K executions of statements 4 to 6. The WORKERs which finish their work earlier must wait for all others, before starting on the next iteration. Thus the theoretical speed-up should be approximately NODES/NODES/K. More precisely, if we let t_1 denote the time for executing one iteration of the for loop in statement 3 of procedure of WORKER, and t_2 denote the time for executing statements 1, 8, 9, and 10 once, then the theoretical speed-up is

$$TS_K = \frac{T_1}{T_K} = \frac{(\text{NODES}\ t_1 + t_2)\ \text{NODES}}{\left(\left\lceil\frac{\text{NODES}}{K}\right\rceil t_1 + t_2\right)\ \text{NODES}} = \frac{\text{NODES} + t_2/t_1}{\left\lceil\frac{\text{NODES}}{K}\right\rceil + t_2/t_1}$$

For our compiled code of Algorithm HEPPF, t_2/t_1 is estimated to be approximately 1/(2NODES+1). Using this estimate, the ratio

$$\frac{\text{observed efficiency}}{\text{theoretical efficiency}} = \frac{E_K}{TS_K/K}$$

is calculated and plotted in Figure 6. From this plot we observe that the overhead for the create and the synchronization is relatively small when the input network is dense.

5. CONCLUSION

Two parallel shortest-path algorithms are designed and proved correct in this paper. They were both programmed to run on the HEP computer.

For the first algorithm, i.e. Algorithm PPDM, random highway-like sparse networks were generated and used as inputs. We observed empirically a speed-up of three when five processors were employed, for networks with 75 or more nodes. For the second algorithm, i.e. Algorithm HEPPF, random arc-length matrices of order up to 40 were generated and used as inputs. We found that the efficiency is higher for larger and denser networks. Thus we have clearly demonstrated theoretically as well as empirically that parallel processing techniques can be used profitably to speed up determination of shortest paths in large networks. We have also shown how this can be accomplished.

REFERENCES

[1] E. Arjomandi, A Study of Parallelism in Graph Theory, Doctoral thesis, Dept. of Computer Science, Univ. of Toronto, 1975 (available as Technical Report No. 86).

[2] E. G. Coffman, Jr. and P. J. Denning, Operating Systems Theory, Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

[3] E.V. Denardo and B. L. Fox, "Shortest-route methods: 1. reaching, pruning, and buckets," Oper. Res., 27 (1979), pp. 161-186.

[4] N. Deo, Graph Theory with Applications to Engineering and Computer Science, Prentice-Hall, Englewood Cliffs, New Jersey, 1974.

[5] N. Deo and C. Y. Pang, Shortest Path Algorithms: Taxonomy and Annotation, Tech. Report No. CS-80-057, Computer Science Dept., Washington State Univ., Pullman, WA (March 1980).

[6] R.B. Dial, F. Glover, D. Karney and D. Klingman, "A computational analysis of alternative algorithms and labeling techniques for finding shortest path trees," Networks, 9 (1979), pp. 215-248.

[7] E. Dijkstra, "A note on two problems in connexion with graphs," Numerische Mithematik, 1, (1959), pp. 269-271.

[8] D. M. Eckstein, Parallel Processing Using Depth-Frist and Breadth-First Search, Doctoral thesis, Dept. of Computer Science, Univ. of Iowa, Iowa City, Iowa, July 1977.

[9] D. M. Eckstein and D. A. Alton, "Parallel searching of non-sparse graphs," to appear in SIAM J. Comput.

[10] R. W. Floyd, "Algorithm 97: shortest path," Comm. ACM, 5 (1962), p. 345.

[11] M. J. Flynn, "Very high-speed computing systems," Proc. IEEE , 54 (1966), pp. 1901-1909.

[12] J. Gosch, "Computer processes multiple instruction sets, multiple data streams," Electronics, (Oct. 1979), pp. 77-78.

[13] D. S. Hirschberg, A. K. Chandra and D. V. Sarwate, "Computing connected components on parallel computers," Comm. ACM, 22 (1979), pp. 461-464.

[14] E. F. Moore, "The shortest paths through a maze," Proc. Internat. Symp. on Theory of Switching, 1957, pp. 285-292.

[15] U. Pape, "Implementation and efficiency of Moore-algorithms for the shortest route problems," Math. Programming, 7 (1974), pp. 212-222.

[16] M. Pollack and Wiebenson, "Solutions of the shortest-route problem--a review," Oper. Res., 8 (1960), pp. 224-230.

[17] E. Reghbati (Arjomandi) and D. G. Corneil, "Parallel computations in graph theory," SIAM J. Comput., 7 (May 1978), pp. 230-236.

[18] E. Reingold, J. Nievergelt, and N. Deo, Combinatorial Algorithms: Theory and Practice, Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

[19] C. Savang, Parallel Algorithms for Graph Theoretic Problems, Ph.D. Thesis, Math. Dept., Univ. of Illinois at Urbana-Champaign (Aug. 1977), Report ACT-4, Coordinated Science Lab., Univ. of Illinois.

[20] B. J. Smith, "A pipelined, shared resource MIMD computer," Internat. Conf. on Parallel Processing, 1978.

[21] D. Van Vliet, "improved shortest path algorithm for transportation networks," Transporation Res., 12 (1978), pp. 7-20.

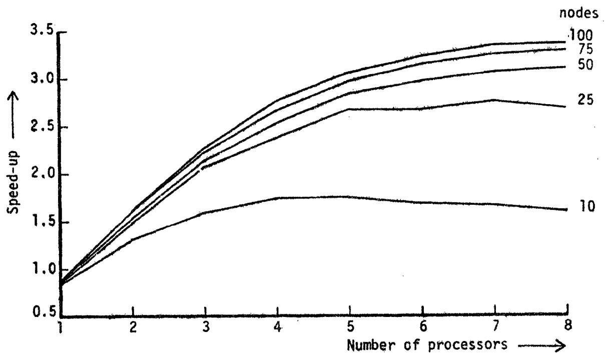


Figure 1. Average Speed-up

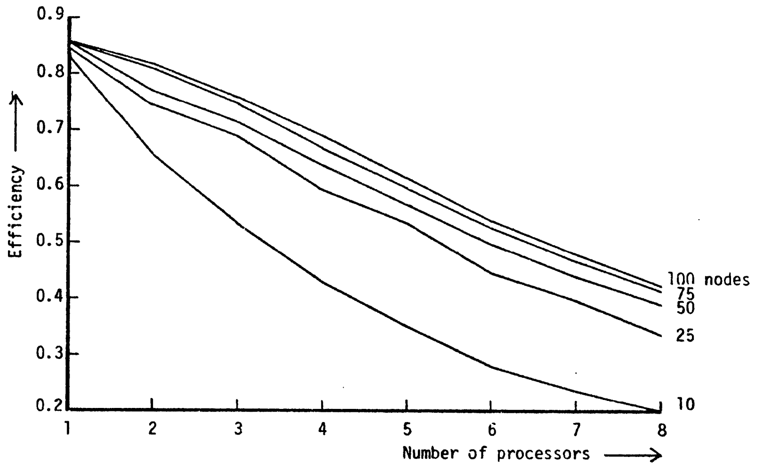


Figure 2. Average Efficiency

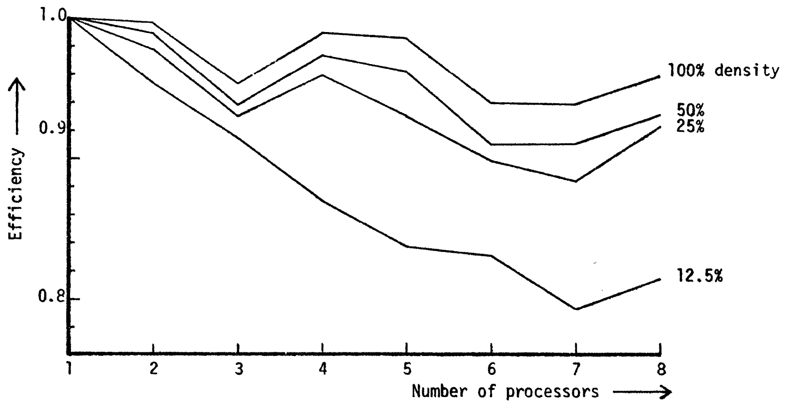


Figure 3. Efficiency for 40-Node Networks

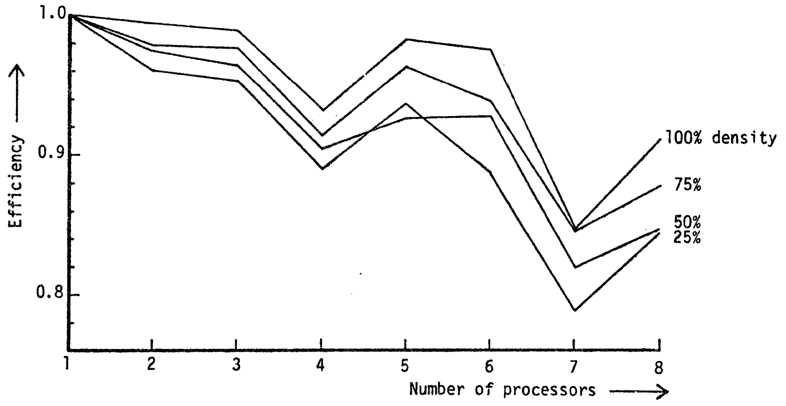


Figure 4. Efficiency for 30-Node Networks

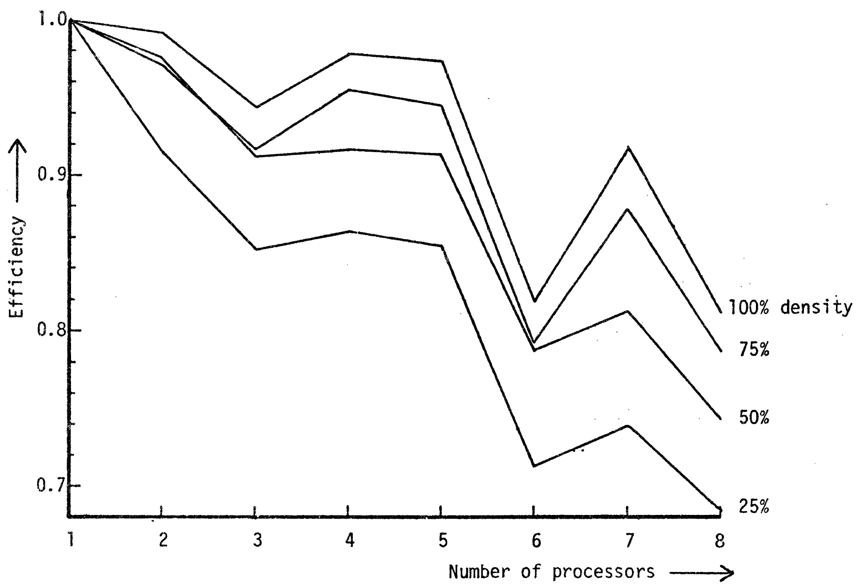


Figure 5. Efficiency for 20-Node Networks

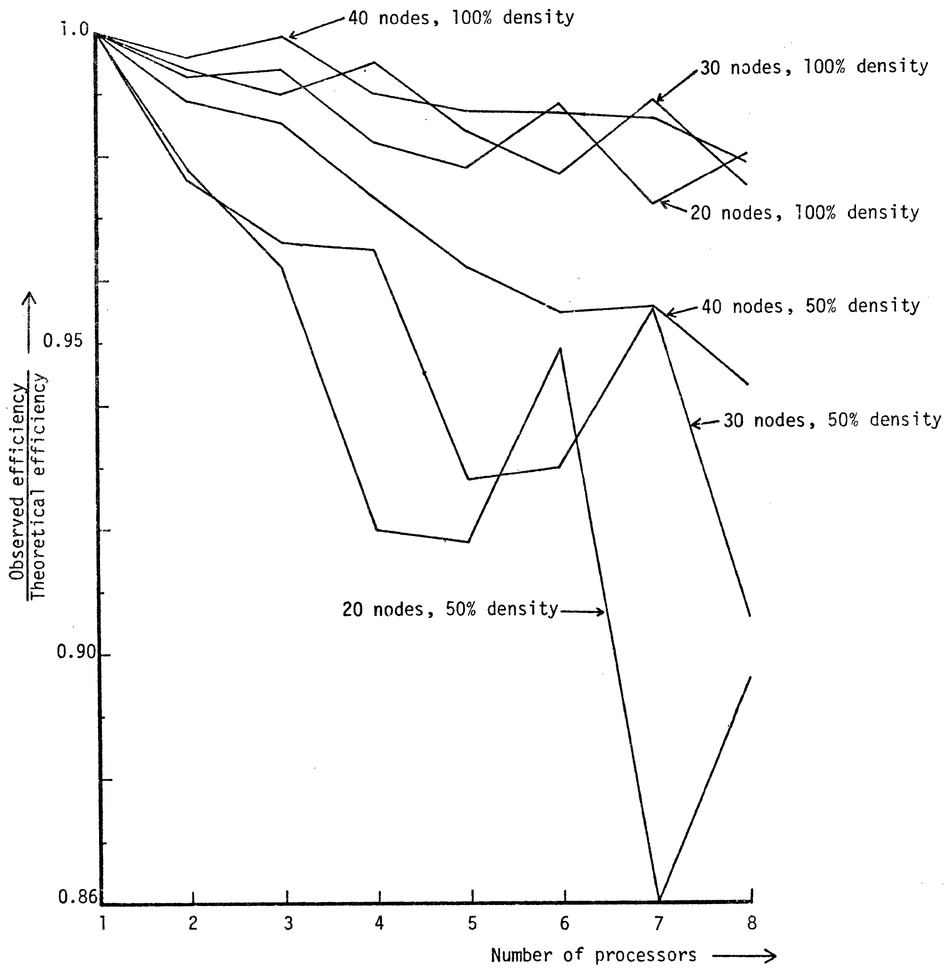


Figure 6. Observed/Theoretical Efficiency

Table 1. Running time of Algorithm HEPPF (in secs).

		Density			
NODES = 40		100%	50%	25%	12.5%
No. of Processors	1	1.30478	1.24866	1.13903	0.88217
	2	0.65522	0.63133	0.58283	0.46305
	3	0.45726	0.44399	0.40812	0.32185
	4	0.32989	0.32097	0.29727	0.25366
	5	0.26484	0.25992	0.24512	0.21071
	6	0.23169	0.22906	0.21123	0.17719
	7	0.19889	0.19627	0.18433	0.15915
	8	0.16693	0.16594	0.15423	0.13571
NODES = 30		100%	75%	50%	25%
No. of Processors	1	0.55024	0.53037	0.49828	0.45644
	2	0.27684	0.27116	0.25537	0.23737
	3	0.18544	0.18088	0.17221	0.15966
	4	0.14774	0.14519	0.13785	0.12816
	5	0.11213	0.11039	0.10760	0.09756
	6	0.09417	0.09429	0.08958	0.08582
	7	0.09294	0.08973	0.08699	0.08280
	8	0.07550	0.07559	0.07361	0.06762
NODES = 20		100%	75%	50%	25%
No. of Processors	1	0.16299	0.15615	0.14249	0.11844
	2	0.08213	0.08028	0.07291	0.06457
	3	0.05753	0.05683	0.05195	0.04626
	4	0.04165	0.04086	0.03888	0.03528
	5	0.03348	0.03304	0.03118	0.02770
	6	0.03317	0.03287	0.03016	0.02767
	7	0.02533	0.02541	0.02503	0.02292
	8	0.02513	0.02479	0.02401	0.02166

A PARTITION ALGORITHM FOR PARALLEL AND DISTRIBUTED PROCESSING *

Shyue B. Wu and Ming T. Liu

Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210

Summary

An efficient partition algorithm can be applied to solve problems in assignment of tasks and resources [5] as well as problems in scheduling and control of distributed processes [3]. Successful solutions to these problems can increase system performance and reliability. This paper presents an efficient partition algorithm and discusses its use in solving the assignment and scheduling problems for parallel and distributed processing in large multi-microcomputer systems.

A general case of the partition problem can be stated as follows: given a graph, G=(V,L), where V is a set of nodes and L is a set of links, each associated with a positive number representing the weight (which in turn represents a communication or execution cost) of the link; we are to partition the graph into K disjoint nonempty subgraphs in such a way that the sum (called partition cost) of the weights of the links which separate the subgraphs is minimized.

An efficient solution to the partition problem for K=2 can be directly obtained from using any of several available network flow algorithms [1] [3]. However, for K>2, the problem has been known to be NP-complete. With the introduction of microcomputers, a distributed system with more than two processors are more common. Therefore, it is important to obtain an efficient algorithm, applicable to K-processor (K>2) systems, for the partition problem. We will show how such an algorithm can be obtained from the use of network flow algorithms.

In order to conform with terminology used in network flow theory, we shall henceforth use the terms networks and subnetworks rather than graphs and subgraphs in this paper.

A K-cut of a network is a minimum set of links, the removal of which separates the network into K disjoint nonempty subnetworks. The cost of a K_cut is the sum of the weights of the links in the K_cut.

From the above, we can see that the partition problem in general cases is equivalent to finding an optimal K_cut (i.e., a K_cut whose cost is minimum among all possible K_cuts). Thus our algorithm for the partition problem can be stated as follows:

Partition Algorithm

```
i <-- 2;
Obtain a 2_cut of the given network by using a
   network flow algorithm;
Do while (i < K);
   Obtain a 2_cut of each of the two subnetworks
      resulting from the cut previously selected
      by using a network flow algorithm;
   Pick up the cut whose cost is minimum among
      all unselected 2_cuts obtained so far;
   The i+1_cut is equivalent to the i_cut plus
      the selected cut;
   i <-- i+1;
End;
```

Our partition algorithm is efficient in the sense that it uses a network flow algorithm only in the order of K (O(K)) times. Our algorithm is also good in the sense that it yields a good solution. The following two theorems are stated to show that our algorithm results in a solution with minimum cost if the given network is tree-like. The empirical results are also presented below to show that our algorithm results in a solution with near minimum cost in general cases. For the proof of the theorems and the detail of the performance studies, readers are referred to [5].

Theorem 1: For a tree-like network, if two nodes are in the same subnetwork of an optimal K_cut (K>2), then these two nodes are in the same subnetwork of an optimal K-i_cut (1<=i<=K-2).

Theorem 2: For a tree-like network, any optimal K-1_cut is a subset of an optimal K_cut.

Our partition algorithm has been programmed for testing 720 randomly generated networks each with six nodes. For each test network, we collected error percentage of an i_cut (E(i)), which is defined by (SUBOPT - OPT)/OPT, where SUBOPT is the cost of the i_cut obtained by using our algorithm and OPT is the cost of the optimal i_cut obtained by an exhaustive enumeration method.

The distribution of E(i) is given in Table 1, where NOPT represents the number of networks with

* Research reported herein was supported in part by the NSF under grant MCS-77-23496.

an optimal solution (no error), 1% represents the number of networks with an error between 0% and 1% (0% < E(i) <= 1%), and so on. For example, 10 of the 720 networks for which we obtained a 4_cut, had solutions with errors between 3% and 4%.

Table 1: Distribution of Error Percentage

i-cut	NOPT	1%	2%	3%	4%	5%	MORE
3	696	0	0	4	1	0	19
4	655	7	4	13	10	13	18
5	581	42	43	11	15	14	14

From Table 1, we see that our partition algorithm obtains an optimal solution about 90% (1932/2160) of the time, and obtains a solution with more than 5% of error only about 2.5% (51/2160) of the time. The average E(i), which is not shown in Table 1, is approximately in the order of 10 ** -3 (0.1%). Thus we feel that our partition algorithm is good enough for general applications.

With the introduction of microcomputers, there has been a great interest in constructing a distributed system from a large number of microcomputers [2]. In such a system, the memory of each processor is restricted [2]. Therefore, there is a need to distributed (or assign) system software resources, such as modules of operating systems, over (to) the processors in the system; this is called resource assignment or resource allocation.

In the following, we outline the use of our algorithm to solve the resource assignment problem in large multi-microcomputer systems [4], and discuss the impact of the solution.

The resource assignment problem in large multi-microcomputer systems can be stated as follows: 1) given a module network, G=(M,R), where M is a set of modules (or software resources) and R is a set of links, each of which (Rij) is associated with a link weight representing the communication cost per unit distance between two modules (Mi and Mj); 2) given a system node network, G=(P,D), where P is a set of processors and D is a set of links, each of which (Dij) is associated with a link weight representing the number of unit distance between two processors (Pi and Pj); and 3) we are to find a mapping function f : M --> P such that the total cost given by $\sum \sum$ Rij * D f(i)f(j) is minimized.

We can use the partition algorithm to obtain a K_cut of a module network and that of a system node network. The subnetworks of the module network after the K_cut can then be assigned to the corresponding subnetwork of the system node network. Therefore, we can assign system software resources to system nodes through the use of our partition algorithm.

A better resource assignment can result in less communication cost when a set of software resources are requested for service. Therefore, a better resource assignment can eliminate unnecessary message traffic in a system, thereby minimizing interconnection limitation which arises due to message traffic saturation. The minimization of interconnection limitation, in terms of the number of nodes that can be interconnected and the amount of message traffic that can be supported, may make it possible to use large multi-microcomputer systems for a variety of applications [4].

A better resource assignment can also make the task assignment easier in large multi-microcomputer systems. The task assignment problem is how to assign program modules to system nodes (processors) so as to minimize the total execution and communication costs. In large multi-microcomputer systems, each node may be dedicated to provide a specific function. A task requesting a specific resource is likely to be assigned to the node providing this specific resource. Therefore, the task assignment can be easily achieved in this case.

For task assignment, if there is an overloaded node, the program modules assigned to this node can be reassigned to other nodes as long as the extra cost introduced by the reassignment is paid for. The decision of reassignment can be made by using our algorithm to partition the reassignment network [3] to see whether the reassignment is worthwhile. Through the use of reassignment of tasks, it is likely that scheduling and control of distributed processes in large multi-microcomputer systems can be improved.

References

1. T. Y. Cheung, "Computational Comparison of Eight Methods for the Maximum Flow Network Problem," ACM Trans. on Math. Software, Vol. 6, No. 1, March 1980, PP. 1-16.

2. J. K. Ousterhout, et al., "Medusa: An Experiment in Distributed Operating System Structure," Comm. ACM, Vol. 22, No. 2., February 1980, PP. 92-105.

3. H. S. Stone and S. H. Bokhari, "Control of Distributed Processes," Computer, Vol. 11, No. 7, July 1978, PP. 97-106.

4. S. B. Wu and M. T. Liu, "A Cluster Structure as an Interconnection Network for Large Multi-microcomputer Systems," submitted for publication.

5. S. B. Wu and M. T. Liu, "Assignment of Tasks and Resources for Distributed Processing," Proc. COMPCON 80 Fall, September 1980, to appear.

SESSION 9: DATABASE ARCHITECTURE AND SOFTWARE I

A HIGHLY CONCURRENT TREE MACHINE FOR DATABASE APPLICATIONS[1]

S. W. Song
Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pa. 15213

Abstract -- In this paper we describe a tree-structured machine, suitable for VLSI implementation, that handles all the frequently encountered database operations efficiently. N elements are maintained on an N-processor version of the tree machine. We shall describe algorithms, based on a new concept of associative search, for insertion and deletion of elements in the tree. The tree machine can handle a large class of searching problems. Insertion, deletion, queries, and updates can all be processed in O(log N) time units. It is especially suitable when a sequence of such operations is to be processed in a pipelined fashion. I/O time dominates the total time to execute more complex operations such as join of two relations or sorting. Once data are in the machine, it takes usually O(log N) time units for the first results to emerge. Therefore it is very suitable for on-line systems where fast response time is needed. Some major obstacles to be overcome are discussed.

1. Introduction

Database management systems are concerned with the task of providing fast retrieval, storage and update operations, in response to users' requests. In recent years, database systems have been growing in size and software systems to manage them are becoming increasingly more sophisticated and complex. Also, as demand for services increases, many data processing installations have reached the point of saturation. Backend database systems have been proposed as a solution to the problem of overloaded installations. The reader is referred to [17] for a discussion on such systems.

[1]This research was supported in part by the Defense Advanced Research Projects Agency under Contract F33615-78-C-1551 (monitored by the Air Force Office of Scientific Research), in part by the National Science Foundation under Grant MCS 78-236-76, and in part by the Office of Naval Research under Contracts N00014-76-C-0370 and N00014-80-C-0236. The author is supported in part by Fundacao de Amparo a Pesquisa do Estado de Sao Paulo under Grant 76/517, and in part by the Institute of Mathematics and Statistics of the University of Sao Paulo, Brazil.

Various design efforts of specialized hardware with novel architectures to handle database problems have been carried out [1], [7], [14], [20], [25]. There are abundant literature and survey articles on these designs [11], [13], [22], [24].

Very Large Scale Integrated circuitry has been increasing in speed and density at an amazing rate. The amount of components on a single chip is claimed to reach several millions by the end of this decade [18]. This has aroused a surge of interest in developing customized designs of algorithms implementable on silicon. Leiserson [15] proposes a systolic priority queue, a structure with the possible operations of insertion, deletion and minimum extraction. The rebound sorter of Chen et al. [8] handles sorting problems. Bentley and Kung [3] present a design of a tree machine for searching problems. Kung and Lehman [12] propose several linear arrays of processors capable of performing such operations as intersection, join and duplicate removal. These designs provide efficient handling of specific tasks. In database applications, some queries require the execution of a sequence of database operations before the answer is obtained. It is therefore desirable to have a single special-purpose device which can provide efficient solutions to all basic database operations such as search, insertion, deletion, updates, sort, join, union, etc. For this purpose we have chosen the tree machine of Bentley and Kung [3], which can solve all of the "decomposable searching problems" [4], and attempted to extend it to handle other basic operations. Tree-structured machines have been proposed to handle other types of problems. The designs by Berkling [5], Mago [16], Sequin, Despain, and Patterson [21] and Wilner [26] are general-purpose computing devices. Hollaar [10] presents a design for merging sorted lists. Browning [6] considers several applications as sorting and NP-complete problems.

CH1569-3/80/0000-0259$00.75 © 1980 IEEE

2. General System Configuration

In Figure 1 we show the tree machine acting as a back-end machine to a host computer. Users' database manipulation commands are passed on to the tree controller which, using some auxiliary information, will locate where in the mass storage the needed information reside. Data clustering is an important issue and is discussed in [2]. It will then command the I/O controller to transfer data to the tree machine. Loading of the tree machine will be the bottleneck of the system and will be discussed later in the section on implementation issues. Once the tree machine is loaded, the tree controller will issue commands to the tree machine to carry out the required operations. The results output from the tree machine will be returned to the host computer.

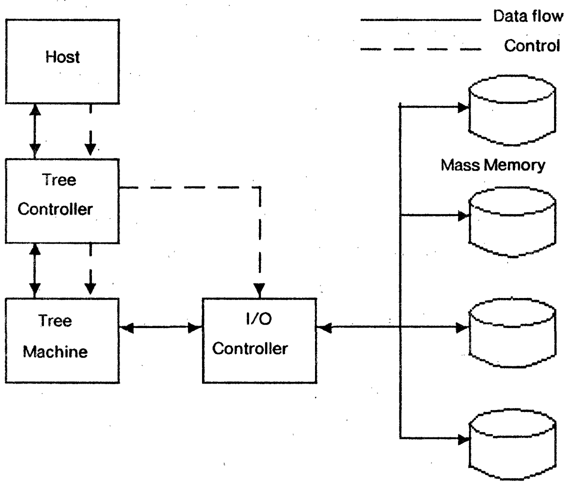


Figure 1: A system configuration.

3. The Tree Machine

The tree machine has three kinds of nodes (see Figure 2): O-nodes, □-nodes, and Δ-nodes. Each one of a collection of records resides in a □-node, which is provided with some logic to carry out a limited repertoire of instructions. The O-nodes broadcast streams of instructions and/or data to the □-nodes where they are executed in parallel. The □-nodes compute results which are then combined by the Δ-nodes to produce the final result. Selection of records satisfying a conjunction of conditions can easily be performed by broadcasting the conditions to the □-nodes which can then decide which ones are to be selected. First we shall review the insertion and deletion algorithms mentioned in [3], and propose a new space allocation scheme. Then we shall discuss how data flow in the O-nodes and Δ-nodes should be disciplined.

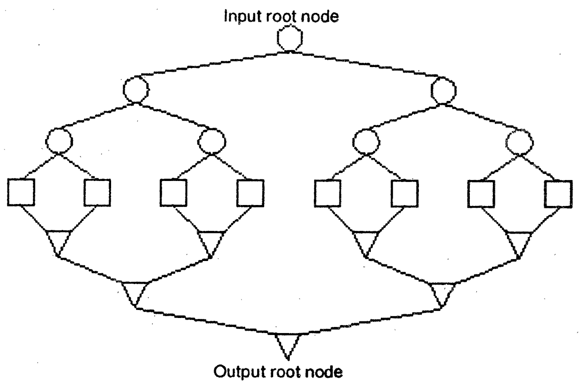


Figure 2: The tree machine.

3.1. A New Space Allocation Scheme

One way of doing insertion is to maintain a count in each of the O-nodes. Each count in a O-node specifies the number of free □-nodes which are its descendants. Each time a new element is to be inserted, a O-node will pass on the element to the son which has free □-nodes below (choosing an arbitrary one if both are eligible). Then it will update its own count by decrementing it by one. Similarly, when an element contained in a □-node is deleted, some of the O-nodes need to have their counts updated. More specifically, these are all the O-nodes which lie on the path from the input root node to the particular □-node where deletion has occurred. This can be done by proceeding backwards from the deleted node to the input root node, adjusting the counts on its way up. O(log N) steps are therefore necessary to adjust the counts, where N is the total number of □-nodes of the tree. While this scheme has the advantage of being very general, it has the drawback of requiring a storage for the count, as well as the associated logic needed for updates, in each of the O-nodes. Furthermore, since counts need to be adjusted after a deletion is made, it makes pipelining of arbitrary sequence of insertions and deletions more difficult.

We wish to design new insertion and deletion algorithms with the following two objectives:

1. Arbitrary sequence of insertions and deletions can be easily pipelined.

2. No counts nor associated logic for updates are to be maintained in the O-nodes.

We have found a way to achieve the above if the following assumptions are made:

1. A single count is kept in the tree controller.

2. For each delete command issued by the tree controller, there exists always one and only one item in a □-node which will be deleted.

Consider each □-node as containing storage for two fields, node.freeposition and node.content. If a □-node is free, then node.freeposition contains an integer from 0 to N-1, where N is the total number of □-nodes in the tree. Also, for simplicity of notation, we write n_i for the □-node whose freeposition field contains i, $0 \leq i \leq N-1$. if a □-node is occupied then its freeposition contains Λ. Node.content is the value of the item stored in the □-node which, for simplicity, will be assumed to be an integer.

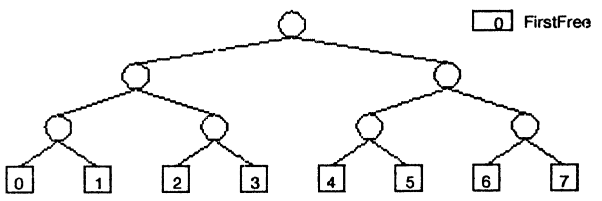


Figure 3: An empty tree.

If the tree is empty (i.e., it stores the empty collection), we assume that the free □-nodes of the tree are n_0, n_1, n_2, ..., n_{N-1}, in any order. (See Figure 3, where we have omitted the bottom half of the tree machine.) We also assume that the tree controller maintains an integer count called FirstFree, such that the free □-nodes are $n_{FirstFree}$, $n_{FirstFree+1}$, ... , n_{N-1}. FirstFree contains 0 if the tree is empty and contains N if the tree is full.

3.1.1. Insertion

To insert an element X, the tree controller will generate an insert instruction which has two parts, namely, instruction.freeposition and instruction.content. Instruction.freeposition will indicate which □-node is to be removed from the pool of free □-nodes. The tree controller assigns $n_{FirstFree}$ to be that node. Instruction.content contains the value to be inserted. This is shown as follows.

```
instruction.freeposition := FirstFree;
instruction.content := X;
FirstFree := FirstFree + 1
```

All that the O-nodes need to do is merely to broadcast the instruction to its two sons. Simultaneously, each of the □-nodes will try to see if it has been selected as the node to receive the element being inserted. Exactly one such node will be found and this will mark itself as occupied after redefining its content field. This is shown as follows. (Figure 4 shows the tree after 6 elements have been inserted to an initially empty one.)

```
if node.freeposition = instruction.freeposition
 then node.content := instruction.content;
      node.freeposition := Λ
```

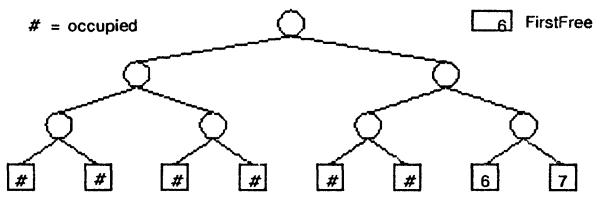


Figure 4: After 6 insertions.

3.1.2. Deletion

We consider deletion of an element from the tree based on the content of that element. Suppose that we wish to delete the element X from the tree. Note that, by the assumption made before, always one and only one □-node will be freed, whenever a delete command is issued. This means that the tree controller will know beforehand that one of the originally occupied □-nodes will be able to return to the pool of free □-nodes, even though it does not know which one. Therefore, the delete instruction issued by the controller will contain not only the content X to guide the deletion, but also the value that should be stored into the freeposition field of the freed node.

```
FirstFree := FirstFree - 1;
instruction.freeposition := FirstFree;
instruction.content := X
```

Again the O-nodes need not to do anything more complicated than merely broadcasting the instructions. Each of the □-nodes will attempt to match the content it has with the content in the instruction. Only one will find a match and that one will be immediately returned to the free pool by redefining its freeposition field.

```
if node.content = instruction.content
 then node.freeposition := instruction.freeposition;
     node.content := Λ
```

Here we have used Λ to indicate the null content. Note that the functions performed by the □-node in the insert and delete commands are symmetrical. We obtain one from the other by merely interchanging the words *content* and *freeposition*. Figure 5 shows what remains after two deletions have been made to the example illustrated by Figure 4.

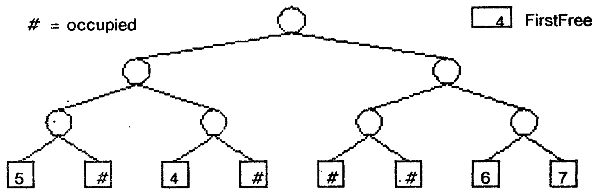


Figure 5: After 2 deletions.

3.1.3. Comments on the Algorithms

In the original insertion scheme mentioned at the beginning of this section, the element to be inserted is passed down the tree through a path of O-nodes until a free □-node is reached. The selection of this path is guided by the O-nodes which use their own count information as well as those of their sons. In the new scheme, the element being inserted does not follow any particular path, but is merely broadcast to all the □-nodes. It takes advantage of the ability of content addressability of the tree machine to do the selection of the free □-node. Also, in the original deletion scheme, log N counts in the O-nodes need to be adjusted. Since we do not know which counts are to be incremented until the deletion is done, pipelining was not so easy to achieve. Here we have only two values to be adjusted, namely, those of FirstFree and of the freeposition field of the deleted node. The interesting thing is that both values can be determined at the time the delete command is issued by the controller. The reader can easily see that pipelining arbitrary sequence of deletes and inserts presents no problem at all. In some sense, we have *factored out* the counts and logic from the O-nodes to the tree controller, thereby reducing the space needed for its implementation. In VLSI designs, there is often a trade-off between space and time. In this case, however, the new space allocation scheme has allowed us to reduce space requirements and at the same time achieve a better performance.

3.1.4. Comments on the Restrictions

Some restrictions have been made in order to make the proposed scheme applicable. In cases where a delete command may find many elements or none qualified for deletion, some more processing is required before the delete command can be issued. First those elements qualified for deletion should be selected and deletion can proceed using fields which uniquely identify them. In some applications deletion of a record is performed after some processing has been done on the same record. Therefore its presence among the collection of records is certain. Furthermore, if deletion is based on a primary key, then the restrictions are met and the scheme applies.

3.2. Disciplining the Data Flow

In operations where only one output is involved, new commands can be issued to the machine while the results are being handled at the Δ-nodes to be output. In other words, pipelining is easily achieved. In some operations, however, many results are produced in the □-nodes. These will traverse through the Δ-nodes until they reach the output root node. Given the funneling nature of the output binary tree (i.e., the bottom part of the tree machine, as shown in Figure 2), the Δ-nodes should cooperate among themselves in order to produce an orderly evacuation of the many results. We shall provide some storage in each Δ-node. If its storage is empty then the Δ-node will examine its two sons and take the contents of a non-empty son. If both sons have information to be transmitted, then it will select one according to some fixed rule (such as always picking the leftmost, or selecting the one with minimum value in some specified key).

Some of the results may have to be retained in the □-nodes for quite a while before they are accepted by the Δ-nodes. In order to protect these results from being destroyed by the incoming stream of instruction or data, the broadcasting of information in the O-nodes should also be disciplined. This can be done by the tree controller which can turn off the pipeline and wait until the machine is flushed before starting another operation. To perform one operation, however, such as the full join, many tree machine instructions may be required, and some of these may also produce multiple results in the □-nodes. While it is reasonable to turn off the pipeline for different operations (such as a full join and a subsequent union operation), it is too expensive to do so with machine instructions performed within one operation. We now face a problem of trying to retain whenever possible pipelining of instructions, some of which may produce multiple results. Flow of the stream of instructions or data will not be continuous and intermittent pauses may sometimes be necessary. This requires the full cooperation of all the O-nodes and the tree controller. Each O-node has storage to hold the information to be broadcast. It will send this information to its two sons only if both are empty, i.e., ready to accept data. Each O-node will examine its two sons. If both are ready, then it will transfer its contents to its two sons and declare itself ready. Similarly, the tree controller will only put new information to be broadcast in the input root node if it is ready. The □-nodes can control the flow of information from the layer of O-nodes immediately above them by declaring themselves ready only if their results, if any, have already been taken out by the Δ-nodes.

3.2.1. Observation 1

If any result formed in a □-node is always readily taken out without delay, then broadcasting items $a_1, a_2, a_3, \ldots$ down the tree will result in alternate empty layers of O-nodes. With a tree machine of N □-nodes, (log N)/2 layers of O-nodes will be empty, as shown in Figure 6. Also, it takes log N steps for any item a_i (counted from the instant it enters the input root node) to reach a □-node.

3.2.2. Observation 2

Consider a situation as above, in which alternate layers of O-nodes are empty. Suppose now the result computed in some of the □-nodes cannot be removed by the Δ-nodes for a long time. This □-node will therefore start to block the flow of

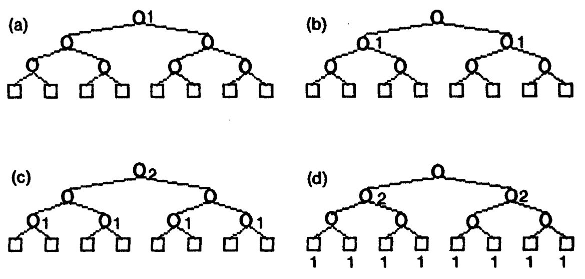


Figure 6: Alternate empty layers of O-nodes.

information above it until all the O-nodes on the path from the input root node to it are filled (see Figure 7). Since alternate layers of O-nodes are originally empty, (log N)/2 more new elements can still enter the tree before the path in question becomes full. Each of these new elements enter the input root node every other step. Therefore it takes log N steps to fill up this path.

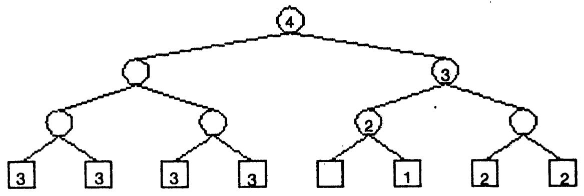


Figure 7: Blocking of flow.

3.2.3. Observation 3

If at a certain instant all the □-nodes are empty (creation of an "empty layer"), then this "empty layer" will be propagated toward the top of the tree in log N time. (Also, if the creation of an "empty layer" of □-nodes occurs every other step in a total of log N steps, then (log N)/2 alternate empty layers of O-nodes will be created, as indicated in Observation 1.)

3.2.4. Observation 4

Since each O-node broadcasts to its two sons only if both are ready, any item a_i which enters the tree will reach all the □-nodes, though not necessarily at the same time. However, the items will visit a fixed □-node in the *same* order they entered the input tree node.

4. Database Operations

First we briefly discuss the search problems. The reader is referred to [3] for details. Next we consider the sort and remove-duplicates operations. The description of the join operation will constitute the major part of this section. Finally, the union and intersection operations will be briefly mentioned. Other database operations (such as division) will not be shown simply because they will not add anything new to the presentation. If we associate one bit to each field of a tuple and consider it to be valid only if the corresponding bit is on, then projection can be done by manipulating the appropriate bits.

4.1. Search Problems

By means of broadcasting, all the N □-nodes of the tree machine can receive a message sent out from the input root node in O(log N) time. A variety of search problems has been considered in [3]. They also show that, with pipelining, M operations can be performed in O(M + log N) time. Furthermore, several selection operations can be pipelined, such that the time will be linear in the total number of conditions.

4.2. Sort and Remove-Duplicates

Sorting a collection of records on some specified key is surprisingly easy in the tree machine. Recall that when trying to output multiple results stored in the □-nodes, the Δ-nodes are instructed to accept data from a non-empty son. A selection rule is used if both sons are non-empty. If we use the rule of selecting the minimum value in a specified key, then the output records will be sorted in ascending order. See an example in Figure 8, where a simplified representation of the bottom part of the tree machine is shown. The records reside in arbitrary positions of the □-nodes. The example shows the first steps to output the records in sorted order.

In log N steps, the minimum of the collection of elements will emerge at the output root node. From then on, at every other step the next element in increasing order will exit the machine. (For simplicity, we have assumed enough datapath to transmit a tuple in a single cycle. This is similarly assumed throughout the paper. In a realistic situation, the time complexities should be multiplied by appropriate factors.)

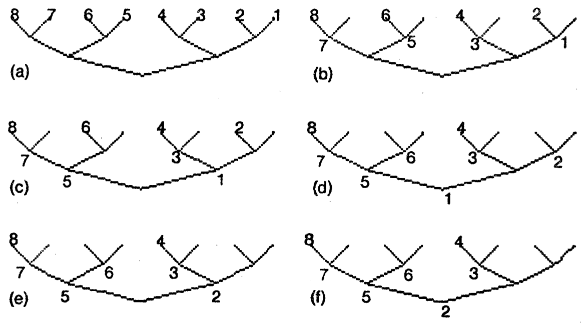


Figure 8: Sorting.

Now it becomes clear how duplicate removal from a collection of K records can be performed. The collection is simply sorted. At the output the controller tests pairs of consecutive elements for equality. For every pair of equal elements it deletes one of the occurrences and this can be done while the sorting is still going on. Therefore in essentially 2K steps we realize the duplicate removal operation.

4.3. The Join Operation

The join operation is performed on two relations over a specified attribute with common domain. The result of the join is another relation. A tuple of the result relation is composed by the concatenation of two tuples, one from each of the two relations, with identical values in the common attribute.

4.3.1. Preliminary Assumptions

We assume that a □-node has storage to hold a relation identification, a tuple and a tuple identification. While a tuple may require quite an amount of storage, the tuple id may need considerable less storage (log K bits if each of the K tuples of a relation is associated to a different number from 0 to K-1). The controller is provided with a parallel associative store, enough to hold log N entries. Each entry can hold a tuple and its corresponding tuple id. This associative store will allow the controller to retrieve a tuple content given its tuple id, assuming the tuple is present in the storage.

4.3.2. Actions of Different Node Types

Suppose we want to perform a join operation of two relations A and B, each with K tuples, over some given attribute. For convenience of exposition, tuples of relations A and B will be referred to as A-tuples and B-tuples, respectively. These tuples reside in arbitrary positions of the □-nodes. See Figure 9. To carry out the join operation, each of the three types of nodes will execute its instructions until the join operation is complete.

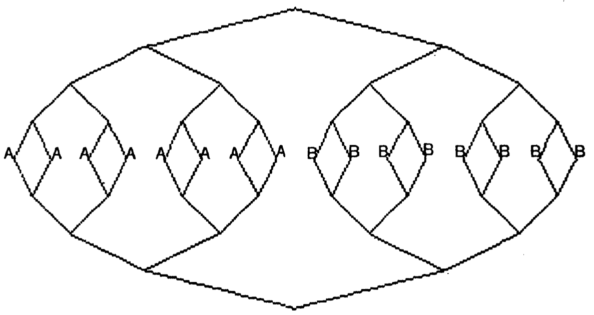


Figure 9: Two relations residing in the tree.

The O-nodes are instructed simply to broadcast whatever they receive to their sons, obeying the protocol established in the previous section.

The □-nodes holding A-tuples are instructed to send a copy of the tuple and its corresponding tuple id to the Δ-nodes. Again, for ease of exposition, an A-tuple plus its tuple id will be referred to as A-information. Once this is done, its mission is considered accomplished. Any information received from above will simply be ignored until the join operation is complete.

Each □-node holding a B-tuple is instructed to extract the attribute value over which the join is being performed and compare it with the incoming information, which include an attribute value of an A-tuple, as well as its tuple id (see below). In case of a match, a copy of the B-tuple and the received tuple id should be sent out to the Δ-nodes. (We shall refer to these as B-information.) This action is to be repeated for every incoming information. As we have seen earlier, these □-nodes can always hold the stream of information coming to them by declaring themselves not ready to accept new data.

The Δ-nodes are instructed simply to pass on data toward the output node of the tree. In case a Δ-node finds both sons with information to be transmitted, it will always give priority to the B-information.

4.3.3. Carrying Out a Join Operation

Starting from the situation as in Figure 9, all the □-nodes holding A-tuples will try to send out their A-information to the Δ-nodes. After log N steps, one of these A-information will emerge at the output. And from then on, a new information will emerge at every other step. If the output contains A-information, the controller will store the tuple and its id in the associative store. Furthermore, this tuple id plus the attribute value over which the join is being done are redirected to the circular input node of the tree, to be broadcast downward. (We shall refer to these as a-information.) After another log N steps, the first of these information will reach all the □-nodes (see Figure 10).

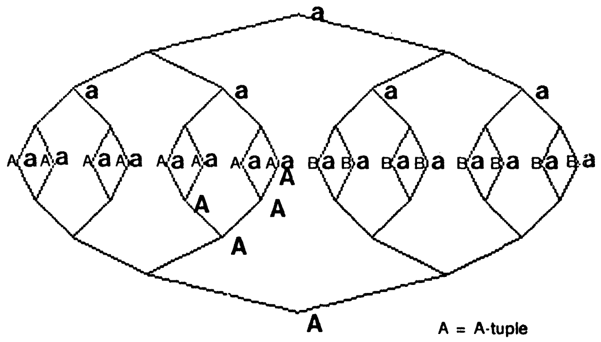


Figure 10

□-nodes holding A-tuples will ignore this information, without ever blocking its flow. □-nodes holding B-tuples, as they are instructed to, will try to match the two attribute values (one extracted from the tuple it contains and the other from the a-information it receives from above). In case of a match, it will make a copy of the tuple B and send it out together with the A-tuple's id (or B-information). This tuple id corresponds to some A-tuple which should be concatenated to the B-tuple to form a result tuple.

These B-information (many such may be formed) will start descending the Δ-nodes, log N steps being necessary for the first of them to reach the output root node (see Figure 11). If the output contains B-information, the tree controller will locate the A-tuple in the associative store, given its id. Thus the result tuple can readily be assembled.

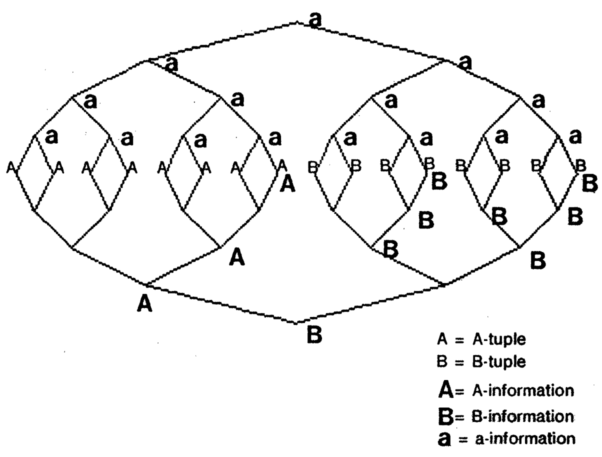


Figure 11

We now show that overflow will never occur, as more and more A-information are added to the tree. The □-nodes holding A-tuples ignore any information broadcast to them and will never block the downward flow. The □-nodes holding B-tuples may block the flow if, after a match, the B-information it has produced for output is still waiting to be taken by the Δ-nodes below. Let c_1 be the clock cycle in which the first B-information is formed in one or more □-nodes. In subsequent clock cycles, more B-information may be produced at the □-nodes. Let $c_2 > c_1$ be the nearest clock cycle to c_1 in which none of the □-nodes is holding any B-information. Recall the B-information have priority over the A-information to traverse among the Δ-nodes. Once the first of these B-information formed between c_1 and c_2 emerges at the output root node, an A-information will have a chance to get out only after all such B-information have been output. However, the leader of these B-information will take log N steps to reach the exit. During the same time, (log N)/2 of the A-information will have emerged. These will not cause overflow because, prior to c_1, the flow in the O-nodes has been unrestricted and, by Observation 1, alternate layers of O-nodes are empty. Therefore these A-information will be appropriately accommodated without causing overflow. At c_2, none of the □-nodes are holding any result to be output. Therefore, by Observation 2, each time this happens, an "empty layer" of □-nodes will be created and it takes log N cycles for the "empty layer" to propagate to the top. The next A-information will have a chance to exit the tree after all the B-information formed between c_1 and c_2 have been output, provided it has not been caught up by some other B-information produced at still later cycles. The last B-information will take at least log N cycles to get to the exit, therefore an empty input root node will have been created to accommodate the next A-information. By a similar reasoning and using Observation 4, we can show that each B-information which emerges from the tree machine will find the needed A-tuple in the associative storage.

The time necessary to perform the join can easily be computed if we fix our attention at the output node of the tree. From the instant the first A-information emerges at the output, some information, either A or B, will come out every other step. There are exactly K A-information to be output from the tree, and each B-information will correspond to a result tuple. Therefore it takes log N + 2(K + # of result tuples) steps to realize the full join of two relations.

4.4. Union and Intersection

The union and intersection of two sets A and B of K elements each can similarly be obtained. Briefly we describe how the intersection can be performed. All tuples of one relation are sent out to be compared simultaneously by all tuples of the other relation. The matches constitute the result of intersection. Thus intersection can be obtained in $2(K + |A \cap B|)$ steps and union in $2|A \cup B|$ steps.

5. Implementation Issues and Major Problems

5.1. Chip Layouts

First we discuss how we can place the different types of nodes on chips. The two "mirrored" binary trees of Figure 12 (a) can first be "unmirrored" to the one as shown in Figure 12 (b), which is then laid out as in Figure 12 (c). This space-economical layout has first been suggested by Mead and Rem [19]. In this layout, the amount of space is proportional to the number of nodes on a chip. Using the layout as in Figure 12 (c), we place the □-nodes on as few number of chips as allowed by the achievable circuit density, and then combine these chips together with chips containing only O-nodes and Δ-nodes.

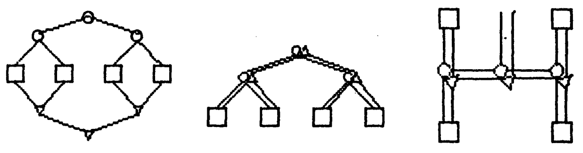


Figure 12: Chip layout.

5.2. Loading the Tree Machine

Loading the tree machine constitutes the bottleneck of the system and is the major problem to be solved. One solution is to provide the capability of reading multiple tracks to load subtrees in parallel, bypassing the input root node of the tree. If such a solution is used, then the proposed space allocation scheme will have to be modified accordingly. The solution also calls for a considerable amount of communication paths from the tree to the outside world. If many chips are needed to implement a tree machine, the required amount of pins for parallel loading will be readily available. If only a few chips are needed, then this solution cannot be used. In this case, we can perhaps construct several tree machines and overlap the I/O and computation proper. The number of such devices depends on the desired response time as well as the various timing characteristics.

5.3. Number of Chips Required

Let us estimate the number of chips to implement a tree machine with a capacity of holding a cylinder of data. This is motivated by the DBC design which assumes that a cylinder of data can be searched in a complete revolution [1]. We choose arbitrarily a tuple size of 64 bytes. With a cylinder capacity of 500,000 bytes, the tree machine will have 500,000/64, or roughly 8,000 □-nodes. We have designed a prototype chip implementing a simpler version of the tree machine where only insertion, deletion and membership testing have been considered [23]. Using that experience, we estimate that, for the complete version, about 8 □-nodes can be put on one chip. Therefore 8,000 □-nodes will require 1000 chips, which is feasible with current technology. With the rapid increase in circuit density, this number will become approximately 60 chips in four years and only 4 chips in about ten years. Provided that the problems which arise with the increased density (such as that of powering) can be solved, this approach seems very promising to implement a large capacity tree machine.

5.4. Problem Partitioning

We have assumed throughout this paper that all the related data can be accommodated in the tree machine. This assumption is certainly not realistic. Problem partitioning for cases in which the problem size exceeds the device capacity should also be studied.

6. Conclusion

We have proposed a design of a high-performance tree-structured machine to handle the basic database operations. The tree structure is very desirable for its logarithmic path from the root to any leaf node. This makes broadcasting of instructions a very convenient and inexpensive operation. Also, being a structure of two "mirrored" complete binary trees, one for input and one for output, the tree machine is especially suitable for pipelining of instructions and data. In the tree machine, data reside in the tree and different operations can be performed without having to move data around. This is important for processing queries which require the execution of a sequence of database operations before the answer is obtained. If all these operations can be performed at one single site, less I/O will be required. Bentley and Kung [3] make an interesting observation about the "computational structure" of the tree machine: it has very small input and output channels, with massive computation going on in between. Although much search or other efforts are needed to process a query, the answers frequently consist of only a few records. The tree machine seems especially adequate for such operations.

The particular design we have proposed here is an attempt to exploit the recent VLSI technology. One peculiar characteristic in this technology is that logic is cheap but communication costly. Also, by replicating one basic cell a large number of times on a chip, design costs are reduced. This is why regularity and locality are such important properties in VLSI design (see [9] for a detailed discussion).

The tree machine possesses precisely these properties. There are only three kinds of basic cells (or nodes), each of which interacting only with a few neighbors in a very regular way. This approach seems especially attractive in the near future when circuit density continues to rise.

Acknowledgment

The author wishes to thank H. T. Kung and Jon Bentley for their encouragement and inspiration provided to this research. Helpful conversations with Izumi Kimura, Phil Lehman, Charles Leiserson, and Clark Thompson are gratefully acknowledged. Thanks are also due to the referees who provided most valuable comments and suggestions.

References

1. J. Banerjee, D. K. Hsiao, and K. Kannan, "DBC - A Database Computer for Very Large Databases", *IEEE Trans. on Computers* 28, 6 (June, 1979), pp. 414-429.

2. J. Banerjee, D. K. Hsiao, and J. Menon, *The Clustering and Security Mechanisms of a Database Computer*, Computer and Information Science Research Center, The Ohio State University (April, 1979), 112 pp.

3. J. L. Bentley, and H. T. Kung, "A Tree Machine for Searching Problems", in *Proc. 1979 International Conference on Parallel Processing*, IEEE (August, 1979).

4. J. L. Bentley, and J. B. Saxe, "Decomposable Searching Problems", *Information Processing Letters* 8, 5 (June, 1979).

5. K. J. Berkling, "A Computing Machine Based on Tree Structures", *IEEE Trans. on Computers* 20, 4 (April, 1971).

6. S. A. Browning, "Computations on a Tree of Processors", in *Proc. of Caltech Conf. on Very Large Scale Integration*, Caltech (January, 1979), pp. 453-478.

7. H. Chang, "On Bubble Memories and Relational Data Base", in *Proc. 4th International Conf. Very Large Data Bases*, West Berlin (1978), pp. 207-229.

8. T. C. Chen, V. W. Lum, and C. Tung, "The Rebound Sorter: An Efficient Sort Engine for Large Files", in *Proc. 4th Int. Conf. on Very Large Data Bases* (1978), pp. 312-318.

9. M. J. Foster, and H. T. Kung, "Design of Special-Purpose VLSI Chips: Examples and Opinions", *Computer* (January, 1980).

10. L. A. Hollaar, "A Specialized Merge Processor for Combining Sorted Lists", *ACM Transactions on Database Systems* 3, 3 (September, 1978).

11. D. K. Hsiao, "Database Computers", in *Advances in Computers*, Academic Press, Vol. 19 (1980).

12. H. T. Kung, and P. L. Lehman, "Systolic (VLSI) Arrays for Relational Database Operations", *ACM SIGMOD International Conference on Management of Data* (1980).

13. G. G. Langdon Jr., "A Note on Associative Processors for Database Management", *ACM Trans. Database Systems* (June, 1978).

14. H. O. Leilich, G. Stiege, and H. C. Zeidler, "A Search Processor for Data Base Management Systems", in *Proc. 4th Conf. on Very Large Data Bases* (September, 1978), pp. 280-287.

15. C. E. Leiserson, "Systolic Priority Queues", in *Proc. of the Caltech Conf. on Very Large Scale Integration*, Caltech (January, 1979), pp. 199-214.

16. G. A. Mago, *A Network of Microprocessors to Execute Reduction Languages*, Department of Computer Science, University of North Carolina (March, 1978), 114 pp.

17. F. J. Maryanski, "Backend Database Systems", *ACM Computing Surveys* 12, 1 (March, 1980), pp. 3-25.

18. C. Mead, and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley Publishing Company (1980).

19. C. Mead, and M. Rem, "Cost and Performance of VLSI Computing Structures", *IEEE Journal of Solid State Circuits SC-14*, 2 (April, 1979).

20. S. A. Schuster, H. B. Nguyen, E. A. Ozkarahan, and K. C. Smith, "RAP 2 - An Associative Processor for Databases and Its Applications", *IEEE Trans. on Computers* 28, 6 (June, 1979), pp. 446-458.

21. C. H. Sequin, A. M. Despain, and D. A. Patterson, "Communication in X-tree, a Modular Multiprocessor System", in *ACM 78 Proceedings*.

22. S. W. Song, "Database Machines: a Taxonomy and Appraisal", in preparation.

23. S. W. Song, "A Database Machine with Novel Space Allocation Algorithms", in *Proceedings of the MPC79 Multi-University Multiproject Chip Set Project*, by A. Bell, L. Conway, R. Lyon, and M. Newell (editors), Xerox PARC/SSL (to appear).

24. S. Y. W. Su, "Cellular-Logic Devices: Concepts and Applications", *Computer* (March, 1979), pp. 11-25.

25. S, Y. W. Su, L. H. Nguyen, A. Emam, and G.J. Lipovski, "The Architectural Features and Implementation Techniques of the Multicell CASSM", *IEEE Trans. on Computers* 28, 6 (June, 1979), pp. 430-445.

26. W. Wilner, *Recursive Machines*, Xerox PARC SSL Internal Memorandum (January, 1978), 24 pp.

A STUDY OF THE INTERCONNECTION OF MULTIPLE PROCESSORS IN A DATA BASE ENVIRONMENT

James R. Goodman[†] **and Alvin M. Despain**

Computer Science Department, University of California, Berkeley, CA 94720

ABSTRACT

In the design of very large data base machines, multiple processors can be employed effectively to increase performance. When massive amounts of data must be moved, the topology of the processor interconnections is important. To determine an appropriate interconnection scheme, a simple model of a very common but difficult data base operation is used to determine an interconnection scheme. This is the elimination of duplicate information in a collection of data elements. In particular, five methods are considered which can perform the elimination of duplicates. These methods and their corresponding interconnection topologies are analyzed and compared to help determine a suitable multiprocessor topology and computer architecture for a data base machine. A hybrid architecture is shown to be near-optimal

Key words: database machine; computer network; multiprocessor; computer architecture; relational data base; computer system analysis.

1. INTRODUCTION

Data base management systems perform well today largely through the use of sophisticated software structures to enhance the retrieval of the desired information. The price paid in terms of software complexity and storage overhead is quite severe, and may become excessive for very large data bases. Unfortunately, there will inevitably be some queries of considerable importance which cannot be performed in an acceptable length of time because the proper structures do not exist in the data organization, even though the information is present. One approach is to build a special purpose machine for data base management systems (DBMS).

A great deal of interest has been generated recently over the concept of a multiprocessor data base machine and the topology that it should take. Since one of the most significant ways that hardware can be used to attack a problem is through parallelism, it is not surprising that virtually all[1]-[12] of the recent proposals have utilized this idea to a significant degree. One might ask the following questions:

(1) What tasks performed by a DBMS can be improved by the use of parallel processors?

(2) How can multiple processors best be organized to optimize the execution of those tasks?

[†] Present address: Computer Sciences Department, University of Wisconsin, Madison WI 53706.

It is instructive to consider the following problem: Given the telephone directory as a data base, determine the name of the occupant at a known address. Although the information is clearly present in the directory it cannot be readily retrieved. If this question were to be asked frequently, the problem could be solved by producing an index of addresses. It is not feasible to build such so-called secondary indices for all possible queries, since those queries can become arbitrarily complicated. The only general solution is to be prepared to search the entire data base.

A number of generic operations on data bases can be identified which can provide a great deal of insight into the requirements for efficient data base support. In the relational model, for example, these operations might include restriction, projection, and equi-join, among others. In the implementation of these operations, some more fundamental operations occur repeatedly. One of the most expensive of these is the elimination of duplicates in a relation, performed every time a projection occurs. An equivalent operation is performed in any data base, however, and this procedure is certainly not unique to the relational model. The elimination of duplicates is so expensive that the user is often given the opportunity to tell the system when it need not be done. We have chosen to study this particular operation at some length.

2. THE ELIMINATION OF DUPLICATES

During an exhaustive search as well as during general set-oriented operations, a DBMS collapses the data required by eliminating certain fields. Further reduction of the data is then possible because the remaining fields contain many duplicate entries. In the handling of a complex query this reduction in data is crucial and must be performed several times.

In terms of our telephone directory model we can consider the following operation:

List all of the street names present in the directory.

The stripping away of the names, street numbers, and telephone numbers will leave the desired information, but in a highly redundant form, since many people live on the same street. The result may be thought of as a list of numbers and the problem is to eliminate multiple occurrences of a number.

We shall assume that the number of elements N making up the list is large – too large to be reasonably supported by a single processor. Duplicates can be eliminated by exhaustive comparison or by sorting, or by some combination of the two. The advantage of the sorting approach is this: direct comparison of all pairs of elements in a list of length N requires $O(N^2)$ comparisons to

CH1569-3/80/0000-0269$00.75 © 1980 IEEE

eliminate all duplicates. Sorting algorithms, on the other hand, may require only[1] $O(N \log N)$ comparisons worst case and may, depending on the order in the list, require a substantially smaller number than that. Algorithms exist, in fact, for sorting in $O(N)$ operations[13]. However, sorting implies the moving of a large amount of data, more or less randomly. This is awkward, particularly if the elements to be sorted are in different processors. Thus a tradeoff exists between the movement of data and the number of comparisons, depending on the approach chosen.

We shall compare the methods to follow in two ways: the cost of computation C and the cost of communication M. The computation will be measured crudely by estimating the number of comparisons required. The interprocessor movement of data is measured by the sum of the transmission of every element across every interprocessor link. If an element must traverse three such links, then three message element links must be counted. Computation and communication costs will be determined both for the total requirements and for the busiest processor and link, respectively. This allows for analysis based on either through-put requirements or response-time requirements, *i.e.* bandwidth or latency.

3. PARALLEL METHODS

Assume that a number of identical computers (P) are connected so that they can communicate in a fairly intimate way among themselves via messages. (P is assumed to be a power of 2, except where noted). Suppose that a list of numbers of total length N, is segmented into P lists of equal length $L = N/P$ and distributed over the P processors. In the general case, a wide range of possible outcomes could result from the elimination of duplicate elements, depending on the degree of redundancy in the data base. In an attempt to establish bounds on the size of the task, we shall consider the two extreme cases:

(1) *All elements are identical.* For this case let $C1$ be the total number of comparisons done in all processors and $M1$ be the total number of message element links. Also, define $C1_{\max}$, the maximum number of comparisons performed in any one node, and $M1_{\max}$, the maximum number of messages transmitted over any one link.

(2) *All elements are unique,* i.e., *there are no duplicates.* In this case, let CN be the total number of comparisons done in all processors and let MN be the total number of message element links. Also, define $CN_{\max}$, the maximum number of comparisons performed in any one node, and $MN_{\max}$, the maximum number of messages transmitted over any one link.

Identifying the duplicates in two ordered lists is equivalent in complexity to merging the lists. For all methods presented, it is assumed that each processor first sorts and eliminates its own duplicates. Since this requirement is the same for all methods, it has been ignored. Thus, to merge two ordered lists of lengths L_1 and L_2 requires $L_1 + L_2$ comparisons[2]. How can the duplicates be eliminated?

[1]Throughout this paper, unless otherwise specified, log means $\log_2$.

[2]Knuth[14] shows that the minimum possible for the worst case is actually $L_1 + L_2 - 1$ if L_1 and L_2 are approximately equal. We shall ignore the constant term and assume that, in general, merging y lists, each of length L, can be performed in $yL \log y$ comparisons, recognizing that this simplification results in the assumption that merging two lists of length one requires two compares.

Consider how it might be done by first eliminating duplicates within a list, then by comparing every pair of lists. Somehow the lists must be transmitted in a regular way so that all lists are compared against each other. However, a method must avoid the problem of mutual destruction of all duplicates. The following method does this by assigning priorities to the processors:

METHOD 1: BROADCAST / BUS ORGANIZATION

The P processors are ordered and connected to a common bus. Each processor eliminates the duplicates within its own list. The first processor broadcasts its condensed list in sorted order to the remaining processors quits. Each processor which receives the list compares it against its own elements and eliminates all elements that match an element of the broadcast list. The remaining processors sequentially broadcast their condensed lists and quit. When all processors but the last are finished, the duplicates have been eliminated.

This algorithm has the desirable property that the message size shrinks as the duplicates are eliminated. Thus the total length of the message units sent is the length of the list of all elements with duplicates eliminated less the number of unique elements in the last processor. Obviously, this is the least possible communication cost.

It solves the problem of saving exactly one copy of each element by serializing the broadcasts. Note that these broadcasts cannot be done in parallel, even if multiple busses are available. As a result, the parallelism is limited. On average, no more than half of the processors are busy. Since each list is sorted before communication begins, the removal of the duplicates can be done in one pass for each broadcast. If there are no duplicates actually present, then the total number of comparisons CN is the sum of the number of comparisons

$$CN = 2L(P-1) + 2L(P-2) + \cdots + 2L(1)$$

$$= 2L\frac{P(P-1)}{2}$$

$$= N(P-1).$$

$$CN_{\max} = 2L(P-1) = 2(N-L).$$

Also

$$MN = (P-1)L = N-L,$$

$$MN_{\max} = MN = N-L.$$

Here the broadcast of a message to many other processors is counted as only one message sent. If there is only one unique element, only that one element is sent, and each of the other $P-1$ processors compare it and eliminate their copy:

$$C1 = 2(P-1), \quad C1_{\max} = 2,$$

and

$$M1 = 1, \quad M1_{\max} = 1.$$

Of course, there will be some messages necessary to notify other processors that no more elements are to be sent, but this is considered overhead which, in general, is small enough to ignore.

Another sort of priority can be introduced by allowing an additional processor to do the comparison of the two or more lists and produce the result:

METHOD 2: TREE ORGANIZATION

Each processor, after eliminating its own duplicates, sends its list to its parent in sorted order, which merges the lists it receives, eliminating the duplicates, and sends the result on to its parent. This continues until the final list is formed at the root of the tree.

In this structure there are actually $(yP-1)/(y-1)$ processors connected as a tree, where y is the branching factor of the tree and P is a power of y. P processors are leaves, $(P-1)/(y-1)$ are non-leaves. The P processors at the leaves have direct access to the data. This method requires more processors, nearly twice as many as in the previous case. It is very effective if the number of duplicates is large. However, if few duplicates exist, the length of the list will increase, by nearly a factor of y at each stage, increasing both the computation and the worst case message traffic with each level up the tree. Thus each succeeding step uses only $1/y$ as many processors, each of which must do y times as much computation. For this case we calculate CN as follows:
There are $\log_y P$ levels (counting the root or the leaves, but not both). There are $y(P-1)/(y-1)$ links, one above every node except the root. Numbering the levels in ascending order starting with the leaves as 0,

level 1 contains P/y processors, each merging y lists of length L,

level 2 contains P/y^2 processors, each merging y lists of length yL,

. . .

level j contains P/y^j processors, each merging y lists of length $y^{j-1}L$.

Assuming that y lists of length L require $yL\log y$ comparisons[3], we get for CN

$$\frac{P}{y}yL\log y + \frac{P}{y^2}y^2L\log y + \cdots + \frac{P}{y^{\log_y P}}y^{\log_y P}L\log y$$

or

$$CN = P\cdot L\cdot(\log_2 y)(\log_y P) = N\log P.$$

$CN_{\max}$ is computed for the top node, merging y lists of length N/y. Again assuming that y lists of length L require $yL\log y$ comparisons,

$$CN_{\max} = y\frac{N}{y}\log y = N\log y.$$

The calculation of MN follows from the observation that each element goes from a leaf node to the root, i i.e., through $\log_y P$ links. Therefore,

$$MN = N\log_y P = \frac{N}{\log y}\log P.$$

Each of the top level links carries N/y elements, i.e.,

$$MN_{\max} = \frac{N}{y}.$$

This difficulty suggests that the upper nodes might require greater power and larger memory. If all elements are identical, no such congestion occurs. Each non-leaf node receives y lists of length 1. If we assume that merging y lists of length 1 requires $y\log y$ operations, then

$$C1 = \frac{P-1}{y-1}y\log y.$$

Since each non-leaf node does the same number of operations,

$$C1_{\max} = \frac{C1}{number\ of\ non-leaf\ nodes} = y\log y.$$

Since exactly one element passes through each link,

$$M1 = \frac{y}{y-1}(P-1),$$

and

$$M1_{\max} = 1.$$

Consider the binary tree case ($y = 2$). Since the lists were sorted before being sent to a parent all that is required of the parent is a merge of ordered lists, a procedure that increases only linearly with the length of the list. Now suppose that instead of sending the list to a common parent, the two processors divide their elements into two lists in a commonly agreed way and exchange one of them. This leads to the first algorithm utilizing a global sorting:

METHOD 3: BINARY MERGE / n-CUBE

P processors are numbered in binary from left to right, starting with 0. Each processor eliminates its duplicates, leaving them in sorted order. The range of values of the sort field is partitioned in a universally agreed-upon way, (the obvious way, for example, is to use the most significant bit of each element), and each processor breaks its list into two parts. It then sends one of the two lists to the processor having the same address except for the most significant bit as follows: If the most significant bit of the address of the sending processor is a 1, it sends the first list. Otherwise, it sends the second list.

After merging the received list with the retained one and eliminating duplicates, each processor repeats the process, but with the following modification:

Each partition of the range of the sort field is further sub-divided into two parts. If the straight-forward way is used, then on step j, the jth most significant bit of the address is used to determine which list to send and to whom it will be sent.

This process is repeated $n = \log P$ times, after which the range is partitioned into P parts, and one processor contains all the values for exactly one partition. If the obvious partition was used, all numbers are sorted into the proper list according to their n most significant bits.

The links required between the processors form the n-dimensional structure known as the n-cube[15] and sometimes called hypercube.

This procedure requires only $\log P$ steps and does not get more complex on subsequent steps – in fact it gets shorter with the elimination of the redundant elements. After each of the $\log P$ exchanges, all P processors merge two lists of length $L/2$. Therefore,

[3]Strictly speaking, this is an equality only if y is a power of 2. Comparison is inherently a binary operation, and a y-way merge can be accomplished with only about $\log y$ comparisons per element using a selection tree when y is a power of 2.

$$CN = (\log P)P\cdot 2\left[\frac{L}{2}\right]\log 2 = N\log P.$$

Symmetry arguments guarantee that $CN_{\max}$ and $MN_{\max}$ are just CN/P and MN/P respectively. Assuming that on each move, half the elements are moved[4],

$$MN = \log P\,\frac{N}{2} = \frac{N\log P}{2}.$$

For the unique case, after exchange j, $P/2^j$ nodes have a list of length 1 while the remainder have the empty list.

$$C1 = 2(1)\log 2\sum_{j=1}^{\log P}\frac{P}{2^j} = 2(P-1),$$

$$C1_{\max} = 2(1)\log 2(\log P) = 2\log P.$$

During exchange j, $P/2^j$ elements are sent:

$$M1 = \sum_{j=1}^{\log P}\frac{P}{2^j} = P-1$$

and $M1_{\max} = 1$.

There is nothing magic about the binary process, however. One could use a y-way sort and divide the elements into y lists, sending $y-1$ off at each step. This would require fewer steps, since

$$\log_y P < \log_2 P$$

for all values of $y > 2, P > 1$. Carrying this idea to its extreme, we could work in base P, in which case only one swap would occur. This results in the following method:

METHOD 4: P MERGE

Each processor orders its own list and, after eliminating its own duplicates, partitions the list into P separate lists in a consistent way for all processors. Numbering these sublists from lowest segment to highest, the jth segment is sent to the jth processor. Each processor retains only that sublist which it would send to itself, and merges it and the $P-1$ incoming sublists as they arrive.

This method again is near optimal in terms of the transmission of information, at least for the case where there are few duplicates. With no duplicates, each processor merges P lists, each of length L/P:

$$CN = P\left[P\frac{L}{P}\log P\right] = N\log P.$$

$$CN_{\max} = \frac{CN}{P} = \log P.$$

Each processor sends $L - L/P$ elements:[5]

$$MN = P\left[L - \frac{L}{P}\right] = N - L.$$

$$MN_{\max} = \frac{MN}{number\ of\ links} = \frac{N-L}{P(P-1)/2} = 2\frac{L}{P}.$$

For the single unique element case, only one processor receives anything: $P-1$ lists of length 1.

$$C1 = P(1)\log P = P\log P,\quad C1_{\max} = C1 = P\log P,$$

$$M1 = P-1,\quad M1_{\max} = 1.$$

Another extension of Method 2 is two build a network which contains the interconnections for one dimension of the n-cube and has the capability to move the data among processors so that each exchange can be accomplished with immediate neighbors. It has been shown [16] that both the shuffling of the data and the exchange can be effected in paths through only one link each for the network known as the perfect shuffle. This leads to our last method.

METHOD 5: BINARY MERGE / PERFECT SHUFFLE

P processors are numbered in binary from left to right, starting with 0. Each processor has a link to one neighbor whose address is the same except for the least significant bit. This link is used to implement the exchange. In addition, each processor has two other links to the two processors having the same address but shifted (end-around) one position. Each of these links is used once for each shuffle. Each processor eliminates its duplicates, leaving them in sorted order. Using the agreed test, (again perhaps the most significant bit of each element), each processor partitions its list into two smaller ones. It then exchanges one list with its neighbor.

After merging the received list with the retained one and eliminating duplicates, a shuffle is performed, i.e., each processor sends its entire list over the link to the processor with the same address shifted one position, say, left.

This process is repeated $\log P - 1$ times, after which the range is partitioned into P parts and each processor has all the values for exactly one partition.

The computation involved here is the same as for the n-cube structure, so CN and $CN_{\max}$ are precisely the same as that case. Assuming again that on each move, half the elements are moved, the communication involved in Method 3 is again required, but additional communication is incurred because of the shuffles. Each shuffle involves sending all surviving elements through one link, and since there are $\log P - 1$ shuffles,

$$MN = \frac{N\log P}{2} + N(\log P - 1) = 3\frac{N\log P}{2} - N.$$

Since more traffic goes over the shuffle links, the traffic on the busiest link is

$$MN_{\max} = \frac{N(\log P - 1)}{P} = L(\log P - 1).$$

For the unique case, again $C1$ and $C1_{\max}$ are the same as for Method 3. Again an additional communication cost is incurred because of the shuffle. During shuffle j, $j = 1,2,3,\cdots,(\log P - 1)$, $P/2^j$ nodes transmit a list of length 1, the remainder transmitting the empty list. Thus the additional communication cost for the shuffles is

$$\sum_{j=1}^{(\log P-1)}\frac{P}{2^j} = P-2,$$

so the total communication cost is

$$M1 = P - 1 + P - 2 = 2P - 3.$$

The busiest link is the exchange link of one particular processor which carries the unique element on every exchange. Thus,

$$M1_{\max} = \log P.$$

[4] In the worst case, when all elements are moved each time, $MN = N\log P$.

[5] $MN = N$, worst case.

4. COMPARISON OF THE METHODS

Table 1 compares the five methods under the assumption that no duplicate data exists. Table 2 compares the five methods for the model where all elements are identical. The parameters have been normalized for the case of all unique elements by dividing by L, the length of the list in each processor. For purposes of comparison, the following assumptions have been made:

(1) Order is initially totally random, but the elements are evenly distributed among the processors.

(2) The numbers are scattered randomly, i.e. evenly, over their possible values.

The first assumption seems reasonable, though presumably it corresponds to some sort of worst case. The second assumption, however, requires some justification. Normally one would expect to find severe clustering of the numbers resulting from the fact that they are normally derived from natural language or other organized sets of data. They can be randomized, however, by hashing the sort field. The sort order is changed, making the end result of little use as a sorted list. This is not terribly important in many cases, however, since the elimination of duplicates is so often an intermediate result and its ordering is not useful anyway.

A more serious problem is that if the hashing function fails to randomize the data sufficiently, some nodes may receive very large lists. This would imply that each processor must have enough memory to hold the entire list, violating our assumptions. This problem can be resolved, however, by aborting an operation as soon as an overflow occurs, and substituting a more appropriate hashing function.

The comparison of message traffic among processors is not straightforward if the processors in the different cases have different kinds or numbers of ports. One might reasonably expect that processors with more ports or faster ports would be more expensive, so that it is also only fair to expect more performance from them. In the above cases the processors vary widely in their I/O capability, from a binary tree or the perfect shuffle, which need only three ports per processor, to the complete interconnection, which requires as many ports on each processor as there are processors, less one. The bus structure is even harder to compare, since although only one port is specified, it nevertheless is obviously much different than the port required by the other cases.

Despain[17] has shown for a single chip computer that power considerations limit the total I/O bandwidth of the processors. Thus if the total bandwidth available is B bits/second, we can assume that each of K identical ports can transmit a maximum of B/K bits per second. They have also shown for the case where Q processors share a bus that a processor using the bus can achieve a bandwidth of only[6] B/(Q-1) bits/second. A further result is that reduced bandwidth is equivalent to an increase in the average path length for a message, *i.e.*, for a given set of message interchanges there exists an average path length A, such that

$$A \cdot B_E = B,$$

where B_E is the effective bandwidth through a port and B is the total bandwidth available to a processor.

[6]In the special case where the processors broadcast sequentially. In the general case where all processors are vying for the bus, it is much worse, *i.e.* $B/(Q-1)^2$.

If we assume that we can obtain a structure with equivalent performance by reducing the number of ports and increasing the number of intermediate nodes traversed, we can define equivalence among the various processors by multiplying the message traffic by the average path length. Thus we define

$$\overline{MN} = A \cdot MN, \quad \overline{MN_{\max}} = A \cdot MN_{\max},$$

$$\overline{M1} = A \cdot M1A, \quad \overline{M1_{\max}} = A \cdot M1_{\max}.$$

Tables 1 and 2 show values for the effective path length $A$and Under these assumptions, the tree (method 2) and the perfect shuffle (method 5) have the least total message traffic, regardless of the duplication factor, though the binary merge with the n-cube (method 3) has less message traffic if P is quite small. Also, the optimal value for y is 4, although the differences are small for values of 2 to 8. On the other hand, when the duplication is low, the tree exhibits congestion near the root, and all methods but the bus are superior to the tree with respect to the busiest link, increasingly so with larger values of P. When the duplication is high, however, the binary tree is exceedingly effective, with the busiest link not affected even with increases in P. Clearly none of these structures is best over our range of consideration.

Some of the methods are asynchronous. The P - 1 messages that each processor sends in method 4 need not be sent simultaneously. Each processor can begin processing the second phase of the P-merge as soon as one message has arrived. Thus communication and processing can be overlapped.

The binary merge (method 3) likewise can proceed asynchronously, with each node having a list of other processors with which it must communicate sequentially. Thus, either of these methods can be implemented on a general computer network where all nodes can communicate with all others. Both require many messages to many different nodes, so it should be noted that efficient communications are vital in the elimination of duplicates using a sorting scheme.

It is interesting to observe in method 2 that if the initial elimination of duplicates results in the elements being sorted, then the processors above the bottom level can proceed asynchronously in a pipeline fashion. Each processor may begin processing as soon as it has received one element from each of its children. After selecting the lowest value of those received, it can immediately send this element on to its parent, and remove it from its own list. Thus it is not at any time required to store the complete lists, which may be growing quite large. Of course the node at the top must do something with the resulting list, and it might turn it around and send it down the tree, where it can be sorted on the way down, thus preserving a useful sort order. Thus all the non-leaf nodes can be working simultaneously, resulting in a higher degree of parallelism than might otherwise be expected.

The model of method 2 uses up to twice as many processors as the other models. The difference in performance, however, is much greater than a factor of two. The amount of computation is no more than for any other method, so the processors on the average do only half as much work. The total message traffic, on the other hand, is much less than for any other method, for large values of P. The important consideration here is how rapidly the requirements grow as the number of processors grows, and in this respect, a mere factor of two is quite unimportant.

	Method				
	1	2	3	4	5
No. of Proc.	P	$\frac{yP-1}{y-1}$	P	P	P
No. of links	1	$\frac{y(P-1)}{y-1}$	$\frac{P\log P}{2}$	$\frac{P(P-1)}{2}$	$\frac{3P}{2}$
CN/L	$P(P-1)$	$P\log P$	$P\log P$	$P\log P$	$P\log P$
MN/L	$P-1$	$\frac{P}{\log y}\log P$	$\frac{P\log P}{2}$	$P-1$	$\frac{3P\log P}{2}-P$
CN_{max}/L	$2(P-1)$	$P\log y$	$\log P$	$\log P$	$\log P$
MN_{max}/L	$P-1$	$\frac{P}{y}$	1	$\frac{2}{P}$	$\log P-1$
A	$P-1$	$y+1$	$\log P$	$P-1$	3
$\overline{MN}/L$	$(P-1)^2$	$\frac{(y+1)}{\log y}P\log P$	$\frac{P(\log P)^2}{2}$	$(P-1)^2$	$\frac{9P\log P}{2}-3P$
$\overline{MN_{max}}/L$	$(P-1)^2$	$\frac{y+1}{y}P$	$\log P$	$2\frac{(P-1)}{P}$	$3(\log P-1)$

Table 1. Comparison of five methods for eliminating duplicates assuming that no duplicates exist. Method 1: Sequential broadcast. Method 2: y-branch tree. Method 3: Binary merge. Method 4: P merge. Method 5: Perfect shuffle. *CN* is the total number of comparisons done in all processors. *MN* is the total number of message element links. CN_{max} is the maximum number of comparisons done in one processor. MN_{max} is the maximum number of message elements passing through any one node. $\overline{MN} = MN \cdot A$ is the total message traffic adjusted to compare processors with different numbers of I/O ports. $\overline{MN_{max}} = MN_{max} \cdot A$ is the normalized measure of busiest link traffic. *A* is the normalization factor for the variable number of ports required.

	Method				
	1	2	3	4	5
No. of Proc.	P	$\frac{yP-1}{y-1}$	P	P	P
No. of links	1	$\frac{y(P-1)}{y-1}$	$\frac{P\log P}{2}$	$\frac{P(P-1)}{2}$	$\frac{3}{2}P$
$C1$	$2(P-1)$	$\frac{y\log y}{y-1}(P-1)$	$2(P-1)$	$P\log P$	$2(P-1)$
$M1$	1	$\frac{y}{y-1}(P-1)$	$P-1$	$P-1$	$2P-3$
$C1_{max}$	2	$y\log y$	$2\log P$	$P\log P$	$2\log P$
$M1_{max}$	1	1	1	1	$\log P$
A	$P-1$	$y+1$	$\log P$	$P-1$	3
$\overline{M1}$	$P-1$	$\frac{y(y+1)}{y-1}(P-1)$	$(P-1)\log P$	$(P-1)^2$	$6P-9$
$\overline{M1_{max}}$	$P-1$	$y+1$	$\log P$	$P-1$	$3\log P$

Table 2. Comparison of five methods for eliminating duplicates assuming that all elements are identical. Method 1: Sequential broadcast. Method 2: Tree. Method 3: Binary merge. Method 4: P merge. Method 5: Perfect Shuffle. $C1$ is the total number of comparisons done in all processors. $M1$ is the total number of message element links. A is the normalization factor for the variable number of ports required. $\overline{M1} = M1 \cdot A$ is the total message traffic adjusted to compare processors with different numbers of I/O ports. $\overline{M1_{max}} = M1_{max} \cdot A$ is the normalized measure of worst case link traffic.

Methods 3 and 4 require substantially more data paths than any of the other methods. Clearly the P-merge is not feasible for large P if $P(P-1)$ dedicated links are required. Even the $PL(P/2)$ links required by the binary merge are hard to justify for large values of P. This would mean $\log P$ links per node if dedicated links were used.

A more serious problem is the lack of expandability imposed by these structures. A processor may have only a fixed number of ports, particularly if it is a single integrated circuit. Methods 3 and 4 require an increase in the number of ports as the number of processors grows, so that if room is left for expansion, then some of the available ports are unused, wasting available resources.

The perfect shuffle seems to have many of the properties needed here, though it is markedly inferior to the binary tree in the case of high duplication. It also has a large enough linear coefficient for MN that its superiority occurs only for large values of P. But it poses some unfortunate problems as well. It certainly cannot be gracefully expanded, since it requires a power of two processors. Furthermore, the routing of messages in such a structure is difficulty because of its lack of symmetry.

On the other hand, the sequential broadcast method takes substantially longer to execute than the other methods. Also, if the duplication factor is low, it requires far more comparisons than any other method and much more communication bandwidth than any method except the complete interconnection.

It is clear that the bus is inferior to the other methods. However, the others all have shortcomings which are extremely serious. The question then arises – is it possible to construct a network on which several of the methods can be implemented so that the best method may be employed in a given situation?

Assuming that each processor in a structure has the same number of ports, a significant variation among these structures is the portion of ports actually used. The complete interconnection and the n-cube algorithms, for example, use every port. But the binary tree uses less than two-thirds of the ports it has, since each leaf node has two unused ports. It has been suggested [18] that this is desirable to allow a convenient placement of I/O devices, a point that all structures must address somehow. Thus a fairer comparison might require that each structure have as many unused ports as it has processors. For methods 3, 4, and 5 this would be approximately equivalent to increasing the value of A by 1.

An alternative approach, and one taken here, is to connect the unused ports of such a structure in some regular way. One possibility for the binary tree is to connect the leaves to form the perfect shuffle interconnection (Fig. 1). The exchange can now be accomplished by messages exchanged through the common parent. Unfortunately, this doubles the traffic during the shuffle, which is already the dominant traffic for method 5. A better possibility exists.

5. X-TREE

A topology recently proposed in connection with X-TREE[18],[19] can implement any of the algorithms. The structure, called *hypertree*, is the binary tree topology, but with each node having one extra link connecting it in a regular way to another node at the same level (Fig. 2). The structure is particularly well-suited for communications among leaf nodes which are nearest neighbors in **the n-cube. Since the structure is a binary tree, obviously method 2 (binary tree) can be implemented directly on the structure. In addition, method 3 (n-cube) can be implemented by using the leaf processors, passing messages through intermediate nodes where necessary. Furthermore, the structure has been shown to be well-suited for communication among all leaf nodes, so that method 4 could also be implemented conveniently. Method 5, the perfect shuffle algorithm, could also be implemented, though nothing is gained by the shuffle, so it is essentially the same as method 3. However, the extra ports of the leaf nodes can be connected in a perfect shuffle so that the horizontal links can be used for the exchange (Fig. 3). With this addition, the structure can perform the binary merge as well as the perfect shuffle network except that the value of A is 4 instead of 3.**

Tables 3 and 4 show the values for this model assuming the structure is used to implement methods 2, 3, 4, and 5. The computations, of course, do not change, being determined by the method and the corresponding logical structure. Degradation occurs for the binary tree structure because the multiplication factor A, is increased by one to accommodate the additional link required for the hypertree connection.

Method 3 is implemented using the extra links. It has been shown[19] that for communication between any pair of leaf nodes, an optimal path exists which goes no more than half way up the tree. This guarantees that the bottleneck which would occur in the simple binary tree if few duplicates are present, will not occur, or at least will be much less serious, since a factor of $\sqrt{P}$ more links are available to handle the traffic over the most heavily used path.

The best method to use varies greatly, depending on the amount of duplication in the list, the number of processors, and the relative importance of total traffic versus busiest link traffic.

For the high duplication case, the simple binary tree algorithm is always the best for the worst case link traffic, though it is slightly inferior to the binary merge (n-cube) algorithm in total traffic. Note also that these two methods have the lowest computational requirements as well, under these conditions.

For the case of low duplication, the results are not so clear-cut. Up to about 128 leaf nodes, methods 3 and 4 are best, with method 4 slightly preferable if total traffic is the consideration, and method 3 creating somewhat less total message traffic. Method 4 is generally superior at 128 nodes.

Above 128 nodes, method 5, the perfect shuffle, becomes the best method because of its attractive balanced link traffic. Method 2, the binary tree algorithm, has slightly less total traffic, but must be rejected because of the excessive bottleneck occurring at the root, both in link traffic and in computation.

6. CONCLUSIONS

The proposed structure is able to implement the best algorithm for a given situation. Under our assumptions, performance is nearly equal, and in some cases superior, to the structure for which the algorithm was originally proposed. The total message traffic, MNp/L is actually improved, approximately by a factor of $P/\log P$ for the algorithm using the complete interconnection (method 4), though the worst case link traffic for the same model is increased by a factor of $\sqrt{P}$ for the case of no duplicates. Since it is the method of choice only for

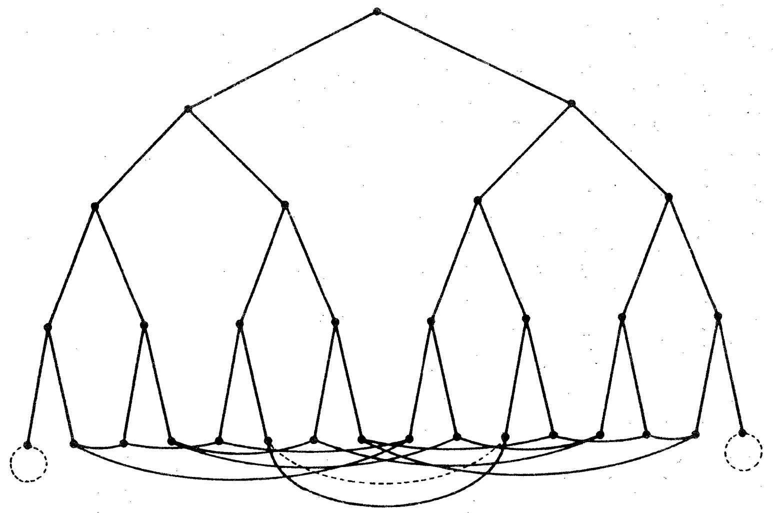


Figure 1. Perfect shuffle interconnection superimposed on the leaves of the binary tree.

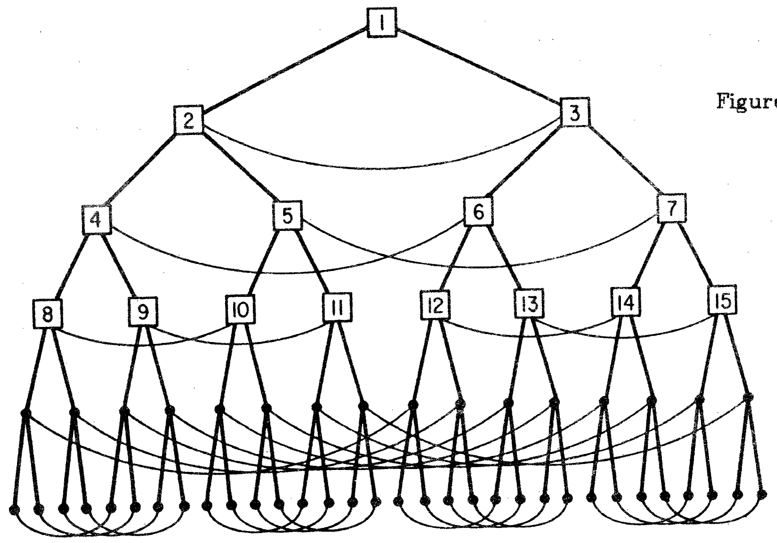


Figure 2. Interconnection of Hypertree I.

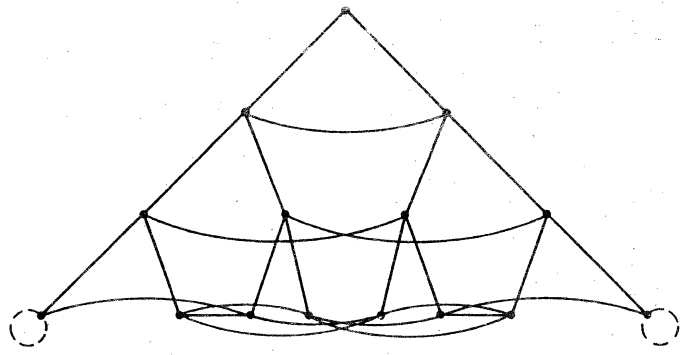


Figure 3. Perfect shuffle interconnection superimposed on the leaves of hypertree. Bottom level hypertree links define exchange pairs, so the leaf nodes are numbered, from left to right: 0, 2, 1, 3, 4, 6, 5, 7.

	Method			
	2	3	4	5
CN / L	$P \log P$	$P \log P$	$P \log P$	$P \log P$
MN / L	$P \log P$	$\frac{P \log P}{4}(\log P + 1)$	$\frac{4}{3} - \frac{17}{12}P + \frac{5}{4}P \log P$†	$\frac{3P \log P}{2} - P$
$CN_{\max} / L$	P	$\log P$	$\log P$	$\log P$
$MN_{\max} / L$	$\frac{P}{2}$	$\frac{\sqrt{2P}}{4}$†	$\frac{P-1}{2\sqrt{P}}$†	$\log P - 1$
$\frac{\overline{MN}}{L}$	$4P \log P$	$P \log P(\log P + 1)$	$\frac{16}{3} - \frac{17}{3}P + 5P \log P$†	$6P \log P - 4P$
$\overline{MN_{\max}} / L$	$2P$	$\sqrt{2P}$†	$\frac{2(P-1)}{\sqrt{P}}$†	$4(\log P - 1)$

Table 3. Implementation of four methods for eliminating duplicates assuming that no duplicates exist and using the "hypertree" structure with the perfect shuffle as shown in Fig. 3. There are $2P - 1$ processors and $4P - 3$ links. The normalization factor, A, is 4. CN is the total number of comparisons done in all processors. MN is the total number of message element links. $CN_{\max}$ is the maximum number of comparisons done in one processor. $MN_{\max}$ is the maximum number of message elements passing through any one node. $\overline{MN} = MNA$ is the total message traffic adjusted to compare processors with different numbers of I/O ports. $\overline{MN_{\max}} = MN_{\max}A$ is the normalized worst case link traffic.

	Method			
	2	3	4	5
$C1$	$2(P - 1)$	$2(P - 1)$	$P \log P$	$2(P - 1)$
$M1$	$2(P - 1)$	$2(P - 1) - \log P$	$\frac{4}{3} - \frac{17}{12}P + \frac{5}{4}P \log P$†	$2P - 3$
$C1_{\max}$	2	$2 \log P$	$P \log P$	$2 \log P$
$M1_{\max}$	1	$(\log P) - 1$	$P - 2$	$\log P$
$\overline{M1}$	$8(P - 1)$	$8(P - 1) - 4 \log P$	$\frac{16}{3} - \frac{17}{3}P + 5P \log P$†	$8P - 12$
$\overline{M1_{\max}}$	4	$4 \log P$	$4(P - 2)$	$4 \log P$

Table 4. Implementation of four methods for eliminating duplicates assuming that all elements are identical and using the "hypertree" structure with perfect shuffle as shown in Fig. 3. There are $2P - 1$ processors and $4P - 3$ links. The normalization factor, A, is 4. $C1$ is the total number of comparisons done in all processors. $M1$ is the total number of message element links. $\overline{M1} = M1A$ is the total message traffic adjusted to compare processors with different numbers of I/O ports. $\overline{M1_{\max}} = M1_{\max}A$ is the normalized worst case link traffic.

† This formula is correct only where P is not a power of 4. If P is a power of 4 the formula is slightly different.

values of $P < 128$ this would seem to be unimportant.

Method 3 shows some degradation in performance. The worst case link traffic is increased, for the case of no duplicates, by a factor of $\sqrt{2P} / \log P$, but for large values of P the perfect shuffle algorithm predominates anyway.

Methods 2 and 5 show only slight, linear degradation due to the increase in the value of A resulting from the unused links for that algorithm. Thus the proposed structure is able to achieve the same order of performance as the best of the methods considered for virtually all circumstances under our assumptions. If other assumptions are made, a different conclusion could also be drawn, as is evident from the results presented in table 1.

The power of the tree structure is clear for the problem of eliminating duplicates. However, the importance of flexibility in choosing the method is apparent. The best structure is clearly one which can handle both extreme cases (and thus presumably the cases in between) reasonably well.

Before an architecture is chosen for a multiprocessor data base machine, other typical data base operations must be analyzed in a similar manner. Nevertheless, the elimination of duplicates is an important data base operation which provides significant insight into the requirements for a data base computer. Other considerations as well undoubtedly will influence the choice. For example, unlike some other models, the tree structure is expandable without a modification to the processor itself, which surely makes it more attractive if it is a single component. Another issue is the fact that adjacent processors may wish to communicate heavily at times. The binary tree, with nodes having fewer ports, each with more bandwidth, is clearly advantageous in this case.

The X-Tree "hypertree" interconnection with the perfect shuffle interconnection among the leaves has been shown to provide an attractive compromise of the models considered. It gives essentially the same performance as the best of the other structures over the range of conditions considered.

7. ACKNOWLEDGEMENTS

The authors are indebted to Clark Thompson for discussions concerning the theoretical advantages of the perfect shuffle approach.

The work has been sponsored in part by the Joint Services Electronics Program, Contract F44620-76-c-0100 and by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 3803, Monitored by Naval Electronic System Command under Contract No. N00039-78-G-0013-0004.

8. REFERENCES

[1] L. D. Healy, K. L. Doty, and G. J. Lipovski, "The architecture of a context addresses segment sequential storage," *Fall Joint Comput. Conf., AFIPS Conf. Proc.* vol. 41. Montvale, N.J.: AFIPS Press, 1972, pp. 691-701.

[2] E. A. Ozkarahan, S. A. Schuster, and K. C Smith, "RAP – Associative processor of database management," *AFIPS Conf. Proc.*, vol. 44, 1975, pp. 379-388.

[3] S. C. Lin, D. C. P. Smith, and J. M. Smith, "The design of a rotating associative memory for relational database applications," *ACM Trans. Database Systems,* vol. 1,pp. 53-75, Mar. 1976.

[4] D. J. DeWitt, "DIRECT - A multiprocessor Organization for supporting relational data base management systems" *Proc. Fifth Annual Symp. Comput. Arch.*, April 1978, pp. 182-189.

[5] Parhami, B., "A highly parallel computing system for information retrieval," *Proceedings of the Fall Joint Computer Conference*, pp. 681-690, 1972.

[6] Coulouris, G. F., Evans, J. M., and Mitchell, R. W., "Towards content-addressing in data bases," *The Computer Journal*, vol. 15, no. 2, February, 1972.

[7] Minsky, N., "Rotating storage devices as partially associative memories," *Proceedings of the Fall Joint Computer Conference*, pp. 587-595, 1972.

[8] Edelberg, M., and Schissler, L. R., "Intelligent Memory," *Proceedings of the National Computer Conference*, 1976, pp. 393-400.

[9] Madnick, S. E., "INFOPLEX – Hierarchical decomposition of a large information management system using a microprocessor complex," *Proceeding of the National Computer Conference*, 1975, pp. 581-586.

[10] Anderson, G. A., and Kain, R. Y., "A content-addressed memory designed for data base applications," *Proceedings of the International Conference on Parallel Processing, Aug. 1976, pp. 191-195.*

[11] Banerjee, J., and Hsiao, D. K. "The Architecture of a Database Computer – Part I: Concepts and Capabilities," Technical Report OSU - CISRC - TR - 76 - 1, The Ohio State University, Columbus, Ohio, Sept. 76.

[12] McGregor, D. R., Thomson, R. G., and Dawson, W. N., "High performance hardware for database systems," *Systems for Large Data Bases*, North Holland Publishing Company, 1976, pp. 103-116.

[13] D. E. Knuth, *The Art of Computer Programming*, vol. 3: *Sorting and Searching*, p. 99-102, Addison-Wesley, 1975.

[14] D. E. Knuth, *op. cite.*, pp. 198-200.

[15] H. J. Segal, "Interconnection networks for SIMD machines," *Computer*, vol. 12, No. 6, June 1979, p. 59.

[16] H. S. Stone, "Parallel Processing with the Perfect Shuffle," *IEEE Transactions on Computing*, Vol. C-20, No. 2, Feb. 1971, pp. 153-161.

[17] A. M. Despain, "A Study of Multicomputer Interconnection Structures," *paper in preparation*, Comput. Sci. Div., Univ. of Calif., Berkeley, Dec. 1979.

[18] A. M. Despain, and D. A. Patterson "X-TREE: A tree structured multi-processor computer architecture", *Proc. Fifth Annual Symp. Comput. Arch.*, April 1978, pp. 144-150.

[19] C. H. Sequin and J. R. Goodman "HYPERTREE: A multiprocessor interconnection topology," submitted to Communications of the ACM, 1979.

ON DATABASE-ORIENTED PERIPHERAL TRANSFORMATION PROCESSOR SYSTEMS

Dieter Schütt
D Dv ST DB3
Siemens AG
8000 München 83, West Germany

Abstract -- The purpose of this paper is to discuss specifications concerning peripheral transformation processor (PTP) systems in a database environment.
A PTP can be seen as a link between a buffer system and secondary storage media for parallel transmission and intermediate manipulation of (blocks of) data.
A PTP mainly consists of a highly modular data manipulation unit and a flexible control.
Built-in fault-tolerant capabilities of a PTP system lead only to slight performance degradation if faulty components are detected. Typical PTP applications include update, simple associative, and cryptographic operations.

3880-like storage control systems provide the capabilities to operate and control several independent data paths between processor(s) and disk storage media. They do not allow, however, intermediate buffering and manipulation of data required for reducing channel and main storage activities in connection with database procedures.

The peripheral transformation processor (PTP) to be discussed is an attempt to incorporate certain functions of dedicated units into a highly paralllel peripheral processor system. In other words, a PTP is a special purpose function architecture mainly for performance enhancement of the corresponding overall system.
A PTP is by no means another database backend machine [1,3,5,6,11,12,13], it could be seen as an evolutionary step towards an intelligent, database-oriented processing system.
A PTP has not been built yet.

Special-purpose data manipulation units described in literature include manipulators for bit-slice functions [7], alignment (scramble/unscramble) networks for multidimensional memory access [9], encryption networks for improved data security [8], associative or quasi-associative search modules [2,10,11,12], and transformation/translation units for database structure operations [1].

A PTP architecture (see Figure 1) must satisfy the following requirements which are vital to I/O-intensive and robust database management.

1. The PTP interfaces to a disk storage system and to a block buffer system, allowing access to different disks and buffers and to cylinder slices of one disk at a time.
   Thus a PTP operates parallel read-out, parallel write-in, and mixed read-out/write-in procedures.
   (Clearly, the attachment of so-called electronic disk devices is desirable for future PTP business).
2. The PTP data manipulation unit (DMU) contains a network of substitution boxes which satisfies the NBS data encryption standard and allows multiple data streams as plaintext input and encrypted/non-encrypted output.
   A substitution box realizes a permutation on $\{0,1\}^k$ as well as the identity (bypass feature).
   A key register is included for modification of the network [8].
3. A more powerful substitution network provides feedback capabilities for multi-phase data manipulation and permits Boolean operations between networks stages (DMU stage logic).
4. The PTP masking facility is a DMU component which allows vertical and horizontal masking of (blocks of) data.
5. The DMU compare logic is restricted to operation on a one-bit-per-word basis (fixed bit location). Thus a PTP does not support sophisticated time-consuming search procedures.
6. Because of the predominant modular structure and the capability to divert multiple data streams a PTP is well-suited for fault-tolerant data processing. Built-in fault-masking and self-repair functions result in a high PTP reliability.
7. Communication between PTP and host processes is governed by a higher-level protocol (probably ISO-level 4) in contrast to protocols which are concerned with standard peripheral/main processor interactions.
   In a host-backend configuration a PTP does not contribute directly to communication between program execution system and database computer system.

Note that requirements 1 to 7 necessitate a flexible PTP microstructure.

The operations listed below demonstrate PTP capabilities concerning support of database management functions:
- copying, i.e. dynamic peripheral duplication of data
- combination of data from different resources (e.g. coincidence or merge of index bit lists)

- composition of (blocks of) data from different resources (e.g. combination of database keys and data items or of primitive search results; simple union operations)
- insertion/deletion of words within blocks of data (e.g. for index list updates)
- differentiation of sets of data (e.g. before/after images, basic/index data)
- projection, i.e. selection of certain data subblocks (domains)

References

[1] J.Banerjee, D.K. Hsiao, and K. Kannan, "DBC - A Database Computer for Very Large Databases", IEEE Transactions on Computers (June, 1979), pp. 414-429

[2] P.B. Berra and E. Oliver, "The Role of Associative Array Processors in Data Base Machine Architecture", Computer, IEEE Magazine (March, 1979) pp. 53-61

[3] R.H. Canaday et al., "A Back-end Computer for Data Base Management", Communications of the ACM (October, 1974), pp. 575-582

[4] E.F. Codd, "A Relational Model of Data for Large Shared Data Banks", Communications of the ACM (June, 1970), pp. 377-387

[5] J. Cullinane et al., Commercial Data Management Processor Study, Cullinane Corporation, Wellesley, Mass., (December, 1975), 76 pp.

[6] D.J. De Witt, "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems", IEEE Transactions on Computers (June, 1979), pp. 395-406

[7] T. Feng, "Data Manipulating Functions in Parallel Processors and Their Implementations", IEEE Transactions on Computers (March, 1974), pp. 309-318

[8] J.B. Kam and G.I. Davida, "Structured Design of Substitution-Permutation Encryption Networks", IEEE Transactions on Computers (October, 1979), pp. 747-753

[9] D. Kuck, "A Survey of Parallel Machine Organization and Programming", Computing Surveys (March, 1977), pp. 29-59

[10] H.O. Leilich, G. Stiege, and H. Zeidler, "A Search Processor for Data Base Management Systems", Proceedings of the Fourth International Conference on Very Large Data Bases, West Berlin (September, 1978), pp. 280-287

[11] C.S. Lin, D.S. Smith, and J.M. Smith, "The Design of a Rotating Associative Memory for Relational Database Applications", ACM Transactions on Database Applications (March, 1976), pp. 53-65

[12] E.A. Ozkarahan, S.A. Schuster, and K.C. Smith, "RAP - An Associative Processor for Data Base Management", Proceedings AFIPS National Computer Conference (1975), pp. 379-387

[13] S.Y. Su and G.J. Lipovski, "CASSM: A Cellular System for Very Large Data Bases", Proceedings of the International Conference on Very Large Data Bases, Framingham (September, 1975) pp. 456-472

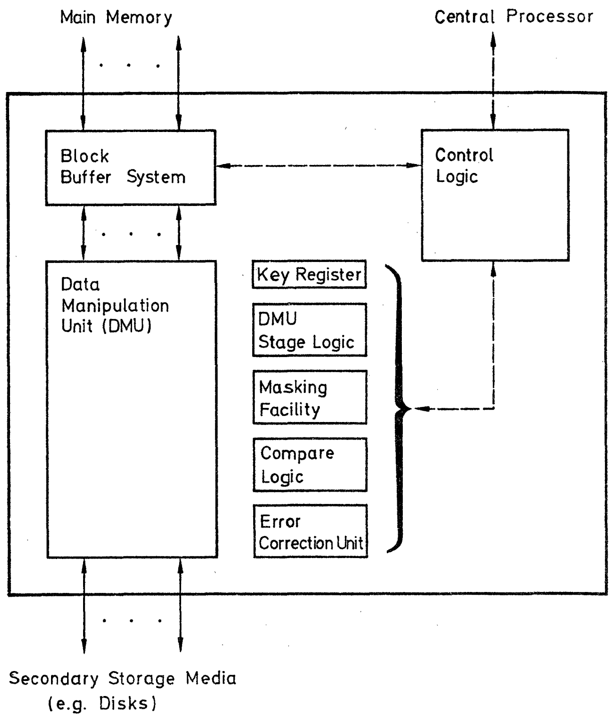


Figure 1. PTP Block Diagram

SESSION 10: DATABASE ARCHITECTURE AND SOFTWARE II

STOCHASTICALLY CONFLICT-FREE DATA-BASE MEMORY SYSTEMS

David Klappholz
Division of Computer Science
Polytechnic Institute of New York
Brooklyn, New York 11201

Abstract -- A multiple module memory system is termed stochastically conflict-free if its performance is (statistically) guaranteed - regardless of referencing behavior. A design for such systems has been proposed. In the present paper we present a formal analysis of its peformance.

1. Introduction

A parameterized design for a family of multiple-module memory systems will be termed "Stochastically Conflict-Free" if for any desired effective bandwidth (i.e., post-conflict bandwidth), β, and for any $\theta < 1$ and any $\varepsilon > 0$, the design makes possible the implementation of a system:

1. whose actual post-conflict bandwidth, β' is a random variable - not as a result of any statistical assumptions which might be made regarding the (memory module destinations of the) access requests entering the system, but rather as a result of an element of randomization deliberately introduced as a part of the design,
2. for which the probability that $\beta' \geq \theta\beta$ is itself within ε of 1 regardless of the referencing behaviors of the devices which input access requests - i.e., for every pattern of access requests which might be input into the system.

Such a design is proposed in [1] where the Stochastic Conflict-Freedom of its performance is argued informally.

The present paper presents formal analysis of the performance of systems implemented according to the design and operating in what we will term data-base mode. That is, it begins the development of a formal methodology for determining the values which design parameters should take in order for a system to meet specified performance criteria.

The type of system which we have in mind (pre-Stochastically Conflict-Free) is that of Figure 1. It consists of some number M, of memory modules accessed via an interconnection structure which might, depending upon M (the number of memory modules) and N (the number of ports) be as simple as a single shared bus or as complex as a routing network [2]. Requests for access to words (records) stored in the memory modules are entered into the system by request-issuing devices, (processors, query stations, etc.) each connected to one of the N ports; access requests traverse the interconnection structure to arrive at queues in front of the appropriate memory modules, and each is serviced once it reaches the head of its queue; finally, the appropriate response to an access request is returned to the requesting device via the interconnection structure.

In Sections 2 and 3 we review a number of required preliminary definitions. In Sections 4, 5, and 6 we motivate the proposed design, indicate its range of applicability, and review those of its details required for an understanding of the performance analysis presented in Section 7. Finally, in Section 8 we present, very briefly because of space limitations, a short design example.

2. Modes of Operation

We will distinguish two different modes of operation of multiple-module memory systems, multiprocessor mode and data-base mode, each mode carrying with it its own design questions.

2.1 Multiprocessor Mode

The use of a multiple-module memory system as a part of a multiprocessor (or MIMD parallel process) entails that the devices attached to the N ports of Figure 1 are processors and that the M modules of memory constitute that part of the primary memory which is shared by all N processors. In this case, the memory modules of Figure 1 would probably (today) be modules of semiconductor memory.

The multiprocessor mode of operation is defined as follows: Initially each processor issues a request for access to a word of memory. Thereafter, a processor will not issue an additional request before it has received a response to its previous request.

The total rate at which requests enter the system is not fixed in advance, but is, rather, self-regulating. On each cycle the number of new requests entering the system is bounded from above by the difference between N and the number of processors which have not yet received responses to their most recent requests.

2.2 Data-Base Mode

The use of a multiple-module memory system as part of a data-base system entails that the devices attached to the N input ports of Figure 1 are query stations and that access requests are for records rather than for individual words. In this case, the memory modules of Figure 1 would probably (today) be disks.

In the case of data-base systems one often ignores the rates at which individual devices connected to the ports of Figure 1 issue access requests, concentrating rather on the ensemble rate at which requests enter the system; this is reasonable because in the case of data-base systems the ensemble input rate is empirically observed to frequently assume an almost constant value. This value, probably as a result of users logging onto and off the system in response to the quality of service received, is often sustained for reasonably

CH1569-3/80/0000-0283$00.75 © 1980 IEEE

long periods of time.

The data-base mode of operation will thus be defined as the case of constant input rate. It is this mode of operation with which we will be concerned here.

3. Memory Contention

It is, of course, precisely the fact that the access request traffic appearing at the memory module queues of Figure 1 will not be uniformly spread across the queues that tells us that N memory modules will not suffice to drive N times as many processors as will a single memory module or to support a constant input rate of N times the response rate of a single memory module.

This is precisely what is meant by memory contention (conflict, interference); the degree to which it affects the performance of a system is, of course, dependent upon the degree of nonuniformity of the memory referencing behaviors of the devices connected to the N ports.

4. The Standard Statistical Assumptions

The following three statistical assumptions, which we will refer to hereafter as the "standard statistical assumptions," have been used (see [3] for example) as a basis for the study of the effective bandwidth to be expected from systems of the type depicted in Figure 1:

1. Each individual request for access to an item stored in memory is to a memory module chosen at random from a uniform distribution over all the modules.
2. The (memory module) destinations of access request input by different devices (attached to ports of Figure 1) are statistically independent of one another.
3. The (memory module) destinations of successive access requests input by the same device are statistically independent of one another.

If these assumptions indeed hold for some particular application, one would expect the effective bandwidth of an M-module memory system put to use in that application not to fall very far short of M times the bandwidth of a single module; the reasoning is, speaking very crudely, that the Law of Large Numbers would assure, in the long run, reasonable uniformity of spread of access request traffic over modules of memory.

5. The Proposed Design

5.1 Design Strategy

The design proposed in [1] is intended to allow the designer to bring the performance (effective bandwidth) of a multiple-module memory system as close as desired to what it would be if the standard statistical assumptions held.

In cases, then, in which the standard statistical assumptions can be assumed to hold, the design proposed in [1] would as we will see involve an unnecessary additional expense. On the other hand, it has great potential for:

1. cases in which neither can the standard assumptions be assumed to hold, nor are any other statistical assumptions regarding referencing behavior known or obtainable,
2. cases in which statistics regarding referencing behavior, although known or obtainable cannot be made use of in system design (hardware) or organization (software) - because actually obtaining such statistics, or using them, would be impractical or infeasible.

Given, then, that the cases in which the design will be of interest are those for which no a priori statistical assumptions can be made regarding the pattern of access requests entering the system, we will not be able, a priori, to view those requests as random variables with some known or knowable distributions and some known or knowable correlations to one another; rather we will view them as logical identifiers of items stored in memory and deliberately referenced by users (request-issuing devices) in whatever pattern suits the needs of those users' particular reasons for using the system.

5.2 Details of the Design

The design itself consists of three points:

1. Deliberate (uniform) random allocation of space to items when they are allocated space in the modules of memory - each item deliberately allocated space independently of all others.

This would be implemented through the use of either software, or, more probably, hardware generation of pseudo-random numbers. (In cases in which referencing behavior cannot a priori be assumed to be random (and independent in the way indicated), but in which allocation can, this step would, of course, be altogether unnecessary.)

2. Distribution of multiple modules of a novel type of memory which we will call "repetition filter memory" - RFM for short - over the internal components of the multistage interconnection structure (see Figure 1) proposed in [1]. (The exact nature of RFM will be detailed in Section 6.)
3. Increase in the number of ordinary memory modules - hereafter referred to as "modules of primary memory" beyond the number which would be required to produce the desired effective bandwidth if memory conflict did not exist.

5.2.1 Consequences of the First Point

The first design point ensures that even in the absence of any a priori assumptions regarding referencing behavior every access request traversing the interconnection structure is to a module of primary memory chosen at random from a uniform distribution over all M modules. It does not, of course, ensure that different access requests are to modules chosen independently of one another; indeed, different requests might deliberately be addressed to the very same item, and therefore the very same module.

The matter of how logical identifiers are translated into physical addresses when memory has been deliberately allocated at random is, by the way, quite simple. Each request-issuing device will have the highest level of file directory (or translation table) stored locally; the rest of the file directory (or translation table) will be stored as items in modules of primary memory in exactly the same way as are any other items. References to items of the former type will, then, be handled in the same Stochastically Conflict-Free manner as will references to any other items.

5.2.2 Consequences of the Third Point

The third design point, taken in conjunction with the first, ensures an increase in the expected uniformity of spread of request traffic over modules of primary memory regardless of the pattern of logical identifiers entering the system - i.e., so long as not all these requests are to exactly one and the same item.

5.2.3 Consequences of the Second Point

The effect of the second design point, whose details we defer to Section 6, will be to enable the designer to ensure that repeated requests to any item, if they are closer together in time than some chosen distance (say a distance of C-1 intervening requests, where C is itself a design parameter) will result in all requests but the first being serviced without actually being sent to the module of primary memory in which the item resides.

This will, in turn, ensure, informally speaking, that, as suggested in [1], from the point of view of actual effective bandwidth no pattern of entering access requests could be worse than one of the form $b_1,b_2,b_3,\ldots,b_C,b_1,b_2,b_3,\ldots,b_C$, (repeated indefinitely) where $i \neq j$ implies $b_i \neq b_j$.

The sense in which manipulation of the parameter C allows the designer to bring the performance of a system as close as desired to what it would be if the standard statistical assumptions held is that the assumptions are essentially to the effect that there are no deliberate repetitions (i.e., that C is infinite)

The extent to which this informal argument translates into formal results is the subject of Section 7.

6. Repetition Filter Memory

In the design proposed in [1], M and N are assumed to be large enough so that a complex routing network is required as the interconnection structure. As a further result of the assumed magnitudes of M and N the RFM has to be modularized to be capable of operating sufficiently fast.

In the present paper we will simplify matters by turning our attention to systems of the type depicted in Figure 2; i.e., we will assume that a single module of RFM is sufficient, and that the interconnection structure required is simple enough to be ignored. We will further assume that the RFM processes every request directed to it - i.e., every request entering the system in systems of the type depicted in Figure 2 - instantaneously.

In the case to be considered here, i.e., the case of multiple module data-base memory systems, if the RFM is built of very fast technology and the modules of primary memory are disks, then this last assumption will, for all intents and purposes be fully justified for values of M and N as high as in the hundreds or even possibly in the thousands. For the case in which a complex interconnection structure is, on the other hand, required, the analysis to be presented in Section 7 can be taken as a partial analysis, concentrating on the "access bandwidth" of the system of primary memory modules rather than on the "communication bandwidth" of the interconnection structure or on the effectiveness of a multiple-module (distributed) RFM in filtering out repeated requests.

6.1 Basic Mode of Operation

The operation of RFM resembles, but is certainly not identical to the operation of LRU cache memory. (We must, however, caution the reader that the reason for the introduction of this novel type of memory in [1] bears no resemblance whatsoever to the reason for which LRU cache is incorporated into conventional memory systems.)

Let $a_1,a_2,a_3,\ldots$ be a sequence of access requests entering the system; each a_i is processed by the RFM as follows:

i) If the item to which a_i requests either "read" or "write" access is already stored in the RFM then the request is serviced there instantaneously. In the case of a write access this means that the new value to be taken by the item is recorded into the RFM entry for the item, and an acknowledgment is sent to the request-issuing device. In the case of a read access the response to the request-issuing device is a copy of the item.

In neither case, of course, is the request sent along to the module of primary memory in which the item resides.

ii) If a_i is a write request and the item in question is not stored in the RFM, then the request is still serviced instantaneously in the RFM. In this case service consists of the creation of an entry for the item - the new value taken from the write request itself - and the sending of an acknowledgment to the request-issuing device.

Again, of course, the request is not sent along to the module of primary memory in which the item resides.

iii) In the remaining case, i.e., that of a_i being a read request for an item not stored in the RFM, a "pre-arrival" entry is created for the item; such an entry consists of the address of the item, and that of the request-issuing device. (The type of entry referred to in i and ii above will be termed a "post-arrival" entry.)

At some later point in time the item itself will reach the RFM - as a result of a response returning to the RFM from primary memory. At that time all requests which resulted in the creation of pre-arrival entries for the item will be instantaneously serviced. (In actuality they will be serviced by the fast RFM between the arrival of access requests to the much slower disk memory.) Exactly one post-arrival entry for the item will be created and will be the only entry for the item retained in the RFM.

In order to be sure that the item will, in fact, eventually reach the RFM from the module of primary memory in which it resides, exactly one of the requests which resulted in the creation of pre-arrival entries for it will be sent on the primary memory - viz. the earliest one.

(N.B. as we have described the operation of RFM a read request might be responded to with the value of the item which was current at the time of the request rather than with the very latest value. The definition of the operation of RFM is easily modified if this is not desired.)

6.2 Replacement Policy

The capacity of an RFM, i.e., the number of (pre- and/or post-arrival) entries it has space for is, of course, limited. Just as with an LRU cache, an RFM will, when it has to in order to make room for a new entry, throw away the least recently referenced entry it holds. In the case of RFM the definition of "recentness of use" is that pre-arrival entries are never considered to be "used" after they are created; i.e., they only age. Post-arrival entries on the other hand can be "refreshed" (viz.a-viz. recentness of use) exactly as are ordinary LRU cache entries.

If a pre-arrival entry whose creating request was not sent on to primary memory were ever simply erased, the creating request would never be responded to. "Throwing away" of such an entry thus consists of not just erasing it, but also reconstituting the request and sending it to the appropriate module of primary memory.

Finally, a post-arrival entry which has been written into in the RFM is "written through" into primary memory if and when it is thrown away.

6.3 Effect of RFM

We will assume, without loss of generality, that although access requests may enter a system from any of the N ports, no two requests enter at exactly the same time; i.e., the sequence $a_1, a_2, a_3, \ldots$ represents the system's input in temporal order. We further introduce the following notation:

1. The M modules of primary memory will be given addresses of $1,\ldots,M$.
2. y_i will be used to denote the address of the module of primary memory to which a service request is sent as a result of inputting a_i; the notation $y_i = 0$ will be used to indicate that inputting a_i causes no request to be sent to any module of primary memory - as a result of the filtering effect of the RFM.
3. Y_i^j will be used to denote the set

$$\{y_i, y_{i+2}, \ldots, y_{i+j-1}\}$$

We will assume for the sake of simplicity that no pre-arrival entry ever has to be thrown away. (A pre-arrival entry has to be thrown away only in the very low probability case that its creating request landed in a very-much-higher-than-average memory module queue.) The effect of the introduction of an RFM of capacity $C = mM$ entries (m an integer) taken together with the consequences of the first design point is then that without any a priori statistical assumptions regarding the sequence $a_1, a_2, a_3, \ldots,$ for every $i = 1,2,3,\ldots$ if $y, z \in Y_i^C$, then either

1. $y = 0$ or $z = 0$ or both

or

2. y and z are random variables uniformly distributed over $\{1,\ldots,M\}$ and are independent of one another.

7. Performance Analysis

We propose to analyze the performance of the type of system under consideration over finite periods of operation. The analysis will, thus, include a precise indication of the length of time a system will have to be in operation for the predicted performance to be achieved.

We will assume that the system is to be run at a rate of $\beta = nM$ input requests per unit of time, where $n \leq 1$ and the unit of time is the amount of time required by a single module of memory to service one request and be prepared to receive another. We will further assume that the system is to be run for at least the amount of time required to input $C = mM$ requests. Our results will apply as long as the system is in operation for at least this length of time.

Suppose, then, that for some ℓ we run the system for a sequence of $\ell C = \ell mM$ requests (for $\ell m/n > \ell m$ units of time) where $\ell \geq 1$ is, for the sake of simplicity, taken to be an integer.

Let:

1. $t = 1$ be the time at which the first of the ℓC requests enters the system.
2. $t = \tau = \ell m/n$ be the time at which the last of the ℓC requests enters the system.
3. $t = \tau'$ be a random variable representing the time at which the request which is serviced last has just been serviced.
4. β' be a random variable which represents the actual rate at which the system responds to the ℓC requests, i.e., let $\beta' = (\tau/\tau')\beta = \theta\beta$.

We will pose the following question about the performance of our data-base mode memory system:

Given any $\theta < 1$ what is the probability $P(\theta,\beta,\ell)$ that when the system is run for ℓC or more requests, it will <u>fail</u> to respond at an actual rate of $\beta' = \theta\beta$ or greater?

$P(\theta,\beta,\ell)$ can thus be thought of as the failure probability, i.e., the probability that the system will fail to perform at a level greater than or equal to that specified by θ, β, and ℓ.

For the purpose of our analysis we will be concerned with sequences of random variables X_1^h, $X_2^h,\ldots,X_{\ell C}^h$, where X_i^h, $i \le i \le \ell C$, represents the contribution of a_i to the number of access requests arriving at the h-th memory module, $1 \le h \le M$, as a result of inputting $a_1,\ldots,a_{\ell C}$. If we restrict our attention to $X_{j+1}^h, X_{j+2}^h,\ldots,X_{j+C}^h$ for some j, $j = 0, C, 2C,\ldots,(\ell-1)C$ then if no two of $a_{j+1},a_{j+2},\ldots,a_{j+C}$ are identical, then the X_{j+i}^h, $1 \le i \le C$, are independent and identically distributed as follows:

$$\text{prob}\left\{X_{j+i}^h = 1\right\} = 1/M$$

$$\text{prob}\left\{X_{j+i}^h = 0\right\} = (M-1)/M$$

If, on the other hand, two of $a_{j+1},a_{j+2},\ldots,a_{j+C}$ are identical, say $a_{j+t} = a_{j+s}$ where $1 \le t < s \le C$, then X_{j+s}^h is identically zero; this is the case because the request for a_s will never reach the memory module in which a_s resides; rather, it will be filtered out of the input stream by the RFM.

Still restricting our attention to a_{j+1}, $a_{j+2},\ldots,a_{j+C}$ for some particular j, $j = 0, C, 2C,\ldots,(\ell-1)C$, we note that the expected number of hits on the h-th memory module, $1 \le h \le M$, i.e., the expectation of $H_j^h = X_{j+1}^h + X_{j+2}^h + \ldots + X_{j+C}^h$, which we will denote by $E(H_j^h)$, is less than or equal to $C/M = mM/M = m$, reaching its maximum value if and only if no two of $a_{j+1},a_{j+2},\ldots,a_{j+C}$ are identical. (Note that in this case H_j^h is binomially distributed with parameters C and 1/M.)

Moreover, at the assumed rate of $\beta = nM$ requests per unit of time, the sequence of requests $a_{j+1},a_{j+2},\ldots,a_{j+C}$ is input to the system in $mM/nM = m/n > m$ units of time - an amount of time in which a single memory module can service $m/n > m$ requests.

We define, for each h, $1 \le h \le M$, a random variable S_j^h to represent the number of access requests (resulting from inputting $a_{j+1},a_{j+2},\ldots,a_{j+C}$) reaching the h-th memory module <u>in excess of</u> the number of requests that it can service in the time required to input C requests into the system. Formally, if we let $\alpha = 1/n > 1$, then the distribution of S_j^h is as follows:

$$\text{prob}\left\{S_j^h = 0\right\} = \text{prob}\left\{H_j^h \le \alpha m\right\}$$

$$\text{prob}\left\{S_j^h = k\right\} = \text{prob}\left\{H_j^h = \alpha m + k\right\} \quad \text{for } k > 0$$

Note that S_j^h is not the number of requests <u>remaining</u> in the queue in front of the h-th memory module just after $a_{j+1},a_{j+2},\ldots,a_{j+C}$ have been input - not even for $j = 0$ - but is rather the total number of requests which have <u>arrived</u> at the h-th memory module during the period of input of $a_{j+1},a_{j+2},\ldots,a_{j+C}$; we have, as yet, said nothing about <u>when</u> the requests that arrive at a module actually arrive.

Finally, we define for each j, $j = 0, C, 2C,\ldots,(\ell-1)C$, a random variable

$$S_j^* = \max_{1 \le h \le M} (S_j^h)$$

i.e., the "excess" at the "most heavily hit" memory module. It is easy to see that

$$\text{prob}\left\{S_j^* = k\right\} \le M \,\text{prob}\left\{S_j^h = k\right\} \qquad (1)$$

In what follows we will need an approximation of $E(S_j^*)$. Using (1) as well as a) the standard derivation of the mean deviation for the binomial distribution [4] pp. 176-177 and b) Stirling's approximation [5] p. 172 it is possible to show that

$$E(S_j^*) < \frac{\sqrt{m}M}{\sqrt{2\pi\alpha}}\, e^{(\alpha-1-\alpha\ln\alpha)m}\, e^{\left(\frac{\alpha}{M}\left(m-\frac{1}{M}\right)+\frac{1}{M}\right)} \qquad (2)$$

We turn now to the question of the number of requests <u>remaining</u> in the highest queue. Let the time line of Figure 3 represent the ℓ periods, each of C input requests, (each of duration αm units of time) with which we are concerned.

Precise results regarding the distribution of $\tau' - \tau$, which is the number of requests remaining in the highest memory module queue at time $t = \tau$, involve not only the excess <u>numbers</u> of arrivals to the various modules during the ℓ periods, but also the <u>precise times</u> of arrival. Since such results appear to be difficult to obtain, we will content ourselves with considering the maximum value that $\tau' - \tau$ could possibly attain for any given value of $S_0^* + S_C^* + \ldots + S_{(\ell-1)C}^*$

We will, in effect, assume arrival times and identities of "most heavily hit" memory modules which, given any particular value for $S_0^* + S_C^* + \ldots + S_{(\ell-1)C}^*$ will leave the greatest possible number of unserviced requests in the highest memory module queue at time $t = \tau$.

To wit, we will assume the following:

1. There is a memory module which is a (the) most heavily hit module for all ℓ periods of input, and this module receives a nonzero excess of requests during each period.

2. However many access requests arrive at the most heavily hit memory module during each period, they all arrive at the very end of the period - i.e., too late for any of them to be serviced during that period.

It should be clear that however many access requests arrive at the various memory modules over the ℓ periods of interest, $\tau' - \tau$ is maximum under assumptions 1 and 2 above.

But under these assumptions we have that

$$\tau' - \tau = \alpha m + S_0^* + S_C^* + \dots + S_{(\ell-1)C}^*$$

i.e., no requests are serviced during the first period, the first αm requests arriving during the first period are all serviced during the second period, and during each subsequent period exactly αm requests are serviced. The number of requests which then remain in the queue in front of the most heavily hit memory module is the sum of all the excesses for the ℓ periods plus the first αm requests which arrived during the first period, but were not serviced then.

Thus we have

$$P(\theta,\beta,\ell) \le \text{prob}\left\{\frac{\alpha\ell m}{\alpha m+\alpha\ell m+S_0^*+S_C^*+\dots+S_{(\ell-1)C}^*} > \theta\right\}$$

$$= \text{prob}\left\{\frac{\alpha\ell m}{\alpha(\ell+1)m+S_0^*+S_C^*+\dots+S_{(\ell-1)C}^*} < \theta\right\} \quad (3)$$

$$= \text{prob}\left\{S_0^*+S_C^*+\dots+S_{(\ell-1)C}^* > \frac{\alpha\ell m-\alpha(\ell+1)m\theta}{\theta}\right\}$$

(Note that from the second line of (3) we can see that, according to our pessimistic approximation we cannot hope to achieve an effective service rate of $\beta' = \theta\beta$ until $\ell/(\ell+1) \ge \theta$ - that is even if $S_0^* + S_C^* + \dots + S_{(\ell-1)C}^* = 0$.)

Now, for any $k > 0$ (see [5], p. 242).

$$\text{prob}\left\{S_0^*+S_C^*+\dots+S_{(\ell-1)C}^* > k\right\}$$

$$< \frac{E(S_0^*+S_C^*+\dots+S_{(\ell-1)C}^*)}{k} \quad (4)$$

$$= \frac{\ell E(S_j^*)}{k}$$

$$< \left(\frac{\ell}{k}\right)\frac{\sqrt{mM}}{\sqrt{2\pi\alpha}}\, e^{(\alpha-1-\alpha\ln\alpha)m} \cdot e^{\left[\frac{\alpha}{M}\left(m-\frac{1}{M}\right)+\frac{1}{M}\right]}$$

Finally, from (3) and (4) we have

$$P(\theta,\beta,\ell) < \frac{\sqrt{mM}}{\sqrt{2\pi\alpha}}\, e^{(\alpha-1-\alpha\ln\alpha)m} \cdot e^{\left[\frac{\alpha}{M}\left(m-\frac{1}{M}\right)+\frac{1}{M}\right]} \times \left[\frac{\theta}{\alpha m-\frac{\alpha(\ell+1)}{\ell}m\theta}\right] \quad (5)$$

Table 1 gives values of $\alpha-1-\alpha\ln\alpha$ for some possible values of α.

α	$(\alpha-1-\alpha\ln\alpha)$
1.1	-.0048412
1.2	-.01878587
1.3	-.04107354
1.4	-.07106113
1.5	-.10819766
1.6	-.15200581
1.7	-.20206803
1.8	-.2580160
1.9	-.31952238
2.0	-.38629436

Table 1

The values in the right-hand column of Table 1 indicate that for α in the range under consideration the factor $e^{(\alpha-1-\alpha\ln\alpha)m}$ in our very pessitic overapproximation of $P(\theta,\beta,\ell)$ decreases exponentially as m is increased. (Note that for $\alpha = e$, $e^{(\alpha-1-\alpha\ln\alpha)m} = e^{-m}$, and for $\alpha = ae$, $a > 1$, $e^{(\alpha-1-\alpha\ln\alpha)m} = e^{-bm}$ where $b > 1$.)

8. A Numerical Example

Consider the design of a fairly large system, i.e., one which we wish to drive at a peak rate of 1000 requests per unit time. Suppose that we desire an effective service rate of .99 or more of the constant (peak) input rate after 100 request cycles, and that we are willing to sustain the cost of 1500 modules of memory to accomplish this. (In an actual design study of course, α need not be chosen in advance; rather α and m can be traded-off against one another on the basis of the incremental cost of memory modules and the incremental cost of RFM capacity.)

In the present example we have $M = 1500$, $\alpha = 1.5$, $\theta = .99$, and $\ell = 100$. A few quick computations using (5) reveal that the following failure probabilities can be achieved with the indicated amounts, m, of RFM capacity per module of primary memory:

m	$P(\theta,\beta,\ell)$
1000	2.8694×10^{-42}
300	7.6093×10^{-19}
250	4.6847×10^{-7}

References

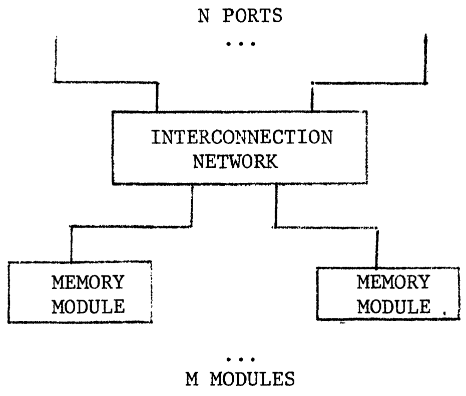
[1] Sullivan, H., Bashkow, T. R., Klappholz, D., and Cohn, L., CHOPP: Interim Status Report 1977, submitted for publication.

[2] Benes, V. E., Mathematical Theory of Connecting Networks and Telephone Traffic, Academic Press, N.Y. 1965.

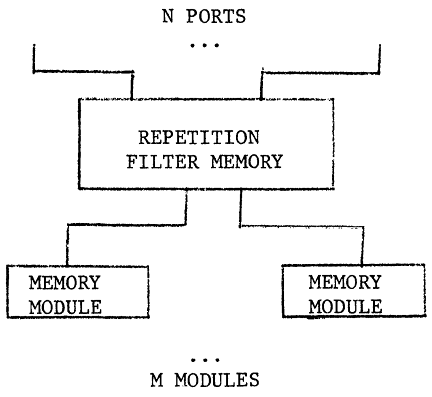
[3] Baskett, F. and Smith A. J., "Interference in Multiprocessor Computer Systems with Interleaved Memory," Commun. Ass. Comput. Mach., Vol. 19, No. 6 pp. 327-334, June, 1976.

[4] Upsensky, J. V., Introduction to Mathematical Probability, McGraw-Hill, New York, 1937.

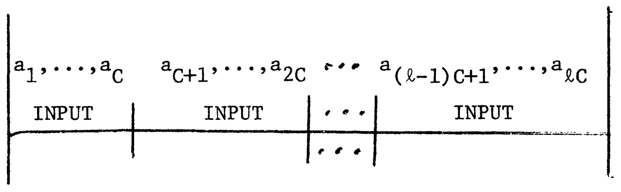
[5] Feller, W., An Introduction to Probability Theory and its Applications, John Wiley and Sons, Inc., New York, 1968.



MULTIPLE MODULE MEMORY SYSTEM
Figure 1



SYSTEM WITH A SINGLE-MODULE REPETITION FILTER
Figure 2



ℓ PERIODS OF C REQUESTS EACH
Figure 3

PANEL DISCUSSION:

DESIGNING HIGH PERFORMANCE COMPUTER SYSTEMS

A MANUFACTURER'S VIEWPOINT

Robert J. Malnati
Advanced Systems, Programs
Product Development, Major Systems Division
Sperry Univac
Roseville, Minnesota 55113

You might not be surprised to hear that scientific processors will be successful products, but the question is will they be successful enough? Development money is costly, engineers are in short supply and general commercial products are selling well.

As manufacturers we capitalize on experience, technology, manufacturability, human resources, service operations, investment in software and the general company processes that we know and understand. This assures a smooth running operation, a reasonable return on investment and continued growth for our users. But, what about attempting to develop an entirely new business area where it may require deviation from the norm? Many decisions concerning new business ventures are made by relatively uninformed product planners and their technical staffs. These decisions affect marketability, the market as a whole, thousands of people (users and vendors alike), and the expenditure of millions of dollars.

Now let's explore the character of this problem and attempt to answer some questions: What does "relatively uninformed" mean? What can we do to improve this process? What can researchers do to help?

Guiding forces of scientific data processing in the past were Federal Government agencies, such as NASA in orbit and reentry dynamics and structural analysis, the Air Forces in wind tunnel simulation, and the Atomic Energy Commission in nuclear modelling. The demand for better modelling in energy conservation, the demand for better weather prediction, and the demand for advances in technology assure "super computers" of a continuously growing market. Today, a new swell of attached scientific processor hardware design activity is emerging out of the need for high performance scientific processing at an extremely low cost. Some single product vendors are reaping the benefits from this economical hardware. Even so, some of this market is not fully satisfied by these vendor products due to lack of software, support and capabilities for a total systems approach.

Sperry Univac has been known for years as an industry leader in scientific computing, and as with any aggressive company, will continue to exploit this market. The logical question is, just what is necessary to satisfy industry's appetite in this fast growing and changing market? Greater performance, to be sure; lower cost per computation...yes, definitely; but industry users are changing in other dimensions as well. Among traditional high performance scientific demands, users expect three primary things:

- ease of use, the ability of a non-computer scientist to use the system;
- program compatibility, the ability to capitalize on millions of dollars invested through program development;
- and high total system availability to prevent the loss of a half finished job.

Yes, the climate has changed, entry into this market today does not have to be a head-on collision with the "super computer" manufacturers. It could fill an un-filled market requirement with a more traditional entry for the well established vendors.

The question remains, what other critical ingredients are necessary to lessen risks for large computer manufacturers and provide motivation to launch into new market areas such as scientific vector processing? There are no simple or pat answers, but careful and cautious strategic and technical planning can minimize the investment risk, establish the proper place in the market, and prevent false starts in implementation. Even with a carefully planned strategy, most new ventures are doomed to disaster. Is it any wonder management is hesitant to enter into speculative markets?

We as product planners search for better mechanisms to provide a broader spectrum of feasibility assurance. What we worry about is the lack of knowledge and experience that could lead to marginal gut-level trade-off decisions in design that might cause the demise of the product.

In-house technical and feasibility studies, research and academic papers, and consultants are, to be sure, heavy contributors to support this critical decision-making process. Even with all this design resource applied at the definition point in the product development process, they cannot provide all the facts necessary for a proper trade-off decision. By default then, many decisions that should be made on a clearly technical basis get supplemented by special considerations of strategy or policy. To illustrate this, if we use CRAY-1 as an experience base, this would imply that a free-standing executive system (memory manager at least) should be used in new or competitive designs. The question is then, was this successful product a result of a strategic decision by CRAY, or did it come from a good solid technical base? Another alternative is, if a vendor has general purpose host capabilities, strategically it would be wise from a development, support, and system configurability point-of-view to assign system management functions or executive functions to the host. To illustrate this, if we use 1100 Systems as an experience base, the technical rationale for a separate or free-standing executive gives way to the strategic one that says: Sperry Univac is in the business and has experience and know-how for multiprocessor systems. This, therefore, means one executive (one master) in the host that manages and schedules all resources, host processors, all peripherals, real memory and I/O traffic. Studies and experience thus far indicate this is viable, but without peripherals (disks) directly on the compute engine, how much peripheral and input/output traffic can the system stand without being brought to its knees?

There are still other ways to view these system design decisions. User communities, it seems, are beginning to demand an integrated system or multiprocessor approach. Even those that once demanded the specialized compute engine now are demanding this compute power together with all features and functionality afforded the general purpose commercial user. They want the benefits offered by a system with a full-blown operating system, tried and proven (stable) executive, FORTRAN/COBOL compilers, data management facilities, interactive capabilities, etc. With this they get 10-20 years of system experience that translate to availability. Conclusion? The tightly coupled (multiprocessor) approach is a good decision in terms of marketability. But what about performance penalties in living with constraints of coexisting with other processors in a multiprocessor system? The question of heterogeneous processor accesses to multiple memory modules contains many unknowns and demands much study. This challenge is typical of the tradeoff facing the product developer.

Again, reflecting on some architectural basics brings to mind another critical area of getting data to and from processors. The question is, which is the most optimal approach register-to-register or memory-to-memory, or some other? Burroughs and CRAY differ in architectural concept; it would be beneficial to have some dialogue on those differences. Where are the research studies to support that decision process? It may even be interesting to perform them after the fact. This is another example of a situation where scientists and implementers could maintain a close correspondence.

By necessity, detailed studies on a particular idea, design approach or concept, result in analyses that show performance in a narrow spectrum. For example: a certain method of combining 1024 processors will neatly handle partial differential equations for a wind tunnel problem. Industry then attempts to interpret, extrapolate, and guess how this problem would map on an architecture that covers a wider spectrum of applications. By necessity, industry must cover a wide spectrum of applications to increase quantities, amortize cost, and in short, make a profit. Vendors do not have time, money and resources to verify in detail those guesses made in an attempt to make the product more palatable to a general market. True, industry has millions to invest in new products, but budgets are always strained...there is no luxury here, either.

The bottom line is that normal processes between researchers and industry vendors do work well much of the time. Researchers concentrate in their specialty areas, while product designers are responsible to evaluate, interpret, and select results that apply best to them. What is needed is to improve upon this process? Gaps between research and industry are too wide. Are there other steps that may be taken? Is another level of iteration possible wherein a decision regarding product posture can be fed back to research in order to further check validity? It seems that if one facet of a major venture could be better focused and coordinated, the energy expended in developing better communication between implementers and scientists would be well worth the labor and trouble.

GENERAL PURPOSE SUPERCOMPUTERS

Burton J. Smith
Denelcor, Inc.
Denver, Colorado 80205

Summary

There are two properties that are shared by all supercomputers, namely, they are parallel and fast. Unfortunately, these may be the only two properties that supercomputers have in common. There are three additional properties that are necessary (although perhaps not sufficient) for a supercomputer to be "general purpose". These properties will be desirable for some supercomputer users and irrelevant for others, just as the general purpose attributes of a more classical computer system are.

First, a general purpose supercomputer should be reasonably fast in its execution of any algorithm that performs well on another machine. The intent of this requirement is that any kind of parallelism should be exploitable. Second, a general purpose supercomputer should provide a machine-independent programming environment; that is, software should be no harder to transport from a given computer system to a general purpose supercomputer than to an ordinary general purpose computer. Third, a general purpose supercomputer should have storage heirarchy performance consistent with its computational capabilities. Such a computer should not be I/O bound to any greater extent than an ordinary general purpose computer is for a given class of problem.

These three requirements are more than just a short list implying what is wrong with today's supercomputers. They are the principal reasons for the schism between the parallel processing business and the mainstream of computing practice. A supercomputer that satisfies these three requirements could enjoy a market several orders of magnitude larger than the current models do. While there will always be a need for special purpose parallel processors of all sizes and capabilities, it will be the general purpose super or not-so-super computers that dominate the marketplace.

If general purpose supercomputers are to become a reality, substantial progress is required in three areas. First, MIMD and data-driven architectures offer the best hope for exploiting many kinds of parallelism, but these architectures have few proponents outside the academic community. In fact, no MIMD or data-driven supercomputer has ever been delivered to a customer for trial. This situation will improve in the next few years, but until it does the design of these kinds of computers will not be able to benefit from experience with practical applications.

The second area in which progress is required is that of machine-independent programming. Two approaches are needed here: parallelizing compilers for existing languages, and new languages in which parallelism is more easily detectable. Although there is some experience in automatic vectorization of FORTRAN, for example, it is only recently that attempts have been made to find and exploit other kinds of parallelism in existing languages. It also seems clear that languages like FORTRAN and COBOL will be in use for a long time to come and will therefore need to perform well on general purpose supercomputers. On the other hand, the advantages to be gained in parallelism by being able to express algorithms in functional programming languages must not be discounted; it is with these languages that the future of very high speed general purpose computing lies.

Finally, the single most important technological factor in general purpose supercomputer development is mass storage bandwidth. The failure of mass storage access times and data rates to keep pace with the speed increases realized by solid state technology are well known. The effect of this deficiency has been to severly constrain the range of application of very high speed computers. A modest increase in mass storage bandwidth would have far more impact than more substantial advances in device speed or packaging density. In fact, only an improvement in interconnection technology would have as much impact on computation in general.

SESSION 11: DISTRIBUTED PROCESSING II

HIERARCHICAL ANALYSIS OF A DISTRIBUTED EVALUATOR

Robert M. Keller

Gary Lindstrom

Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

ABSTRACT

We outline the analysis of a distributed evaluator for an applicative language FGL (Function Graph Language). Our goal is to show that the least fixed point semantics of FGL are faithfully implemented by the hardware evaluator envisioned in the Applicative Multi-Processor System AMPS. Included in the analysis are a formalization of demand-driven computation, the introduction of an intermediate graphic language IGL to aid in our proofs, and discussion of pragmatic issues involved in the AMPS machine language design.

INTRODUCTION

Programming languages for distributed computing systems are receiving increased attention currently, as are languages based on function application. Distributed systems are of interest because of a desire to exploit potential concurrency in programs. Applicative languages tend to reveal potential concurrency by eliminating arbitrary sequencing within program representations, and by circumscribing side-effects. In addition, applicative languages often allow programs to be written so that their text closely resembles that of a correctness specification, thereby easing verification.

Although the idea of using applicative languages as a basis for concurrent programming has come into vogue only recently, the reader should refer to the prophetic paper [1] for an anticipation of many of the relevant ideas currently being put forth. Subsequent proposals, which share some aspects of our own, include [2] through [7].

Sketched herein is an analysis (i.e. an informal correctness proof) of an evaluator for an applicative language suitable for exploiting the features of a distributed computing system. This evaluator has been proposed for use in the Applicative Multi-Processing System AMPS [8]. Such a proof would be of interest for several reasons:

1. The evaluator has been implemented ([9]), so it is desirable to certify its correctness.

2. Although parts of similar proofs have been sketched, notably in [10] and [11], these proofs have been for serial evaluators, and are for models having fewer machine-level details than the one presented here.

3. A graphical approach to semantics seems to us to be quite enlightening in comparison to the one-dimensional representations largely used heretofore.

4. We intend the present exposition as the first step toward a more comprehensive proof which also involves a storage manager.

The extrinsic language, called FGL (for Function Graph Language), includes features deemed relevant to highly concurrent distributed evaluation. The hardware implementation we consider includes a <u>demand-driven</u> data-flow evaluator for effective support of the data structuring primitives of our language. The implementation naturally provides for single evaluation of common subexpressions and parameters.

Locality considerations give rise to a <u>two-level evaluation</u> strategy for the machine language (ML): at the <u>intra-processor</u> level, a rather rigid structure is imposed, in which each atomic function is executed with bounded value fan-out and communication delay for greatest efficiency. At the <u>inter-processor</u> level it is infeasible to place such a bound, as one function may well have to send its result to others, the number and locations of which are not determinable a priori.

<u>Block storage allocation</u> is used in ML for the following reasons:

1. It enforces locality of communication among nodes which are logically closely related.

2. It permits economical use of address bits by requiring only relative addresses within a block.

3. It avoids the need for code relocation and extensive dynamic binding.

4. Tuples of data values are stored as blocks, or pieces of blocks, permitting fast indexing.

5. Fewer interactions with the storage allocator are required.

6. Blocks may be transmitted and initialized in a "burst mode" of communication, rather than in a word-by-word mode.

The proof that the distributed evaluator is correct with respect to FGL's fixed-point semantics is complicated by the two-level

CH1569-3/80/0000-0299$00.75 © 1980 IEEE

block-oriented strategy. For this reason, we have found it convenient to introduce a language IGL intermediate between the extrinsic language and that of the target machine. This language allows the analysis to be naturally decomposed into two levels (not corresponding to the levels of evaluation), but does not appear explicitly in the implementation.

We express the notion of demand and value flow in IGL programs as a state-transition system (cf. [12]). The states are marked IGL graphs, with transitions expressed by a set of formal rules. This system is the basis for the FGL evaluator. The notion of the correctness of such an evaluator with respect to FGL semantics is presented. We then discuss the proof of correctness of the IGL evaluator with respect to ML.

The following diagram summarizes the levels of the hierarchy and their functions.

Acronym	Name	Purpose
FGL	Function Graph Language	Programming
IGL	Intermediate Graph Language	Internal program representation for FGL
ML	Machine Language	Physical program execution

The analysis may be outlined as follows:

1. IGL-->FGL mapping theorem: IGL defines the correct result for FGL.

2. IGL partial correctness theorem: IGL can produce the correct result.

3. ML-->IGL mapping theorem: ML defines the correct result for IGL.

4. IGL finite delay: ML provides a finite delay property for IGL, so that "can" above becomes "will".

5. Pragmatic aspects: Certain invariants desired for implementation reasons hold for ML executions.

FUNCTION GRAPH LANGUAGE

Our extrinsic language, FGL (Function Graph Language) is Lisp-based [13], extended to include non-strict atomic and programmer-defined functions. This permits ease in dealing semantically and pragmatically with unbounded data structures, as discussed in [6] and elsewhere. The components of such structures may be distributed among physical processing elements and concurrently constructed and transmuted, using stream-like communication between computing modules which are both physically and logically distributed. Because of the functional nature of FGL, logical aspects of the computation are insensitive to delay in and among physical elements.

The objects supported are not restricted to streams of simple components, such as characters or records, but also permit components which are functions, other streams, and generally arbitrary data objects. FGL allows treatment of functional objects with full lambda-calculus generality [14].

The **cons** operator of FGL permits an arbitrary number of arguments, thus providing an efficient and natural array capability. The usual **car**, **cdr** selectors are generalized to an indexing selector **select**. For simplicity, however, we will primarily use **car** and **cdr** here; **car** selects the first component of a tuple and **cdr** selects the last. Other aspects of our generalization are discussed in [15].

In this presentation, the set of data objects of FGL will be

Objects = Atoms U Tuples U Graphs U {error} U {?}

where

1. Atoms = Integers U Characters U {NIL}, where Integers is the set of integers and Characters is the set of characters of some alphabet. We assume that NIL plays the role of the Boolean value **false**. Any atom other than NIL and error may play the role of the Boolean value **true**.

2. Tuples: A tuple is a sequence of N Objects, for N an arbitrary natural number.

   The limit of a sequence (i.e. "tree") of nested tuples of objects, as nesting occurs ad infinitum, is an object. For example, the stream of odd prime numbers could be represented as

   (3, (5, (7, (11, (13, ...)))))

3. Graphs: We allow the enveloping of a graph, as described in [16], and its use as a function data object (i.e. as a "closure").

4. error: an error value which propagates itself through each function which demands it as an argument.

5. ?: the undefined object, i.e. the result of a computation which has not yet (and might never) produce any value.

A fully operational system might include side-effect operators, but we prefer introducing them within the context of an applicative style, in which the programmer is highly aware of their use (i.e. their use will be permitted only on tuples which are created as explicitly modifiable). Side-effect operators are not included in the model presented here, with the exception of **read** and **print**, which are described subsequently.

For the purposes of this exposition, a program in FGL appears as either a "function graph" or as a "set of equations" [22]. Each equation is determined by naming a FUNCTION being defined, which has zero or more formal parameters. The function name is equated to the RESULT expression, which involves names of defined functions, names of atomic operators, formal parameters, and imported values. Abbreviations of multiply-used values are provided by LET expressions, which are also equations equating the left-hand side identifier of a BE to the right-hand side expression. The latter expression may involve the identifier on its own left-hand side, as can the function being defined involve itself. Finally, an IMPORTS declaration allows values defined externally to a function definition to be used inside the definition. Algol-like lexical scoping is used, except that imported values are declared implicitly.

When a value defined in a LET.... BE.... involves itself, or when a function f defined in terms of a formal variable x involves the expression f(x), or when a value is defined in terms of an expression which involves the importation of the value itself, we say that there is an "applicative loop". Such loops permit implementation of data structures in terms of themselves, thereby providing for the generation of infinite data structures without either the obvious infinite recursion or use of side-effect operators such as Lisp's **rplaca**. The latter often have the effect of destroying local determinacy, a property useful in verifying concurrent programs.

As an example of a textual representation of an FGL program, consider the following:

```
FUNCTION oddprimes(limit)
LET primes be
        cons(3, primesfrom(5))
RESULT  primes
WHERE

FUNCTION primesfrom(n)
IMPORTS (primes, limit)
LET rest BE primesfrom(n+2)
RESULT  if n > limit
              then nil()
              else if relprime(primes)
                  then cons(n, rest)
                  else rest
WHERE

FUNCTION relprime(stream)
IMPORTS n
LET first BE car(stream)
RESULT  (square(first) > n
         or ((not divides(first, n)
                 and relprime(cdr(stream)))
```

The program above generates the list of prime numbers beginning with 3 and not exceeding the value of the argument **limit**. It does so by forming a sequence of numbers, a number being included in the sequence only if it is prime. The primality of the number is tested by using lesser members in the sequence as trial divisors.

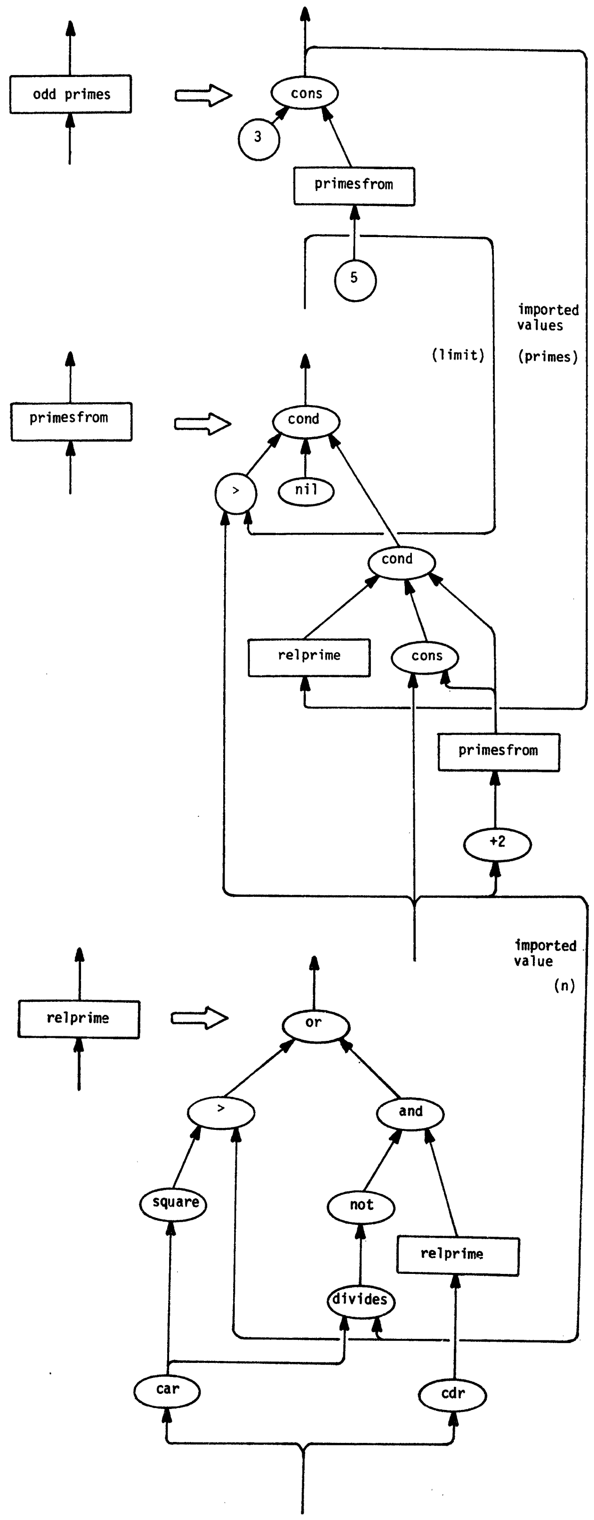


Figure 1: FGL graph of the Odd-Primes Example.

An applicative loop exists, in that **primesfrom** is used to define the sequence primes, but also uses that sequence as an imported value in its definition. The **or** above is a sequential function, in that it only demands arguments in sequence as they are needed to determine the value.

An expression in FGL is formally represented as a directed graph, with the nodes being identified with the operators in the expression. We think of each arc in the graph as being a carrier for an FGL data object. A node defines an input/output functional relationship between the ultimate values on the arcs directed into the node and the ultimate value on the arc directed out. (We assume that each node has a single outgoing arc for simplicity.) In the graphical form of FGL, each functional equation may be represented by a <u>graph grammar production</u> in which the antecedent names the function being defined, and the consequent presents the graph of the defining expression.

The graphical form of the preceding program is shown in Figure 1. The applicative loop which results from the compilation of the textual FGL program is evident there. Although in this figure we represent imported values by direct connections into the consequents of productions, accurate treatment of scoping rules demands that productions involving imports be replaced with the concept of an <u>enveloped</u> graph, which may eventually be presented as an argument to the **apply** function [16]. To simplify the discussion, we shall not consider this treatment here.

Certain atomic functions are provided, such as the following:

add, **and**, **divides**, **mult**, etc. which have the obvious interpretation,

cons groups its arguments into a tuple, even if the arguments are not completely known at time of application. That is,

$$\textbf{cons}(x_1, x_2, \ldots, x_n) = (x_1, x_2, \ldots, x_n)$$

where the right hand tuple exists independent of what the x's might be.

select is defined by

$$\textbf{select}(i, (x_1, x_2, \ldots, x_n)) = x_i$$

provided $i \neq ?$. It is undefined if $i = ?$, but when $i \neq ?$, there is no requirement that $x_j \neq ?$, for any j.

car and **cdr** are defined by

$$\textbf{car}\ (x_1, x_2, \ldots, x_n) = x_1$$

$$\textbf{cdr}\ (x_1, x_2, \ldots, x_n) = x_n$$

which is consistent with the Lisp definition when n = 2.

cond is the name of the conditional function (i.e. "if.... then.... else....").

nil, returns the value NIL.

null, tests for the atom NIL.

Additionally, there are "pseudo-functions", such as **print**, which has the side-effect of printing its argument on some external device, and **read** which has the side-effect of reading an external device to determine its result. The use of such functions can be completely avoided outside of utility routines provided for input and output.

Additional auxiliary functions are provided for extra evaluation control. Examples are **seq**, which causes its arguments to be evaluated in sequence, and **par** which causes its arguments to be evaluated concurrently. (Strict functions such as **add**, **mult**, etc. also have the latter effect.)

Through the use of pre-compilation and removal of certain recursions and common subexpressions, our evaluator incurs no combinatorial explosion of the type which would normally occur in circular recursive evaluation of applicative loops. All theorems proved in [10] also hold for the FGL evaluator. However, the fact that we compile applicative loops without additional recursions provides a feature for yielding terminating executions for evaluations which would be non-terminating in other systems. For example, we can state

<u>Theorem</u>: The FGL evaluator terminates on some programs for which the evaluator of [10] fails to terminate.

To prove this, consider the program (which would differ syntactically when presented to the Friedman and Wise evaluator):

```
FUNCTION main
RESULT  print f(0)
WHERE
        FUNCTION f(x)
        RESULT car f(x)
```

The Friedman and Wise evaluator would recurse infinitely, generating

```
print(car(car(car(car(...)))))
```

The FGL evaluator stops (without printing anything) when it dynamically and implicitly "discovers" that f(x) is trying to compute a <u>strict</u> function of itself.

We do not present the fixed point semantics of FGL here, instead referring the reader to [16]. However, we give a brief intuitive description of these semantics. For a directed <u>acyclic</u> function graph, the meaning can be understood simply from the definitions of the functions assigned to each node. That is, the output of each node is the function prescribed for the node applied to the input values of that node. Note that this makes sense even if the graph is infinite, so long as each path from each of the graph's inputs to its output is finite.

In FGL, the program representations are always finite, but these representations can be understood by (but are not implemented by) expanding the representations into acyclic graphs which are sometimes infinite. Namely,

1. Each node having a function prescribed by a production is effectively the same as replacing that node with the consequent of the production.

2. Each cycle in the graph can be "unwound" by repeated "node-splitting" to obtain an equivalent infinite acyclic graph.

The validity of this means of understanding depends on the fact that all FGL functions are "continuous" over an appropriate Scott data-type ordering. Although this fact is used later, space does not permit further elaboration of its meaning, and the essential ideas may be understood without it. The reader may refer to [16] for further explanation.

Space limitations also preclude further definition of "node-spliting", but the idea is reasonably intuitive. Further discussion may be found in [16]. We henceforth understand by the phrase dag form of an FGL program the acyclic graph as determined above. The above description is equivalent to the "least fixed point" semantics of FGL programs, which is also equivalent to the viewpoint of the program as a system of equations. It also points out the determinacy of FGL programs, i.e. that each program represents a unique function.

The diagram of Figure 2 illustrates the scheme of evaluation in the odd primes example of Figure 1. It shows the loop formed by using the sequence of primes being generated to assist in their own further generation, as well as concurrent evaluation of **primesfrom** for different arguments. The dag form resulting from unwinding the cycle is shown in Figure 3.

Implicitly included in an evaluation such as the one above is an arbitrary number of "producer-consumer" relationships which the evaluator must implement so that needed values are produced and used consistently, independent of system-wide interleaving. These evaluations could be distributed among processing elements to heighten concurrency and thereby reduce computing time. The arbitrary fan-out of values, alluded to earlier, is quite apparent in the diagram.

It should be noted that the definition of FGL semantics is embodied in the language, not the evaluator. That is, its semantics are given denotationally, by specifying the semantics of each of the atomic functions. This is why we prefer to use the term "lenient **cons**" instead of saying that we have a "lazy evaluator" [11]. For a denotationally-defined language, an evaluator is either correct or is not. Similarly, if one wishes a **cons** to have a different effect, this amounts to a redefinition of **cons**, not a change in the evaluator. We happen to prefer the lenient version of **cons** as a standard, but our results in no way rely on the presence of this operator. We can also include other forms of **cons** (with different names, of course). The main

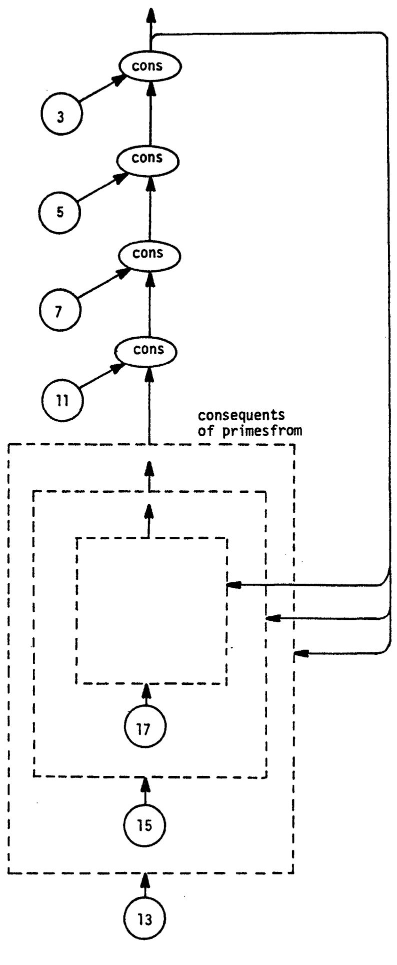


Figure 2: Expansion of the Odd-Primes Example.

reason lenient **cons** is of interest here is because it is the source of a need for "forward chaining", to be discussed later.

A single equation, which defines the "top level" function **main**, acts to drive the others, its value being demanded externally by the system. In a sense, it is the goal of the evaluator to produce the "value" of **main**. For example, we might include the definition of **oddprimes** above in the following program, which reads a number, then prints all odd primes not greater than that number.

```
FUNCION main
RESULT  printall(oddprimes(read()))
WHERE

FUNCTION printall(x)
RESULT   if null x
                   then nil()
                   else seq(print car x,
                                   printall cdr x)
```

The program above, when run on our evaluator, will not produce a particularly high degree of concurrency. However, it is a simple matter to enhance its concurrency with the special operator, **par**, which is functionally transparent (it is the identity function on its first argument), but which has the effect of introducing additional demands for values. In the present example, we need only modify the definition of **primesfrom**, obtaining the following:

```
FUNCTION primesfrom(n)
IMPORTS (primes, limit)
LET rest BE primesfrom(n+2)
RESULT  if n > limit
          then nil()
          else par(
                     if relprime(primes)
                        then cons(n, rest)
                        else rest,
                     rest
                     )
```

In this example, the sub-expression **primesfrom**(n+2) is demanded concurrently with the testing of **relprime**(n, primes), so that the latter does not cause the generation of the sequence of primes to be sequentialized. Since common sub-expressions are identified as the same value, the same value of **primesfrom**(n+2) will be used in evaluating the if.... then.... else.... No recomputation will take place.

TARGET MACHINE LANGUAGE

As mentioned previously, our ultimate motivation for the FGL evaluator is its realization on the highly parallel machine architecture AMPS. While the physical details of such a machine are not relevant here, its language ML and execution semantics are. Hence we include here a brief sketch of these aspects.

The machine consists of a large number of identical processing elements (PEs), each possessing a portion of a uniformly-addressed, but physically distributed, memory. The fundamental observable action in a PE is a task, involving bounded space and time behavior, such as the execution of an atomic function or the propagation of a value instance or a demand. Parallelism is achieved by exporting, to neighboring processors, function application tasks which have been spawned by strict operators. Unlike the proposal of [7], no "sergeant" tasks are generated for computations which might not be required. However, the programmer may include functions, such as **par** in the preceding example, which cause such tasks to be generated. Further aspects of resource control in FGL are discussed in [17].

Unlike FGL, not every interconnection of ML operators is a valid program. For example, it is possible to construct incorrect linkages. However, the compiler insures that only valid ML programs are generated from their FGL inputs. We have insufficient space to include a presentation of what is or is not valid in ML.

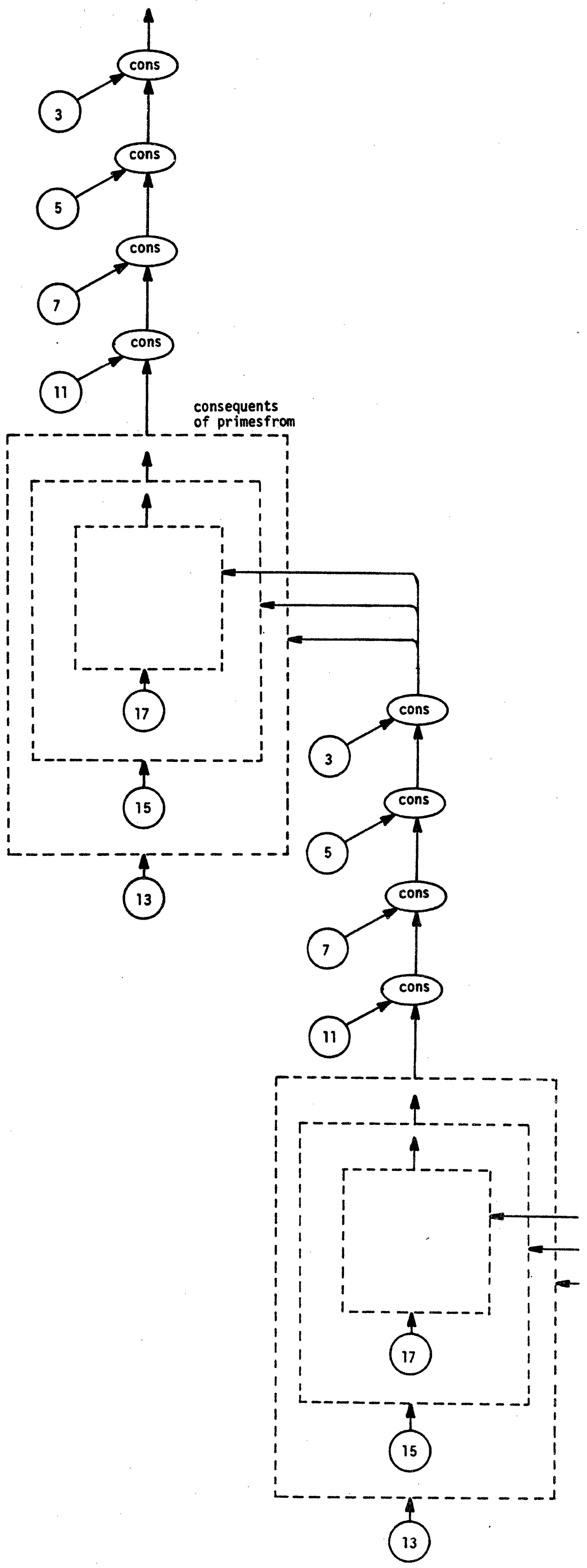


Figure 3: Dag form of the example in Figure 2.

Each programmer-defined function is represented (in pure code) as a block encoding of its graph. The code inside a block has roughly one word corresponding to each node. A typical code word contains the name of the node's operator, local (relative) addresses representing the node's arguments, and space for local notifiers (addresses used to tell which other nodes are to be informed when the node's value is ready).

The action corresponding to application of an FGL production is triggered as each instance of the antecedent is demanded. This action entails the allocation of a block into which the encoded graph is copied, and the linking the arguments and imports of that block with the block containing the antecedent, in effect splicing the graph represented by the code in place of the antecedent itself.

Evaluation of a node entails overlaying the node with its result. Of course, its notifiers are first temporarily saved by the processor, as they occupy some of the space required by the result itself. Here we see a contrast in that FGL values are viewed as appearing on the arcs, whereas ML values appear as transformed nodes. A more important contrast is that FGL objects can be infinite, whereas ML objects must each fit into bounded space.

With these considerations in mind, the FGL model must be refined toward the target machine representation so that fixed word and block sizes are possible. In particular:

1. Because of their disparity in size and meaning, local addresses cannot be freely converted to global addresses and vice-versa. Instead, special operators are provided at compile time to interface from one block to another.

2. While arcs within a block have statically bounded fan-out, global arcs can experience unbounded fan-out (e.g. due to multiple remote demands on a given tuple component's value).

3. In distributing values according to (2), the ML evaluator should not create new nodes (words) to mediate dynamic fan-out, lest storage management become more complicated.

4. The ML evaluator evaluates tasks using a task list which is generally distributed over the available processing elements. This list is used to determine an ordering of the application of transitions. Not all properties of the ordering are important. It only matters that once a transition rule is eligible for application, it does eventually get applied. This effect is achieved by FIFO queuing in ML, and finite-delay is the corresponding property in IGL.

As remarked above, ML code blocks use small relative addresses to express the local connectivity within a function graph. Global addresses are used to represent objects referenceable across code block boundaries. These include references to function definitions (pure code), function closures, function applications (for passage of parameters, globals, and result), and tuple values. In ML execution diagrams, e.g. Figure 7 global addresses will be represented by arcs with hyphenated lines.

The principal operators involving global addresses are **forward** and **fetch**. The operator **forward** connects a local argument (e.g. a function result value) to a global demander (e.g. its place of application). The operator **fetch** does the complementary action. It may be noted in each step that global address arcs only emanate from **forward** nodes, and that no new nodes are created in any step. Thus the creation of global pointers and the use of existing code space is well-disciplined.

Task list: a

Block contents:

address	operator	operands	notifiers
1	a	2 3	....
2	b	4	
3	c	5	
4	d	....	
5	e	....	
⋮		⋮	

$\Rightarrow$

Task list: b, c

Block contents:

address	operator	operands	notifiers
1	a	2 3	....
2	b	4	1
3	c	5	1
4	d	....	
5	e	....	
⋮		⋮	

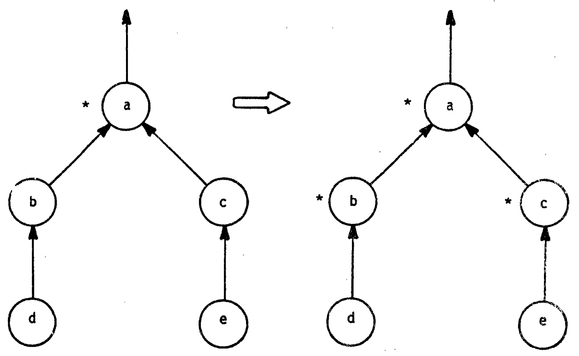


Figure 4: Example of execution in ML and the corresponding IGL transition.

INTERMEDIATE GRAPHICAL LANGUAGE

In attempting to prove that ML is a valid implementation of FGL, the disparity between the two languages seems best approached by the introduction of a third graphical language, IGL. The data objects of IGL are close to those of FGL, except that they use references, whereas FGL avoids references in favor of objects with more mathematical elegance.

The IGL objects are:

1. atoms, as in FGL
2. ?, the undefined value
3. error, the error value
4. references, of one of two kinds:
   a. tuprefs, references to tuples
   b. coderefs, references to master copies of code blocks
   c. funrefs, references to function closures (i.e. pairs consisting of a coderef and a tuple of imported values)
5. tuples of IGL objects of the above types only. (Tuples with tuples as components are not allowed. These must be provided by references.)

Unlike FGL, IGL objects must be finite. There are no limit objects. Instead, limit objects are implicitly represented by fixed points of equations, as will be described presently.

ML and IGL have the same data objects in common, but ML is more restricted in the way it can handle those objects, and includes the special linkage operators mentioned in the previous section. Another common characteristic between ML and IGL is that both are viewed as replacing the operator nodes with a value, whereas FGL is viewed as producing a value on an arc. Hence, we introduce the intermediate language to provide a convenient link between a very mathematical language on the one hand and a very pragmatic language on the other. Table I summarizes the differences between FGL, IGL, and ML. Like FGL, each arc in an IGL graph determines (again, by fixed point semantics) a data object. However, we need to progress toward the operationally defined ML. Hence we must at this point give an alternate, operational, definition of IGL which relates to its denotational definition in an obvious way. Accordingly, we choose to think of the nodes of an IGL graph as having values, which are identifiable as the same values determined on their (single) output arcs. Operationally, an IGL node will ultimately be replaced with that value if there is a demand for it. Another way of viewing this replacement is that the function in the IGL node is changed to a constant function having that value.

The precise operational behavior of our IGL evaluator, as well as its correctness with respect to the denotational semantics, will be approached in terms of "marked IGL graphs", which reflect demand and data flow in a manner similar to ML.

A marked IGL graph is an IGL graph in which each node is either marked *, for demanded or unmarked. Marked IGL graphs are the states of an abstract state transition system (cf. [12]) which models the flow of demand and values among nodes. The transitions in this system are based on transition rules for each of the node operators, as determined by the type of these operators.

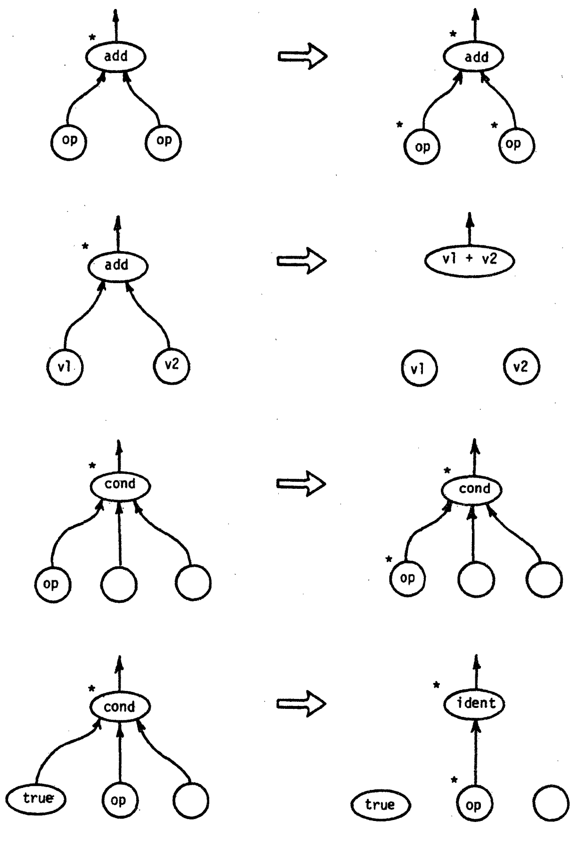


Figure 5: Transitions between marked IGL graphs.

	FGL	IGL	ML
Values manifest	on arcs	on arcs, or replacing nodes	replacing nodes
Infinite values	allowed	not allowed	not allowed
Fan-out	arbitrary	bounded	bounded
Linkages	implicit in productions	tuples and selectors	tuples and selectors converted to fetches and forwards

Table I: Comparison of the three language levels

We list in Figure 5 some of the rules in terms of markings. An evaluator becomes completely specified when the transition rules are accompanied by a specific order for their application. However, in a distributed system, this order will be difficult to control. Thus, instead of giving a rigid order, we assume for IGL only a finite-delay property: A rule cannot remain applicable forever without being applied by the evaluator. This property is insured by the ML realization, as will be later sketched.

IGL TO FGL MAPPING

The use of IGL as a conceptual "implementation" of FGL is achieved through the mathematical device of a mapping from the data values and operators of IGL to those of FGL. As mentioned previously, the main distinction to be drawn between FGL and IGL lies in the data types. Whereas the FGL data types are based purely on mathematical structure, IGL introduces objects which refer to parts of the graph to aid in the progression toward ML.

Another distinction between FGL and IGL of a more technical nature is that the arguments and imports to function objects in FGL are achieved simply by splicing the appropriate arcs together. In IGL, this effect is created by packaging into separate tuples the arguments and imports. These tuples reside in the applying block and the environment block, respectively. Selectors are used inside the applied block which accesses the tuples.

We have already discussed how a unique FGL object is determined on each arc of an FGL program, given that each of its input arcs have been assigned values. In the context of such an input assignment, if x is an arc, then we denote the determined value by Fval(x). In a similar way, a unique IGL object is determined on each arc of an IGL program, and we denote this value by Ival(x).

The IGL program graph gives rise to a system of FGL equations whose least fixed point defines, for each IGL object x, a corresponding FGL object h(x) as follows:

1. If x is ?, error, or atom then h(x) = x.

2. If x is a tupref, referring to

   $$(x_1, x_2, \dots, x_n),$$

   then $h(x) = \text{cons}(h(x_1), h(x_2), \dots, h(x_n))$.

3. If x is a funref, then h(x) is the function graph referenced by x, together with bound import arcs as determined by the tuple part of the referenced closure.

Each arc of the FGL graph can be identified as a unique arc of the IGL graph. Since IGL has additional operators for linkage, the converse is not true.

The link between the partial correctness of IGL and that of FGL may now be stated in terms of an equation involving the mapping h.

IGL-->FGL mapping theorem: For any arc x of an FGL graph,

$$\text{Fval}(x) = h(\text{Ival}(x))$$

To prove this theorem, we need only observe that h is a homomorphism from the space of IGL functions and domain to the corresponding FGL space. Here we may rely on the dag forms of the corresponding IGL and FGL programs. The technique is essentially that explained by [18]. [11] presents a similar theorem, stated in terms of a "semantic memory" instead of FGL arc values.

Since it is generally meaningless to speak of an evaluator producing a full FGL object, we phrase our definition of evaluator correctness in terms of IGL objects, as follows:

IGL Partial Correctness Theorem: If q is a state and x an arc marked demanded in q, and $\text{Ival}(x) \neq ?$, then the IGL evaluator can reach a state q' such that x is marked with its corresponding IGL value.

To justify this theorem, we identify node x as the node having x as its output arc. Consider the corresponding dag structure of the IGL graph with root node x, assuming now that $\text{Ival}(x) \neq ?$. Then either node x is a constant function having value Ival(x), or x produces Ival(x) based on the values of its arguments. In the first case, one transition rule gives us the desired result. In the second case, the inductive assumption is that the arguments evaluate appropriately so that evaluating the function in node x gives the desired result. Thus, the inductive conclusion tells us that these arguments can be produced by application of the transition rules. Therefore application of one or more transition rules for the root node will produce Ival(x).

The above use of induction is technically justified from the continuity of IGL operators. Informally, this says that a finite value (e.g. any IGL value) producible from an arbitrary composition of operators is also producible from a finite truncation of that composition. For a further discussion of such uses of continuity, see [19], [20], or [16].

Given this partial correctness, we have the corollary that any finite piece of an IGL value can be produced by an appropriate set of demands. Simply affix to the arc in question a supplementary function graph of selectors which evaluate to that piece formally, then apply the above criterion to the output of the supplementary graph.

By assuming that the underlying IGL evaluator has the finite-delay property, the "can" above effectively becomes "will". This property is provided in the definition of ML. This approach is necessary since there is no mechanism for insuring the finite-delay property within IGL itself.

In the next section, we appeal to ML to provide the necessary infrastructure for total correctness of the IGL evaluator.

ML TO IGL MAPPING

As stated earlier, ML and IGL have the same data objects. As the corresponding ML-->IGL mapping is rather trivial, involving only replacement of linkage operators by identities, it will not be elaborated upon here. Furthermore, both IGL and ML are evaluated by changing the operations of their nodes into values. In ML however, we provide an implementation of demand/value propagation symbolized by markings in IGL.

In IGL, the presence of a demand for a node's value is indicated by marking the node with an asterisk. It would be infeasible, in ML, to search the memory for all demanded nodes each time a new value is computed. Instead, ML employs a task list structure which contains pointers to all nodes on the wavefront of demand/value propagation (see Figure 6). The wavefront may be thought of as initially propagating in the direction opposite to the argument arrows and being reflected in the opposite direction when computed values are encountered.

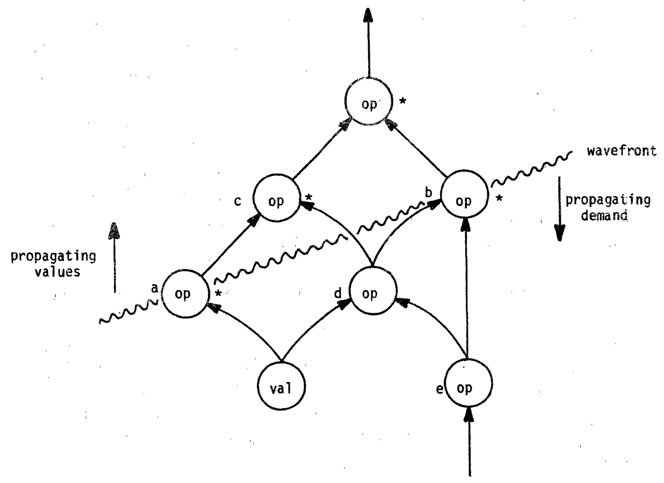


Figure 6: Wavefront of demand/value propagation. Nodes a and b are currently on the task list. a will be evaluated and notify c; b will propagate its demand to d and e.

For demanded nodes not on this wavefront, the fact that the node has been demanded is recorded by the presence of a notifier or a forward pointer (see next section) in some other demanded node. Thus, consider the following definition of a set of nodes S:

1. Nodes on the task list which do not yet have a value are in S.(a)
2. If x is a node in S, and x contains a notifier or forward pointer to node y, then y is in S.
3. All nodes in S are there because of one of the above reasons.

Wavefront Lemma: S consists of exactly the demanded nodes.

The proof of the above lemma is by transition induction (cf. [12]) on the ML transition rules. All initially demanded ML nodes are externally placed on the task list. A case analysis of the ML transition rules reveals that any newly demanded node is put in S. Similarly, any node which is replaced with its value cannot remain in S, but nodes requiring that value are put in S.

We repeat that the finite-delay property for IGL means that every demanded node in a given state, if entitled to eventually receive a value (because the IGL output value of that node is not ?), will receive a value. As is well known, FIFO processing of nodes in a directed graph gives rise to breadth-first visitation of the nodes, i.e. the wavefront effect. By processing the task last in FIFO order, it is clear that any node in need of attention eventually receives that attention. In particular, every node gets attention when it is first demanded, and when it is able to compute its value.

In the proposed AMPS architecture, the task list is not monolithic, but instead is distributed among many processing elements. However, each of the segments is processed in FIFO order, so the same wavefront effect is obtained.

PRAGMATIC ASPECTS

Although not required for correctness as stated, parsimonious evaluation is also achieved. That is, each node is evaluated at most once, since the presence of a notifier inhibits potential secondary demand propagation. This idea, applied to the cons operator, was called "suicidal suspension" in [10]. It has also been used in operating systems (e.g. the dynamic linking mechanism of Multics) for some time. Our evaluator includes this technique for all operators.

ML includes additional operators apart from IGL, namely the special operators used to control data flow across block boundaries. Specifically, whenever a selector in one block refers to a tuple in another, the selector is replaced with the special fetch operator which matches a forward operator in the tuple component. The fetch operator contains the global address of the forward. A demand of the fetch (which occurs

(a) Because of some redundancy in the evaluator, a node could be on the task list and have a value. For example, it could be notified by two different nodes, and become evaluated before the second notification "takes effect".

automatically when the selector is demanded) is then propagated to the **forward**, which propagates the demand to another operator local to its block. At the same time, a forward pointer back to the **fetch** is set to point to the **forward**, so that when the demand is satisfied, the forward will know where to send the result. A **fetch/forward** pair is also used to pass the result of the block to its destination.

A possible alternative to forward chaining is to use "busy waiting". That is, the second and subsequent fetches for the same value are simply re-cycled back to the task list to be re-tried again and again. This solution is viewed as unacceptable, as the wait can be arbitrarily long.

As described thus far, the ML **fetch/forward** pairs resemble identity functions which carry out the linkage needed to implement an arc crossing blocks in IGL. However, a complication arises when there is more than one demand on the same component of a tuple. This complication was not mentioned in [10] where it does not occur because evaluation is sequential, but neither was it mentioned in [7]. The property asserted there of the existence of at most one reference to any "suspension" seems infeasible for a parallel evaluator, as we now discuss.

Since the number of demands may, in principle, be arbitrary, there is no fixed word size which can accommodate sufficiently many forward pointers. Hence a scheme called forward chaining is used. This scheme maintains the invariant (provable by transition induction) that at most one forward pointer is ever stored in a given **forward** node. This is accomplished by having each additional fetch to the same **forward** operator assume the responsibility for forwarding to the location to which the **forward** pointer pointed, while the forward operator then points to the most recent fetch only. The handling of **fetch** and **forward** in ML is demonstrated in Figure 7.

Although there is no limit on the number of (local) notifiers a node may entail, the number actually needed in each case can be detected at compile time. Hence it is possible for the compiler to cascade extra identity operators in such a way that the number of notifiers for each node does not exceed the maximum pragmatically allowed.

CONCLUSIONS

We have described some considerations which arise in the evaluation of an applicative language in a manner capable of exploiting a multiplicity of physical processing elements. The present exposition focuses on the analysis of a hardware evaluator for the AMPS system. In addition to the graphically-represented extrinsic language and machine language, an intermediate graphical language has been introduced, to separate questions of value flow from more pragmatic issues of communication and demand flow.

The important aspects of this work thus concern the distributed evaluator itself, the analysis techniques, the graphical models, the formalization of demand-driven computation and accompanying correctness criterion, and further technical exposition of machine evaluation of unbounded data objects.

We view this analysis as a step toward a proof for a fuller system in which a reference-counting storage manager is implemented (cf. [21]), as well as other language and pragmatic issues, such as shared resource management and load control [17].

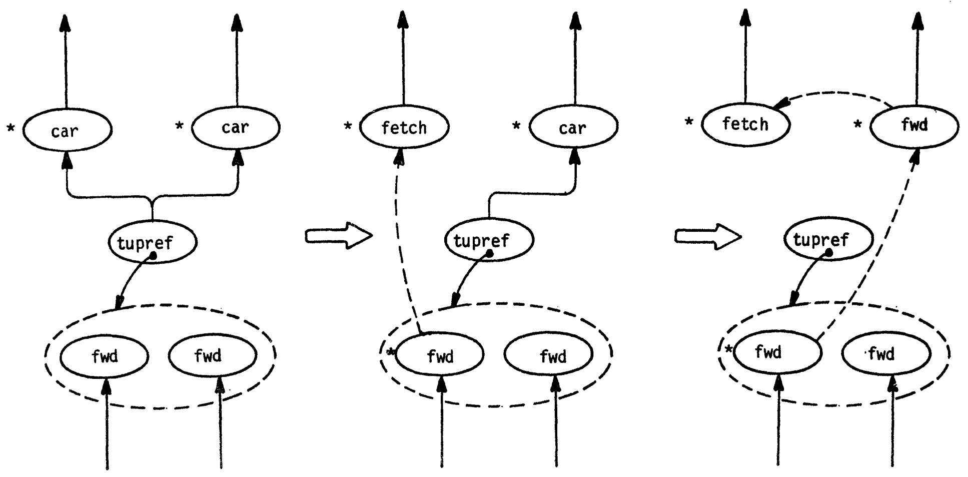


Figure 7: Example of forward chaining in ML.
Hyphenated arcs denoted global addresses.

REFERENCES

[1] G. Brown. A new concept in programming. in M. Greenberger (ed.), Management and the computer of the future. Wiley (1962).

[2] S. Patil. An abstract parallel-processing system. M.S. Thesis, MIT Dept. of Electrical Engineering (June 196).

[3] L.L. Constantine. Control of sequence and parallelism modular programs. AFIPS Proc., 409-414 (Spring 1968).

[4] L.G. Tesler and H.J. Enea. A language design for concurrent processes. AFIPS Proc., 403-408 (Spring 1968).

[5] D.A. Adams. A model for parallel computations. in Parallel processor systems, technologies, and applications. Spartan Books, 311-333 (190).

[6] W.H. Burge. Recursive programming techniques. Addison-Wesley (195).

[7] D.P. Friedman and D.S. Wise. The impact of applicative programming on multiprocessing. IEEE Trans. on Computers, C-2, 4, 289-296 (April 198).

[8] R.M. Keller, G. Lindstrom, and S. Patil. A loosely-coupled applicative multi-processing system. AFIPS Proc. (June 199).

[9] R.M. Keller, B. Jayaraman, G. Lindstrom, J.B. Marti, A.K. Nori, and D. Rose. FGL Programmers' Guide. Unpublished manuscript, University of Utah (March 1980).

[10] D.P. Friedman and D.S. Wise. CONS should not evaluate its arguments. in Michaelson and Milner (eds.), Automata, Languages, and Programming, 25-284, Edinburgh University Press (196).

[11] P. Henderson and J.H. Morris, Jr. A lazy evaluator. Proc. Third ACM Conference on Principles of Programming Languages, 95-103 (196).

[12] R.M. Keller. Formal verification of parallel programs. CACM, 19, , 31-384 (July 196).

[13] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, I. CACM, 5, 2-3 (190).

[14] A. Church. The calculi of lambda-conversion. Princeton University Press (1941).

[15] R.M. Keller. Divide and CONCer: Data structuring for applicative multiprocessing systems. to appear in Proc. 1980 Lisp Conference.

[16] R.M. Keller. Semantics and applications of function graphs. Manuscript (March 1980).

[17] B. Jayaraman and R.M. Keller. Resource control in a demand-driven data-flow model. Proc. International Conference on Parallel Processing (Aug. 1980).

[18] C.A.R. Hoare. Proof of correctness of data representations. Acta Informatica, 1, 21-281 (192).

[19] G. Kahn. The semantics of a simple language for parallel programming. Proc. IFIP '74, 41-45 (194).

[20] J. Stoy. The Scott-Strachey approach to the mathematical semantics of programming languages. MIT Press (19).

[21] A.K. Nori. A storage reclamation scheme for AMPS. M.S. Thesis, Dept. of Computer Science, University of Utah (Dec. 199).

[22] M. O'Donnell. Computing in systems described by equations. Lecture Notes in Computer Science, 58 (19).

This material is based upon work supported by the National Science Foundation under grants MCS 77-09369 A01 and MCS 78-03832.

SPECIFICATION AND SYNTHESIS OF SYNCHRONIZERS

Krithivasan Ramamritham
Robert M. Keller
Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

Abstract -- Presented is a specification language for expressing properties required among operations accessing shared resources in a concurrent environment. Such constraints are necessary in order to maintain the integrity of resources. The language is founded on Temporal Logic and possesses constructs for expressing, in a natural manner, properties such as mutual exclusion of operation execution, priority among operations, invariance of resource state, and scheduling disciplines. Each of the above properties is expressed independently of the others resulting in modular specifications. An algorithm is outlined for systematically synthesizing code for a synchronizer from the given specifications. Synthesis is achieved by successive transformation of the specifications into target language code. Feasibility of the specification and synthesis technique is demonstrated by applying it to a standard synchronization problem.

INTRODUCTION

Two main approaches exist for the development of any provably-correct software system. The first involves construction of programs, followed by a posteriori verification that the program meets intended specifications. In this case, the specifications themselves often provide a *descriptive* role rather than a prescriptive one, since there are many different sets of specifications which can be met by a given program. The second approach involves automatic synthesis of a program directly from the specification. Here the specifications must be sufficiently *prescriptive* to enable a synthesis to be carried out.

This material is based upon work supported by the National Science Foundation under Grant MCS-77-09369 A01.

The advantages of the synthetic approach are therefore that the tedious task of a posteriori verification is eliminated and the specification is required to be sufficiently free of ambiguity. The disadvantages are that synthesis algorithms are difficult to devise, such algorithms themselves must be verified (but this is a one-time cost), and the results of a synthesis algorithm sometimes have less efficiency than desired.

In this paper, we suggest some principles for construction of a specification language and an accompanying automatic program synthesis system for synchronizer code. A system of this type would accept specifications that characterize the synchronization problem to be solved and generates a program that conforms to the problem description. The solution proposed uses temporal logic as the basis for the semantics of the synthesis system. Our approach consists of:

1) Designing a rich class of primitives and constructs for a high-level language in which synchronization properties can be expressed unambiguously in a non-procedural form.

2) Devising a methodology for algorithmic translation of specified properties into appropriate target language code for a synchronizer.

The temporal approach to specification and implementation of synchronization carries with it the advantages of a unified approach. When we refer to ordering of operations, scheduling discipline etc., the underlying concept is temporal ordering. Thus it is appropriate to adopt a system of reasoning based on temporal logic for expressing the semantics of synchronization of concurrent processes.

Since the reliability of programs that share resources depends upon the correctness of the underlying synchronizer, it is highly desirable that the synchronizer construction be as reliable as possible. Automating the synthesis of synchronizers is proposed as a technique which will aid in the development of reliable programs.

The Specification Language

The approach taken is to systematize and abstract features of synchronization control into a set of language constructs based on Temporal Logic which provides an excellent natural tool to express both invariant and time-dependent properties of software systems [15]. Current specification techniques do not handle both types of properties as uniformly as the temporal approach does. Use of temporal constructs such as 'henceforth', 'eventually' and 'until', along with the constructs derivable from them, result in intuitive specifications for synchronization problems.

The specification language satisfies the following criteria:
-- It facilitates expression of the complete semantics of a system of concurrent processes, providing constructs for specifying constraints, invariants and other behavioral aspects.
-- It is modular and easy to apply.

The language constructs are able to independently express, properties such as scheduling constraints, priority of operations, mutual exclusion of operations, invariance of resource state, absence of starvation and other relevant properties. Each construct has an appropriate formal temporal semantics. Language features such as arrays of operations and macro notation can be used to enhance the readability and succinctness of the specifications.

Another aspect of synchronizer behavior desired in the final implementation of most schemes is 'fairness'. Our specification language provides for expressing a fairness criterion appropriate for the problem under consideration.

The Synthesis Algorithm

Given the specification of the desired behavior of a synchronizer of operations, the second goal is to develop an algorithm for automatically synthesizing a synchronizer in a prespecified target language. The synthesis algorithm successively transforms the specification statements, applying appropriate meaning-preserving transformation rules, each step bringing the resulting statements closer to the target language code. The transformation is complete when the derived statements can be mapped directly into primitives of the target language. Also, the resulting synchronizer will display the desired fairness. By requiring the derived statements to retain the semantics of the top-level specification, the resulting synchronizer need not be verified for correctness. Instead, only the synthesizing algorithm need be verified.

By providing a specification language based on temporal logic, and a synthesis algorithm that guarantees the validity of the specified properties, this work will contribute towards better specification techniques, and construction of reliable software for concurrent systems.

The paper continues with a presentation of the synchronization model. The specification language and the synthesis algorithm are then developed. Discussion of related work precedes concluding remarks on the proposed approach.

THE SYNCHRONIZATION MODEL

To maintain the integrity of a shared resource, an answer to the question, "Who is to access the resource, when, and how?", is essential. A protection mechanism is responsible for who accesses the resource and how the resource is accessed. On the other hand, the synchronizer is responsible for when the access actually takes place. In this paper, we shall be concerned with the problems of synchronization.

A synchronizer, in our model, is a centralized sequential process that guarantees disciplined access to shared resources. Access to the shared resource is through specific operations, the execution of which is controlled by the synchronizer. Constraints essential for maintaining the integrity of the resource are enforced by the synchronizer. Concurrent processes access the shared resource by requesting execution of any of the specified operations. A request for an operation on a shared resource is serviced by the synchronizer after ensuring that the constraints are not violated. A serviced request becomes active when it is executed by either the synchronizer or, on its behalf, by another process.

A requested operation may be thought of as being in one of three states:

1. Active -- Currently executing.
2. Enabled -- Can be serviced without infringing some constraint.
3. Disabled -- Cannot be serviced without infringing some constraint.

Two or more processes are said to be in conflict if they are simultaneously enabled. Conflict resolution occurs when the synchronizer services one of the enabled operations, based on a specified scheduling discipline or priority.

The model assumes that

1. Arrival of a request is synonymous with recognition of its presence by the synchronizer.
2. Once an operation is enabled, it will be serviced after a finite amount of time, unless it is meanwhile disabled by the servicing of some other operation, as in the case of conflict resolution in favor of some other operation.
3. There may be a finite delay between servicing a request and its subsequent activation. The synchronizer services no other operation until the serviced operation becomes active.

4. An active process cannot be aborted or interrupted by the synchronizer.

5. An operation remains active for a finite but indefinite period of time, after which it is said to have terminated.

These assumptions are formalized in the next section after the introduction of the language constructs. They do not introduce any major restrictions on the class of synchronization problems that can be solved, but are motivated by a desire to achieve a suitable abstraction of the notion of synchronization. Many specific synchronization primitives fit this abstraction.

THE SPECIFICATION LANGUAGE

In our language, specifications are statements in first-order predicate calculus augmented with temporal operators, as introduced presently. The underlying semantics of the language is based on a computational model involving the notion of events and conditions [10]. In this model, the effect of concurrent execution of processes is considered to be the enabling and disabling of certain conditions during the execution process. The choice of conditions reflects those aspects of the system of parallel processes in which we are interested, viz. synchronized access to shared resources. Events do not appear in the specifications, only conditions do.

We begin with a description of the primitives used in the specification language.

Language Primitives

Every pending operation has four primitive conditions associated with it having the following semantics.

req(a)	There is a request for operation 'a'. This condition becomes True when a concurrent process requests operation 'a'.
start(a)	Operation 'a' is permitted to execute (The permission is irrevocable). This condition becomes True when the synchronizer services request 'a'.
exec(a)	Operation 'a' is executing now. This condition is True when operation 'a' is active.
term(a)	Execution of operation 'a' has terminated. This condition becomes True when operation 'a' terminates.

We refer to each distinct type of operation on a shared resource as an operation class. All operations of a particular type are said to be instances of that operation class. In the above definitions, 'a' stands for a specific instance of a particular operation class.

We will now introduce the temporal operators along with their semantics. These are strongly influenced by [12], [15] and [17].

[]C	To be read 'always C'. This means, condition C will remain true from now on, i.e., C is true now and throughout the future.
<>C	To be read 'eventually C'. This means, condition C will eventually become true, i.e., C will be true sometime in the future.
A UNTIL B	To be read as "A remains true until B becomes true". This means, if B eventually becomes true, then A remains true from now until B becomes true; otherwise []A.

Statements that do not involve the temporal operators are considered to be about the present, or 'now'. In general, statements in the language will involve the predicate logic operators: V(or), &(and) ~(not) and =>(implies)[b] in addition to the temporal logic operators.

Given below are the axioms and inference rules that form the temporal logic system[c]. A and B are arbitrary Temporal Logic formulas.

Axioms:

```
[]A          => A & <>A & [][]A
<><>A        => <>A
[](A => B)   => ([]A =>  []B)
A UNTIL B & ~B UNTIL C =>  A UNTIL C
[](A => B) <=> [](A => (B UNTIL ~A))
```

Inference Rules:

```
If A is a valid first-order logic formula
                                then |- A.
If |- A and |- (A => B) then |- B
If |- A then |- []A
```

Certain temporal operators are derived from these primitives, and are introduced to enhance the readability of the specification language. They are,

P ONLYIF Q	(P => Q) i.e., P is true only if Q is True.
P IFF Q	(P => Q) & (Q => P).
P ONLYAFTER Q	(~P UNTIL Q) i.e., P can become True only after Q does.
P AFTER Q	[(~P UNTIL Q) & <>P] i.e., P will become True after Q.

---

[b]The operator precedence is ~, {<>,[]}, {V,&}, UNTIL, followed by =>.

[c]The choice of axioms and inference rules listed here is based upon their utility in subsequent sections. No claim is made for their completeness.

P CAUSES Q	(P => <>Q) & $\nexists$ R {(R $\neq$ P) & (R $\neq$ Q) & (P => <>R) & (Q AFTER R)} i.e., P is the sole cause for Q to become True.

where P and Q are arbitrary conditions.
The following are true for a particular operation 'a'.

```
req(a)   =>  [req(a) UNTIL exec(a)]
start(a) ONLYIF req(a)
start(a) =>  [start(a) UNTIL exec(a)]
start(a) CAUSES [exec(a) & ~start(a) & ~req(a)]
start(a) =>  [∀b≠a ~start(b) UNTIL ~start(a)]
~exec(a) =>  [~exec(a) UNTIL start(a)]
exec(a)  =>  [exec(a) UNTIL term(a)]
[term(a)&exec(a)]  CAUSES [~term(a)&~exec(a)]
```

These statements are the axioms formalizing the synchronization model.

Using the primitive conditions, we define the following:

req(a)[cond]	there exists a request for operation a satisfying 'cond', i.e., $\exists$a(req(a) & cond).
req$A	$\forall$aεA req(a), i.e., there exists a request of class A.
exec$A	$\forall$aεA exec(a), i.e., an operation of class A is active.

The Specification Statements

The temporal operators defined earlier serve as the building blocks for our specification language. The semantics of the various specification statements are given in terms of these temporal operators.

While developing our specification language, and the synthesis procedure, it will be instructive to consider a typical synchronization problem encountered in the context of operating systems. Although it is a simple example, it serves to illustrate the important aspects of the approach.

The Limited Resources Problem: [8]. A fixed number of similar resources is managed by an operating system. User processes acquire a resource by executing the operation 'acquire', and release the resource by the operation 'release'. The variable 'free' maintains the number of available resources, while 'max' gives the maximum number of resources in the pool. 'Release' is given priority over 'acquire'.

Operation variables The specification Language possesses features that result in succinct specifications. One of these, is the facility to refer to a class of operations using a generic operation name. Specifications involving this operation name apply to all operations in that class.

```
Specification OPERATIONS a:A;
                           S;
```

Semantics $\forall$aεA (S);
where S is a specification statement involving 'a' and applies to each operation in class A.

```
Example     OPERATION r: release;
                      a: acquire;
```

The above specification declares operation variables for the limited_resources problem, where 'r' refers to any 'release' operation, and 'a' refers to any 'acquire' operation.

Resource state Information During the active phase of an operation, the "state" of the shared resource may be altered. For instance, 'acquire' reduces the number of free resources. Scheduling constraints often involve predicates on the resource state. For instance, 'acquire' can be serviced only if there are free resources. The above discussion demonstrates the need for expressing the synchronization constraints that depend on resource state. This language facilitates the specification of the following aspects of the resource:

- The data structures that determine the resource state.
- Initial resource state.
- The modification to resource state by operations in each class, and
- Invariance of resource state.

Example
```
RESOURCE_STATE_INFORMATION
 STATE_VARIABLES ARE
  free : integer;
  max : constant integer <- 10;
 INITIALLY
  Free<-max;
 STATE_CHANGES
  Acquire: free<-free-1
  Release: free<-free+1;
 STATE_INVARIANCE
  0 ≤ free  ≤ max;
```

These statements specify resource state information needed for a synchronizer of 10 resources.

Scheduling Constraints Scheduling constraint specifications express the explicit conditions under which an operation can be "serviced". For example,

Specification
```
cond1@req(op_name) =>
        []{Start(op_name) ONLYIF cond2};
cond3@req(op_name) =>
        []{Start(op_name) ONLYAFTER cond4};
```

For an operation 'p', cond1@req(p) refers to the value of cond1 when the request for p arrives. In general, cond1 and cond3 are conditions dependent on resource_state, or arguments to the requested operation, or both. In the case of synchronized operations, 'req(op_name)' is a necessary constituent of cond2. If the 'cond1@req(op_name)' clause is not specified, then it is true by convention.

Example
```
SCHEDULING_CONSTRAINT
 []{Start(a) ONLYIF Req(a)};
 []{Start(r) ONLYIF Req(r)};
```

These specify the requirement that release and acquire operations should be serviced only if requests exist for them.

Invariance The invariance specifications express the constraints with regard to the resource state, in the following manner:

Specification : STATE_INVARIANCE I
Semantics : $[]I$
Example : STATE_INVARIANCE $0 \leq free \leq max$;

Exclusion of Operations In our specifications, concurrency is assumed to be the rule, and exclusion the exception. So when two operations are to exclude each other, there has to be a specification so stating.

Exclusion among operations in different classes
Specification : A EXCLUDES B
Semantics : $[]\sim(exec\$A \ \& \ exec\$B)$
$A,B \in$ {operation classes}[d]

Exclusion among operations in a class
Specification : A's EXCLUDE
Semantics : $[]\sim\{exec(a1) \ \& \ exec(a2)\}$
$\forall a1,a2 \in A$, A an operation class.

Total exclusion of all operations
Specification : EXCLUSION all
Semantics : I EXCLUDES J & I's EXCLUDE
$\forall I,J \in$ {operation class}.

Example
```
Acquire EXCLUDES Release;
Acquire's EXCLUDE;
Release's EXCLUDE;      or equivalently,

EXCLUSION all;
```

Priority among Operations We classify priority into the following two categories:
- Priority within requests of a particular operation class, otherwise known as intra class priority.

- Priority between different operation classes, otherwise known as inter class priority.

In general, both inter_class and intra_class priorities can depend on resource state. This dependence can be specified in this language through the use of 'resource_state_predicates'. A 'resource state predicate' is a predicate on the state of the resource and is said to be True if current resource state implies truth of the predicate.

[d] {operation classes} stands for the set of operation classes.

We will see how the priority statements ar specified, and give their temporal semantics.

Specification: INTRA_CLASS_PRIORITY
 operation_class:-
  resource_state_predicate: priority_rule
Informal semantics:
If "C:- r: expr" is a intra_class priority specification, then 'expr' gives the priority rule applicable to operations in class C when the resource state satisfies 'r'.
Formal Semantics:
I = {intra_class priority specification}
OP ε {operation class},
r ε {resource_state_predicate}
pr_rule is an arithmetic expression that evaluates to an integer.

$$(OP :- r : pr\_rule) \in I, \ \forall op_1, op_2 \in OP,$$
$$[]\{[r \ \& \ req(op_1) \ \& \ req(op_2) \ \& \ (expr|_{op_1} < expr|_{op_2})] \Rightarrow [Start(op_1) \text{ ONLYAFTER } Start(op_2)]\}$$

where $expr|_a$ stands for the value of expr evaluated in the context of req(a). This specification expresses the requirement that in a given class, operations with lower priority should start only after all other relevant requests with higher priority have started.

In the absence of an intra_class priority statement, order of arrival of requests determines the priority of operations in each class. This corresponds to an FCFS discipline.

Specification: INTER_CLASS_PRIORITY
 resource_state_predicate :
  operation_class_b > operation_class_a
Informal Semantics:
Given an inter_class priority statement "r: B > A", if current resource state satisfies r, then operations in class B have higher priority than those in class A.
Formal Semantics:
I = {inter_class_priority specification}
r ε {res_state_predicate}
OP = {operation class}

$$\forall opc_1, opc_2 \in OP, \ \forall r \in R,$$
$$(r : opc_2 > opc_1) \in I, \ \forall op_1 \in opc_1, \ \forall op_2 \in opc_2,$$
$$\{[][r \ \& \ req(op_1) \ \& \ req(op_2)] \Rightarrow [Start(op_1) \text{ ONLYAFTER } Start(op_2)]\}$$

As noted earlier, an operation (say p) is enabled, if its becoming active will not infringe specified scheduling constraints, mutual exclusion and invariance. This is written as 'enabled(p)', and its negation 'disabled(p)'. In the semantics above, priority was specified among requested operations. However, there are cases when only those operations which are enabled are to be considered for priority. We refer to this as 'priority among enabled operations'. In such cases, an operation can start only after enabled operations of higher priority have been serviced. Formal semantics in this case is obtained by substituting 'enabled(op)' for 'req(op)' in the above specifications.

Example In the limited resource problem, 'release' operations are given higher priority than 'acquire' operations. Because the number of resources is limited, priority based on requests may result in a deadlock. Hence we have

```
  INTER_CLASS_PRIORITY_AMONG_ENABLED_OPERATIONS
          release > acquire ;
```

Scheduling Discipline In case more than one operation is enabled, the synchronizer resolves the conflict using the priority or scheduling discipline specifications, and eventually services one of the enabled operations. Scheduling discipline statements specify "fair" behavior of the synchronizer, or in practice, what a user construes fairness to mean [16]. They effectively express behavior of the conflict resolution strategy. We say that a scheduler is fair if it conforms to the specified scheduling discipline. Possible versions of scheduling discipline are:

Scheduling Discipline 0
An operation that is enabled is serviced, i.e.,

```
    Enabled(op) => <>Start(op)                    (SD0)
```

If 'op' is such that it can be disabled before the synchronizer recognizes that it is enabled, then SD0 will not be appropriate.

Scheduling Discipline 1
If we want to express the fact that enabling of an operation causes its start, then we have SD1, defined as follows.

```
    Enabled(op) CAUSES Start(op)                  (SD1)
```

This expresses the direct causality between enabling of an operation and its starting.

Scheduling Discipline 2
If an operation is going to remain enabled till it is serviced, then it will be serviced, i.e.,

```
    [enabled(op) & (enabled(op) until Start(op))]
                              => <>Start(op)      (SD2)
```

Scheduling Discipline 3
If an operation would otherwise be enabled infinitely often, then the operation is serviced.

```
  [enabled(op)    &    ({[]<>enabled(op)}    UNTIL
Start(op))]
                              => <>Start(op)      (SD3)
```

This will be suitable for operations that are not continuously enabled but are repeatedly enabled.

Scheduling Discipline 4
This type of fairness is based on the order of arrival of requests. The earliest to arrive will always be chosen for service. Formally, the expression

$\{[\mathrm{Req}(op_2) \text{ AFTER } \mathrm{Req}(op_1)]$
$\quad \Rightarrow [\mathrm{Start}(op_2) \text{ ONLYAFTER } \mathrm{Start}(op_1)]\}$
(SD4)

states that given that Request for op_2 arrived after that of op_1 then op_2 can be serviced onlyafter op_1.

Whenever priority specifications are applied, the following variation of SD2 is required.

```
{[enabled(op) & P] &
   [(enabled(op) & P) UNTIL Start(op)]}
                           => <>Start(op)   (SD2P)
```

Here P is a condition which holds iff priority specification is satisfied. This is necessitated by the sequential model assumed for the synchronizer and the fact that requests originate in external processes.

Overall Specification of the Limited Resources Problem Given below is the overall specification for the Limited_Resources problem. Note that the specification for the problem is obtained by conjoining individual specifications.

```
SYNCHRONIZER Limited_Resources IS
OPERATION_CLASSES acquire,release;
OPERATIONS a:acquire; r:release;
SCHEDULING_CONSTRAINT
 Start(a) ONLYIF Req(a);
 Start(r) ONLYIF Req(r);
RESOURCE_STATE_INFORMATION
 STATE_VARIABLES ARE
  free : integer;
  max : constant integer <- 10;
 INITIALLY
  free<-max;
 STATE_CHANGE
  acquire: free<-free-1;
  release: free<-free+1;
 STATE_INVARIANCE
  0 < free  < max;
EXCLUSION all;
INTER_CLASS_PRIORITY_AMONG_ENABLED_OPERATIONS
        release > acquire ;
SCHEDULING_DISCIPLINE     SD2P;
END limited_resources;
```

This example illustrates the salient features of the language. The fact that each distinct property of the limited_resources problem was specified independent of the rest attests to the modularity, and extensibility of specifications in the language. Using the top-level constructs, we have been able to specify standard synchronization problems including different versions of readers-writers problems [4], and disk-scheduler problems incorporating priority [9].

THE SYNTHESIS ALGORITHM

Given the Top Level specification of required synchronization, we propose an algorithm which derives in stages, synchronization code (in a prespecified target language) which will achieve the required synchronization. Synthesis is achieved by a series of transformations from the top-level specifications until a stage is reached when statements can be directly translated into primitives in the target language. The transformation is carried out in a target-independent fashion until that stage. We pursue the example of limited_resources synchronization to exemplify the synthesis steps. To keep this presentation managable, only transformations required for constructs in the example will be discussed here.

Effecting Resource state changes

In this step, all changes to resource state by the synchronized operations are "mirrored" within the synchronizer in the following manner: For each resource_state variable, a "synchronizer variable" local to the synchronizer is created with the same type and initial value. The synchronizer mirrors a resource state change (effected by a serviced operation) by addition of statements of the form

```
     start(op) CAUSES caused_action;        (Rule1)
```

where caused_action modifies 'synchronizer variables'. These modifications correspond to resource state changes specified for operation 'op' when executed in exclusion.[e] The semantics of caused_action is obtained from the 'Resource_state_change' statements. All specification statements that involve resource state variables are respecified in terms of the synchronizer variables.

Example: Retaining the names of the resource state variables as in the specifications but making them local to the synchronizer, the resource state changes will be mirrored by the following statements derived using Rule1.

```
Start(a) CAUSES (free <- free-1);
Start(r) CAUSES (free <- free+1);
```

Every future resource state modification by a serviced operation is faithfully reflected by the synchronizer variables. Hence this step is a meaning-preserving transformation.

The next step in the transformation process is to derive necessary conditions for servicing a request, i.e. starting an operation. These are embedded in the Scheduling Constraint, Mutual Exclusion, and Resource state Invariance specifications. Deriving the necessary conditions from these statements is the subject of the following discussion.

Transforming the Mutual Exclusion Statement

The specification 'A excludes B' is transformed into

```
Start(a) =>  ~Exec$B
Start(b) =>  ~Exec$A                          (Rule2a)
```

where a is an operation in class A and b in B.

The case of exclusion of different instances of the same operation class A translates to the intermediate specification

```
Start(a) => ~Exec$A                          (Rule 2b)
```

Since an operation in a class is serviced only if no there are no active operations belonging to classes which exclude it, mutual exclusion is guaranteed.

[e] Since operations that change resource state need execute in exclusion, this transformation is appropriate.

Achieving Resource state invariance

Resource state changes as mirrored within the synchronizer are 'caused' by the Start of an operation. Invariance will be maintained by ensuring that the invariant will not be falsified by the action. This can be done by deriving a precondition (e.g., by the backward substitution technique of program verification [14]) for the synchronizer action from the invariance specification and, the semantics of changes to the resource state by the operation. An operation is enabled only when the precondition is True. For example, if we had the following specifications,

```
Start(op) ONLYIF Cond (op)
Start(op) CAUSES caused_action(op),  and
Invariance : INV
```

then the transformed specification will be

```
Start(op) ONLYIF cond(op) & precond        (Rule3)
```

where 'precond' has to be true when start occurs in order for the invariant to be true after the "caused_action". Thus Rule3 preserves specified invariance of resource state.

Example: The precondition for release is derived to be "free<max" and for acquire it is "free>0". Using rule3 we arrive at the following statements.

```
Start(r) => free<max & req(r);
Start(a) => free>0 & req(a);
```

When invariance, scheduling constraint and mutual exclusion statements have been transformed, we can have for each synchronized operation 'op', statements of the form

```
<cond1>@req(op) =>
  []{Start(op) ONLYIF necessary_condition(op)};

<cond3>@req(op) =>
  []{Start(op) ONLYAFTER cond4};

   Start(op) CAUSES caused_action(op);
```

where necessary_condition(op) is the conjunction of all necessary conditions for 'op' to be enabled. Scheduling discipline and priority statements are inherited from the top-level specifications.

Example: In the limited resources problem, conjoining all necessary conditions for acquire and release respectively, we derive the following :

```
Start(a) => {~Exec$acquire & ~Exec$Release &
                             free>0 & req(a)};
Start(r) => {~Exec$acquire & ~Exec$Release &
                             free<max & req(r)};
Start(a) CAUSES (free <- free-1);
Start(r) CAUSES (free <- free+1);
```

Priority and scheduling discipline specifications are yet to be transformed.

Transformation of Priority Specifications

Transformation of priority specifications brings in the issue of manifestation of requests within the synchronizer. This requires some insight into the notion of implementation which is presently introduced.

We assume that the target language possesses an abstract data type called 'queue' with primitives to enqueue elements onto and dequeue elements from them. A queue 'element' is designated by a queue name (say Q) qualified by an 'element index' (say i), as in Q[i].

The attributes of an operation are :

op_name	Name of the operation.
op_class	Operation class.
nec_cond	Conditions necessary for the operation to be enabled.
intracp	Priority of the operation within its operation class.
intercp	Priority of the operation with respect to operations in other classes.

Op'attr denotes the attribute 'attr' of operation 'op'. From the informal definition of 'enabled(op)' given earlier, enabled(op) iff op'nec_cond.

Attributes of a queue are:

op_class	The class(es) of operations that can enqueue onto that queue.
pr_rule	The priority rule that applies to all operations in the queue, if one such rule exists.
pr_class	Intercp value of the operations in the queue if all have the same inter class priority.
len	Number of elements in the queue.

Q'attr refers to attribute 'attr' of the queue named 'Q'.

A queue element corresponds to a request for an operation. Thus Q[i] and Q[i]'nec_cond refer, respectively, to the operation and necessary condition corresponding to the i^{th} element in Q. Given a queue Q, $i \varepsilon Q$ stands for $i \varepsilon \{1..Q'len\}$ and $Op \varepsilon Q$, iff $\exists i \varepsilon Q(Q[i]=op)$.

General Statement of Priority

Before we translate priority specifications, it will be instructive to examine what is meant by priority in general, and how the specifications determine priority among operations. When we say that an operation (say b) has higher priority than another (say a), we mean that a can be serviced only after b is serviced, i.e.,

```
[]{req(a) & req(b) &  (a'pr < b'pr)
        => start(a) ONLYAFTER start(b)}
```

where op'pr is a pseudo attribute of 'op' computed using op'intercp and op'intracp (as shown below). This expression of priority can be shown to be equivalent to

```
[]{start(b) ONLYIF ~req(a)[a'pr > b'pr]}
```

The general semantics of priority is then,

```
[]{start(a) ONLYIF
   ~req(b)[b'pr > a'pr]}                          (P1)
[]{start(a) ONLYIF
   ~req(b)[enabled(b) & (b'pr > a'pr)]}           (P2)
```

where P1 is applicable when priority is specified among requested operations and P2 among enabled operations.

Now we will discuss how op'pr is determined for any operation op. The inter class priority specification "r : $opc_2 > opc_1$" has the following semantics :

$$\forall op_1 \varepsilon opc_1,\ \forall op_2 \varepsilon opc_2,$$
$$\{r\ \&\ req(op_1)\ \&\ req(op_2) \Rightarrow op_2'intercp > op_1'intercp\} \qquad \text{(DEFN1)}$$

The intra class priority specification "opc_1 :- r : pr_rule" has the following semantics:

$$\forall op_1, op_2\ \varepsilon opc_1,$$
$$\{[r\ \&\ req(op_1)\ \&\ req(op_2)\ \&\ (pr\_rule|_{op_1} > pr\_rule|_{op_2})] \Rightarrow [op_1'intracp > op_2'intracp]\} \qquad \text{(DEFN2)}$$

These follow directly from the definitions of intracp and intercp. Also,

```
If (a'intercp > b'intercp) then (a'pr > b'pr).
If a'op_class = b'op_class and
   (a'intracp > b'intracp) then (a'pr > b'pr).
                                          (DEFN3)
```

As was noted earlier, since an operation's intercp and intracp can vary with resource state for any operation 'op', op'pr is also dependent on resource state. Notice that '>' defines a partial ordering among operations.

Now we proceed with the transformation of priority specifications. Transformations will be consistent with the specifications if:

1. There exists a one-to-one mapping from requested operations to elements in queues. This is ensured by enqueuing each request onto its "waiting-queue".

2. From each queue, always only a certain "preferred" operation is serviced.

3. An operation is serviced only if it is enabled.

4. When an operation is serviced, appropriate (specified) actions are caused.

Systematic transformation rules exist for priority specifications. These are based on

- The type of priority specified, viz. among enabled or requested operations, or inter class or intra class priority,

- The dependency of pr_rule on resource state,

- The behavior of necessary conditions of operations, etc.

Space limitations preclude discussion of the details of these transformation rules. Instead, the translation required by the limited resource problem will be explained in detail. The following is true for this problem.

1. Priority applies among enabled operations only.

2. All acquire (release) operations have the same necessary conditions.

3. Order of arrival determines the priority within each class.

1) Since intra class priority is not specified, and all operations in a class have the same necessary condition, a queue is designated for each operation class and for each queue 'Q',

```
[]{start(Q[i]) ONLYIF i=1}.
```

2) Since priority is specified among enabled operations, <u>inter class priority</u> manifests itself as follows:

```
[]{start(Q[i]) ONLYIF
 [∀Q1(Q1'pr_class > Q'pr_class) disabled(Q)]}.
```

Here disabled(Q) stands for ∀op∈Q{disabled(op)} and enabled(q) for ~disabled(q).

For the limited resources problem, we designate 'aq' and 'rq' to serve as the queues for acquire and release respectively. The following four pairs of statements result.

```
req(a) => a∈aq;
req(r) => r∈rq;

[]{start(aq[i]) ONLYIF
       i=1 & enabled(aq[i]) & disabled(rq)};
[]{start(rq[i]) ONLYIF
       i=1 & enabled(rq[i])};

start(aq[i]) CAUSES (free<-free-1);
start(rq[i]) CAUSES (free<-free+1);

enabled(aq[i]) means [free<max & req(a) &
              ~exec$release & ~exec$acquire]

enabled(rq[i]) means [ 0<free & req(r) &
              ~exec$release & ~exec$acquire]
```

The above transformations ensure that

1. A request is enqueued onto a designated queue depending on its class and necessary conditions.

2. An element is serviced only if it is enabled and there are no requests with higher priority.

3. Servicing a request causes the required action.

Hence the transformation above preserves the semantics of the top-level specifications.

From an examination of the possible primitive conditions, we observe that since at this stage of the transformation, requests are manifest as elements in queues, conditions of the form req(a)[cond] will have to be expressed in terms of conditions on elements in the queues. This is achieved by knowing the relationship between queues, operations, and their necessary conditions.

<u>Deriving Target Language Code</u>

In an abstract sense, the synchronizer code consists of the following types of statements:

- Enqueuing statements: These indicate how the synchronizer responds to the arrival of requests. After determining the class of the request and the conditions that hold at the time of arrival, the synchronizer determines the queue onto which a request has to be enqueued. Equivalently, the synchronized processes may enqueue onto the appropriate queue.

- Servicing statements: These involve the conditions necessary for a request in a queue to be serviced. After determining whether these conditions hold, the synchronizer takes actions tantamount to servicing a request.

- Causal statements: These indicate the changes to resource state, etc, that have to be caused after an operation is serviced. After servicing a request, the synchronizer effects these changes.

These are the only categories of executable statements that may be found in a synchronizer other than initialization code.

The scheduling discipline specification expresses the behavior that enabled operations should possess in order to be serviced. Thus their effect will have to be displayed by the choice made by the synchronizer in servicing enabled operations. They are, in turn, manifest in the servicing statements. Space constraints prevent detailed analysis of appropriateness of scheduling disciplines here.

We proceed to see how scheduling discipline specification manifests in the synchronization code constructed. We assume that Scheduling Discipline is independent of resource state.

If Scheduling Discipline specified is SD0-SD2, then each operation class has a separate queue. For each queue Q,

```
[]{∃i enabled(Q[i]) & ∀j 1≤j<i disabled(j)
                          => start(Q[i])}.
```

If SD3 is specified, all requests are enqueued onto a single queue, and the first enabled operation on the queue is serviced before the rest, i.e.,

```
[]{∃i enabled(Q[i]) & ∀j 1≤j<i disabled(Q[j])
                          => start(Q[i])}.
```

This transformation is valid if an enabled operation can be disabled only by the synchronizer.

For SD4, a FCFS scheduling discipline is required and hence all requests are enqueued onto a single queue. The first element in the queue is always serviced once it is enabled, i.e.,

```
 [ ]{enabled(Q[1]) => start(Q[1])}.
```

If all operations in a class have the same nec_cond then all operations in a queue are enabled or all are disabled. This can be used to advantage as follows: If Scheduling Discipline is specified as SD0-SD2 then each operation class has a unique queue. For each queue Q,

```
 [ ]{enabled(Q[1]) => start(Q[1])}.
```

In situations where priority specifications apply, scheduling discipline SD2P will be in effect which will be satisfied if the servicing was done as in

```
 [ ]{enabled(op) & cond_on_op => start(op)},
```

where cond_on_op was the condition derived from priority specifications.

Realization of a synchronizer for the Limited resources Problem

The preconditions for servicing operations are first simplified thus:

```
 [non-empty(aq) & free>0 & ~exec$acquire &
            ~exec$release & disabled(rq)] =>
 [non-empty(aq) & free>0 & ~exec$acquire &
        ~exec$release & (empty(rq) V free=max)].
```

Mapping the result of the transformations derived so far into the primitives of Sentinels, a construct introduced in [11], we arrive at the following Sentinel implementation of the synchronizer.

```
 Procedure Limited_Resources (rq,aq : queues);
 max : constant integer := 10;
 a_count,r_count,free : integer;
 a_count:=0; r_count:=0; free:= max;
 Do while (True)
  If non_empty(rq) & free<max &  a_count=0 &
                                      r_count=0}
    then begin
         detach execute rq[1] count(r_count);
         free := free+1
         end;
  If {non_empty(aq) & 0<free & a_count=0 &
        r_count=0 & [free=max V empty(rq)]}
    then begin
         detach execute aq[1] count(a_count);
         free := free-1
         end
  end;
```

This sentinel program is arrived at after the following translation from primitive conditions in the specification language to primitives in sentinels.

```
    Start(q[i]) --> detach execute q[i]
    Exec(op) --> op_count>0
    Req(op) --> non_empty(op_q)
```

Each request has a distinct request queue in the sentinel. Concurrent processes will enqueue acquire requests onto aq, and release requests onto rq as per the transformation rule for priority.

Obviously, this sentinel will be grossly inefficient since it is 'busy waiting' for one of the necessary conditions to hold. Applying techniques similar to Schmid [18] and Ford [5], more efficient code can be generated. For instance, when a primitive condition becomes true, we need to evaluate only those necessary conditions which are "influenced" by it. Means to arrive at optimum code is one of the directions on which we are currently working. This example was presented only to give an example of the structure of the resulting solution. By providing rules to translate primitive conditions in the specification language to primitives in Serializers [1] and Monitors [9], we should be able to synthesize code for Serializers and Monitors.

RELATED WORK

There have been a variety of specification languages based on regular expressions [19]. Of them, Path Expressions [8] are perhaps the most widely referenced. Numerous versions of Path Expressions have since been published. Since specifications are in the form of permitted operation sequences, rather than exclusions, invariants, etc., contrived path expressions may result. Further, the notion of eventuality is not expressible in path expressions.

Synthesis of synchronizers from specifications in Grief's language [6] has been described by Laventhal [13]. In that approach, properties such as exclusion, priority, etc. are engineered by suitable ordering specification for some 'key' events pertaining to an operation. The language seems to lack the expressiveness to specify eventuality and synchronization properties dependent on resource state. One of the drawbacks of the synthesis algorithm is the necessity to consider all possible orderings of event expressions contained in the specifications in order to determine the set of allowable orderings.

Another work with a goal similar to ours is that of Griffiths [7]. The problem description supplied to her synthesizer consists of a low-level specification of the problem. Calls to synchronizing functions surround code that access shared resources. Code for the synchronizing functions is generated using the assertions that precede and immediately follow the calls. Our problem description is at a higher level in that it specifies the problem and not a solution to the problem.

A language and implementation for mutual exclusion only has been proposed by Brinch Hansen and Staunstrup [3].

CONCLUSION

This presentation reflects some of our current thinking with respect to specification languages for concurrent systems and certain aspects of synthesis of synchronization code for concurrent programs. The previous sections elaborated our present ideas with respect to the two broad goals: Mechanisms for Specification, and Synthesis of Synchronizers.

The specification language has constructs for stating the set of properties that are normally relevant to concurrent operations, namely ordering, fairness, priority, exclusion (and by default, concurrency), and invariance of resource state. Temporal logic provides the framework to express their semantics precisely. Some of the positive features of this language, are evidenced by the example used in the paper.

An important problem in specifying program behavior is whether or not one can verify that the specification itself is correct. This problem is aggravated by the conceptual gap that normally exists between the informal notion of what the problem is expected to solve and the formal specification technique. Hopefully, the specification language we have proposed here will help bridging this gap. We believe that the approach taken here meets Bloom's criteria [2] for a synchronization mechanism to be suitable for the construction of well-structured software. Development of a specification language that aids programmers and at the same time is amenable to automatic synthesis of programs has been our prime concern here, not a formally complete language.

With respect to the goal of automatically constructing code for synchronization, we presented an algorithm which derives synchronization code through successive transformation of specification statements. At each stage, we gave qualitative reasoning for correctness of the translation process. We consider noteworthy its ability to synthesize synchronizers with prespecified fairness. The issue of formal validation of the synthesis algorithm has been explored, but is beyond the scope of this paper.

An important issue is the practicality of the synthesis algorithm. One virtue of the present technique is the direct correspondence between specifications and their implementation. However, this can also lead to construction of inefficient programs. We have not approached this problem in any systematic way, since derivation of correct programs has been our main concern so far. Also, not all steps in the synthesis fall under the category of pattern directed translation. Rule 3 for instance, and simplification of necessary conditions for servicing operations, require a logical simplifier (albeit with limited capabilities) built into the system. The development herein provides the setting for a more general automatic synthesis procedure, which demands additional work before it becomes a viable tool.

[1] Atkinson, R.R. and Hewitt, C.E. Specification and Proof Techniques for serializers. <u>IEEE Transactions on Software Engineering SE-5</u> (Jan 1979), 10-23.

[2] Bloom, T. Evaluating Synchronization Mechanisms. Proceedings of the Seventh Annual Symposium on Operating Systems Principles, (Dec, 1979), pp. 24-32.

[3] Brinch Hansen, P. and Staunstrup, J. Specification and Implementation of Mutual Exclusion. <u>IEEE Transactions on Software Enginering SE-4</u> (Sep 1978), 365-370.

[4] Courtois, P.J., Heymans, F. and Parnas, D.L. Concurrent Control with 'Readers' and 'Writers'. <u>Communications of the ACM 14</u> (Oct 1971), 667-668.

[5] Ford, W.S. Implementation of a Generalized Critical Region Construct. <u>IEEE Transactions on Software Engineering SE-4</u> (Nov 1978), 449-455.

[6] Greif, I. A Language for Formal Problem Specification. <u>Communications of the ACM 20</u> (Dec 1977), 931-935.

[7] Griffiths, P. SYNVER: A System for the Automatic Synthesis and Verification of Synchronization Processes. TR 22-74, Harvard University, (1974).

[8] Habermann, A.N. Path Expressions. Carnegie-Mellon University, (June, 1975).

[9] Hoare, C.A.R. Monitors: An Operating System Structuring Concept. <u>Communications of the ACM 17</u> (Oct 1974), 540-557.

[10] Holt, A. and Commoner, F. Events and Conditions. Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, (June, 1970), pp. 3-52.

[11] Keller, R.M. Sentinels: A Concept for Multiprocess Coordination. UUCS-78-104, University of Utah, (June, 1978).

[12] Lamport, L. 'Sometime' is Sometimes 'Not Never'. Proceedings of the Seventh Annual Symposium on POPL, (Jan, 1980), pp. 174-185.

[13] Laventhal, M.S. <u>Synthesis of synchronization code for data abstractions</u>. Ph.D. Th., Massachusetts Institute of Technology,(June 1978).

[14] Manna, Z. <u>Mathematical Theory of Computation</u>. McGraw-Hill, (1974).

[15] Pnueli, A. The Temporal Semantics of Concurrent Programs. Lecture Notes in Computer Science, (June, 1979), pp. 1-20.

[16] Pnueli, A. On the Temporal Ananysis of Fairness. Proceedings of the Seventh Annual Symposium on POPL, (Jan, 1980), pp. 163-173.

[17] Rescher, N. and Urquhart, A. <u>Temporal Logic</u>. Springer-Verlag, (1971).

[18] Schmid, H.A. On the efficient Implementation of Conditional Critical Regions and the Construction of Monitors. <u>Acta Informatica 6</u> (1976), 227-249.

[19] Shaw, A.C. Software Specification Languages Based on Regular Expressions. Software Tools Workshop Conference, (May, 1979), pp. 1-39.

SESSION 12: INTERCONNECTIONS II

DATA BROADCASTING IN SIMD COMPUTERS*

David Nassimi** and Sartaj Sahni

University of Minnesota

Summary

An SIMD (Single Instruction stream, Multiple Data stream) computer consists of some number, N, of processing elements (PEs). Each PE(i), $0 \le i \le N-1$, has a local memory. The PEs communicate through an interconnection network. Three models of SIMD computers are considered; these models differ only in the way the PEs are interconnected [8]: 1) Mesh Connected Computer (MCC) with $N = n^q$ PEs forming a q-dimensional $n \times n \times \ldots \times n$ mesh. Each PE is connected to (at most) 2q nearest neighbors. 2) Cube Connected Computer (CCC) where each PE is connected to $\log N$ other PEs. 3) Perfect Shuffle Computer (PSC) with each PE connected to (at most) three other PEs.

Let D(i) be a data item contained in PE(i), $0 \le i \le N-1$. The data broadcasting problem for SIMD computers can be posed in two different ways:

(i) Random Access Read (RAR)

In this formulation, an index S(i) is contained in PE(i), $0 \le i \le N-1$. PE(i) is to receive data from PE(S(i)). If PE(i) is not to receive data from any other PE, then $S(i) = \infty$.

(ii) Random Access Write (RAW)

Here, an index W(i) is contained in PE(i). Data from PE(i) is to be transmitted to PE(W(i)), $0 \le i < N$. If $W(i) = \infty$ then data from PE(i) is not transmitted to any PE.

Some applications of RARs and RAWs may be found in [4] and [5].

The RAR form of the data broadcasting problem has been studied by Thompson [6]. He shows that any RAR can be performed by making use of the switch settings of a generalized-connection-network (GCN) realizing the input-output mapping that corresponds to the RAR. On an nxn MCC, his algorithm requires no more than 13n-16 unit-routes (a unit-route is a data transfer between PEs that are adjacent in the interconnection network) for any RAR. On an N-PE CCC and PSC, his algorithm requires respectively $4\log N - 3$ and $8\log N - 7$ unit-routes. None of these complexity figures includes the time needed to determine the GCN switch settings. If this time is included, the complexity of Thompson's algorithm is determined by the complexity of the GCN set-up algorithm. The best known parallel GCN set-up algorithms (Nassimi and Sahni [5]) have complexity of $O(n)$ on an nxn MCC, and $O(\log^4 N)$ on both CCCs and PSCs with N PEs.

In this paper, we present an algorithm for the RAR problem which runs in $O(q^2 n)$ time on a q-dimensional $n \times n \times \ldots \times n$ MCC, and in $O(\log^2 N)$ time on an N-PE PSC or CCC. Thus, the algorithm of this paper is asymptotically faster than the Thompson-Nassimi-Sahni algorithm for CCCs and PSCs. For MCCs, we expect our algorithm to be significantly faster than the Thompson-Nassimi-Sahni algorithm as the algorithm of this paper is significantly simpler and has much less overhead.

The RAW problem can be solved using the sub-algorithms developed for the RAR problem. Let d be the maximum number of data items to be written into any one PE. The time complexity of the RAW is $O(q^2 n + dqn)$ on a q-dimensional MCC, and $O(\log^2 N + d\log N)$ on an N-PE CCC or PSC.

RARs and RAWs are performed using certain well defined steps. These are described below:

(i) SORT: In a sort, records are rearranged so as to be in non-decreasing order of a specified key. Let G(i) denote the record in PE(i), $0 \le i < N$. Let H(i) be the key field of record G(i). H(i) is also in PE(i). Following a sort, the records will have been rearranged such that $H(i) \le H(i+1)$, $0 \le i < N-1$.

(ii) RANK: The rank of a selected record is the number of selected records in PEs with a smaller index. For example, assume we have 8 PEs each containing one record. Let the key values for these 8 records be (6, 4, 2, 2*, 6, 6*, 3*, 4*) where an asterisk over a key value denotes a flag or selected record. The ranks of the flagged records are (-,-,-, 0, -, 1, 2, 3).

(iii) CONCENTRATE: Let $G(i_r)$, $0 \le r \le j$, $j \le N-1$, be a set of records with $G(i_r)$ initially in $PE(i_r)$. Assume that the records have been ranked so that $H(i_r)=r$. A concentrate results in record $G(i_r)$ being moved to PE(r), $0 \le r \le j$.

(iv) DISTRIBUTE: Let G(i), $0 \le i \le j < N$, be a set of records with G(i) initially in PE(i). Let H(i), $0 \le i \le j$, be a set of destinations such that $H(i) < H(i+1)$, $0 \le i < j$. The purpose of a distribute is to route G(i) to PE(H(i)), $0 \le i \le j$. It is easy to see that a distribute is the inverse of a concentrate.

(v) GENERALIZE: A generalize makes multiple copies of records. The initial configuration is record G(i) in PE(i), $0 \le i \le j < N$. Each record has a field H (high). The H values are such that $0 \le H(0) < H(1) < \ldots < H(j) \le N-1$, and $H(i) = \infty$ for $j < i < N$. Generalize copies record G(i) into PEs $H(i-1)+1$ through H(i), $0 \le i \le j$ (we assume, for convenience, $H(-1) = 0$).

Our RAR algorithm is best described by considering an example (Figure 1). We have N=8 PEs and S(0:7)=(2, 6, 2, ∞, 5, 6, ∞, 6). (Recall that S(i) specifies the PE from which the data for PE(i) is to be fetched, and $S(i) = \infty$ iff PE(i) is to receive no data.) Let $T(i) = i$ and $FLAG(i) = 1$, $0 \le i < N$. Our RAR algorithm begins by sorting the records $G(i) = \langle S(i), T(i), FLAG(i) \rangle$. Records are sorted on S; T is used to resolve ties

* This research was supported in part by NSF grant MCS 78-15455.

** Dr. Nassimi is now with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, Illinois 60201.

CH1569-3/80/0000-0325$00.75 © 1980 IEEE

(i.e. records with the same S value are ordered by their T value). The sorting algorithm we shall use is a comparison sort. We require that during the sort whenever a comparison between G(i) and G(j) is made, if $S(i) = S(j)$ and $T(i) < T(j)$ then FLAG(i) is set to zero. As a result of this, following the sort, $FLAG(i) = 1$ only for records with distinct S values. For records with the same S value, $FLAG = 1$ only for the record with highest T value. Lines 3-4 of Figure 1 give the result of the sort. The S values with an asterisk above them correspond to records with a FLAG of 1.

The next step is to rank the records with a flag of 1. This results in the rank assignment of line 5 (Figure 1). For PEs containing a record G with $FLAG = 1$, we may define a new record G' where $G'(i) = \langle R(i),U(i),S(i)\rangle$, R(i) is the rank just determined, $U(i) = i$ and S(i) is as in line 4 of Figure 1. The G'(i)s are concentrated to obtain the configuration of lines 6 and 7. At this point, we define a new record, G", for each PE containing a G' type record. $G''(i) = \langle S(i),V(i)\rangle$ where $V(i)=i$. The newly defined G" type records are distributed according to S to get line 8. Observe that now a PE contains a G" type record iff its data is to be transmitted to another PE. Let D(i) be the data in PE(i) that is to be broadcast. The T, U and V registers of each PE contain return addresses that will now be used to broadcast the data.

First, the data to be broadcast is concentrated using the ranks contained in the V registers (line 9). Next, the data is generalized using the values in the U registers as the corresponding H values in the definition of generalize. This yields the configuration of line 10. Finally, the broadcast data is sorted using the T value in each PE as the sort key. The result (line 11) is that data has been broadcast to all PEs requesting data. It should be easy to see that the algorithm just described provides a correct solution to the RAR problem.

The RAW problem is simpler to handle than the RAR problem. When all the W(i)s that are not equal to ∞ are distinct, the RAW problem may be solved by first sorting the broadcast data into non-decreasing order of W(i). This sort is followed by a distribute step in which the data being broadcast is distributed according to the W values. When the W(i)s are not distinct, the distribute step will not be free of conflict. The conflict is to be resolved in a manner that depends on what is desired by the RAW. We consider two situations:

(i) If $W(i_1) = W(i_2) = \ldots = W(i_r) = i \neq \infty$ then PE(i) is to receive only the data from PE(j) where $j = \min_{1\le k\le r} \{i_k\}$.

(ii) If $W(i_1) = W(i_2) = \ldots = W(i_r) = i \neq \infty$ then PE(i) is to receive data from all r PEs.

The first situation is handled by beginning with records $G(i) = \langle W(i),T(i),D(i),FLAG(i)\rangle$, $0 \le i < N$, where $T(i) = i$ and $FLAG(i) = 1$. Records are sorted by W(i) (using T(i) to break ties). The sort is similar to the first SORT step of an RAR except that FLAG(i) is set to zero if $W(i)=W(j)$ and $T(i) > T(j)$. The sort is followed by a ranking of the records. Define a new record $G'(i) = \langle W(i),D(i)\rangle$ for each PE containing a record with a FLAG of 1. The records G' are next concentrated using the ranks just computed. Finally, the D(i)s of the concentrated records are distributed using the W fields. Thus, a RAW essentially corresponds to lines 1 through 8 of Figure 1.

Situation (ii) of an RAW can be handled in a manner similar to situation (i). The details may be found in [8].

References

1. Batcher, K., "Sorting networks and their application," Proc. AFIPS 1968 SJCC., Vol. 32, AFIPS press, Montvale, N.J., pp. 307-314.

2. Nassimi, D. and Sahni, S., "Bitonic sort on a mesh-connected parallel computer," IEEE Trans. on Computers, C-28, No. 1, Jan. 1979, pp. 2-7.

3. Nassimi, D. and Sahni, S., "Parallel permutation and sorting algorithms and a new generalized-connection-network," Univ. of Minnesota, TR #79-8, 1979. To appear in Journal of the ACM.

4. Nassimi, D. and Sahni, S., "Finding connected components and connected ones on a mesh-connected parallel computer," Univ. of Minnesota, TR #79-18, 1979. To appear in SIAM Journal of Computing.

5. Nassimi, D. and Sahni, S., "Parallel algorithms to set-up the Benes permutation network," Univ. of Minnesota, TR #79-19, 1979.

6. Thompson, C., "Generalized connection networks for parallel processor intercommunication," IEEE Trans. on Computers, C-27, No. 12, Dec. 1978, pp. 1119-1125.

7. Thompson, C. and Kung, H., "Sorting on a mesh-connected parallel computer," CACM, Vol. 20, No. 4, 1977, pp. 263-271.

8. Nassimi, D. and Sahni, S., "Data broadcasting in SIMD computers," Univ. of Minnesota, TR #79-17, 1979.

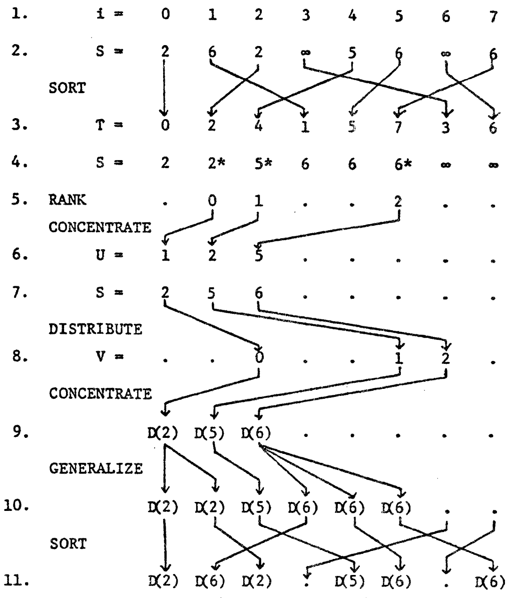


Figure 1 RAR example

PACKET COMMUNICATION IN MULTISTAGE SHUFFLE-EXCHANGE NETWORKS *

Daniel M. Dias and J. Robert Jump
Department of Electrical Engineering
Rice University
Houston, TX 77001

Summary

This paper summarizes research we have done on asynchronous packet communication in buffered, 2^n input, Shuffle Exchange Networks with k stages, $1 \leq k \leq n$, (denoted by SEN(2,n,k)) shown in fig. 1. In this asynchronous packet communication environment the networks can deadlock. The research reported here considers deadlock detection, recovery and the performance of these networks.

The single stage SEN(2,n,1) [2,3,7] and the OMEGA network [4] (which, without "broadcast" at switches, is essentially an SEN(2,n,n) without the feedback links from stage n to stage 1) have been studied for their permutation capability. The performance of delta networks (which include the SEN(2,n,n) with the feedback links from stage n to stage 1 deleted) has been studied in [1,6], for a packet communication environment. Simulation results and bounds on network performance indicate that a range of performance can be obtained by varying the number of stages and size of buffers between stages of the network.

The environment we consider is one in which input packets, containing both the data to be transferred and the address of the network output link to which the data is to be passed, arrive asynchronously at SEN input links. The SEN uses different bits of this destination address to direct a packet as it advances through the stages of the network [1-7]. The operation of (2 x 2) switches in the SEN is modelled essentially as follows [1]: A fixed maximum queue length of waiting packets is allowed between stages. Each switch handles an input packet at each input link simultaneously. It takes time "t_select" to determine the successor node to which the packet is to be sent. If that output is in use (i.e. another input packet is in the process of being passed to that output) it waits its turn for the use of that output link (with equiprobable selection of packets that request simultaneous passage through the same switch output link). When the selected output link becomes available, it delays the data for another time interval "t_pass" which represents gate delay. At this time the data is available at output lines of the link. A wait state is now entered (if necessary) until a buffer in the selected output queue becomes available.

A packet incident on a switch is said to be blocked if it encounters a full buffer at the switch output link through which it must pass. A deadlock is said to occur if a set of packets in the SEN is permenantly blocked. A necessary and sufficient condition for a deadlock is the occurrence of a cycle of blocked packets. It can be shown that the SEN can recover from a deadlock by advancing each packet in a blocked cycle by one stage.

Schemes for the detection of a deadlock have been proposed. In these schemes, when a packet is blocked, test packets are passed along the blocked path to determine if a deadlock has occurred. For an SEN(2,n,k) the longest possible cycle length is $L = (2^n \times k)$. Suppose that it takes time t_test for a test packet to pass through a switch and suppose that a deadlock is caused by a cycle of blocked packets of length m. A deadlock detection scheme has been proposed that does not require knowledge of t_test and which takes time $(t\_test.(m + mL - m^2 - 1))$ to detect a deadlock. Another proposed scheme depends on the knowledge of time t_test and takes time (2.t_test.L) to detect a deadlock.

Each cycle of blocked packets must pass through a switch in stage 1 of an SEN(2,n,k) (fig. 1). Thus, for deadlock recovery, it is sufficient to have an additional buffer at each stage 1 switch, specifically for this purpose. It then takes time (t_recovery.k) to recover from a detected deadlock in an SEN(2,n,k), where t_recovery is of the same order of magnitude at t_pass. Alternatively, deadlock recovery can be speeded up by having a "deadlock recovery buffer" at each switch input and simultaneously advancing all packets in a detected blocked cycle. The deadlock recovery time, t_recovery, for this case is a constant of the same order of magnitude as t_pass.

Event driven simulations of SEN(2,n,k), $1 \leq k \leq n$, have been performed. These simulations vary the number of input links (2^n), stages (k), buffer lengths between stages, t_select, t_pass, t_test and t_recovery. Simulation results indicate the following:

(i) When single stage networks, with one buffer between switches, are operated at very high input rates, deadlocks occur very often (approximately one deadlock for every 3 packets that enter the network).
(ii) The frequency of deadlock occurrence can be dramatically reduced by
  (a) increasing the number of stages in the network. (The SEN(2,n,n) is deadlock free),

* This work was supported by the National Science Foundation under grant DCR74-14283.

CH1569-3/80/0000-0327$00.75 © 1980 IEEE

(b) increasing the buffer size between stages,
(c) controlling the input rate to the network.

(iii) A range of "maximum performance" can be obtained by varying the number of stages in the network. The upper limit of performance of these networks is comparable to the same size crossbar switch [1].

Typical simulation results for SEN(2,n,5), $1 \leq k \leq 5$ are shown in fig. 2. Some of the research in progress is as follows:

(i) The performance of the outlined schemes is being compared with the synchronous technique in [5].
(ii) The performance of the networks with other input packet flow control strategies is being studied. (An example is to restrict the number of packets in the network from each source).
(iii) The multiplicity of paths from each network input link to each network output link (as opposed to the unique paths in delta networks [1]) can be used to improve SEN reliability. This aspect is being investigated.

References

[1] Dias, D. M., Jump, J. R. , "Analysis and Simulation of Buffered Delta Networks", Proceedings of the Workshop on Interconnection Networks for Parallel and Distributed Processing , (April 80), pp. 84-92.

[2] Lang, T. , "Interconnections between Processors and Memory Modules Using the Shuffle-Exchange Network", IEEE Transactions on Computers, Vol. C-25, No. 5., (May 76), pp. 496-503.

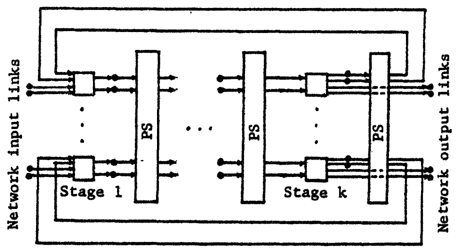
[3] Lang, T. , Stone, H. S. , "A Shuffle-Exchange Network with Simplified Control", IEEE Transactions on Computers , Vol. C-25, No. 1, (Jan. 76), pp. 55-65.

[4] Lawrie, D. H. , "Access and Alignment of Data in an Array Processor", IEEE Transactions on Computers , Vol C-24, No. 18, (Dec. 75), pp. 1145-1155.

[5] Lawrie, D. H., Padua, D. A. "Analysis of Message Switching with Shuffle-Exchanges in Multiprocessors", Proceedings of the Workshop on Interconnection Networks for Parallel and Distributed Processing, (April 80), pp. 116-123.

[6] Patel, J. H. , "Processor - Memory Interconnections for Multiprocessors", Proceedings of the 6th Annual Symposium on Computer Architecture , IEEE, (April 79), pp. 168-177.

[7] Stone, H. S. , "Parallel Processing with the Perfect Shuffle", IEEE Transactions on Computers , Vol. C-20, No. 2, (Feb. 71), pp. 153-161.



Notation: ● denotes a (finite length) first-in-first-out buffer.
□ denotes a switch that can pass a packet from any input link to any output link.
PS : denotes the Perfect Shuffle permutation

Fig. 1 A 2^n input, k-stage Shuffle Exchange Network (SEN(2,n,k)) for $1 \leq k \leq n$.

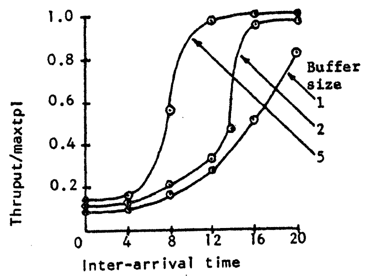


Fig. 2(a) Thruput/maxtpl vs. inter-arrival time for an SEN(2,5,1) with a variable buffer size.

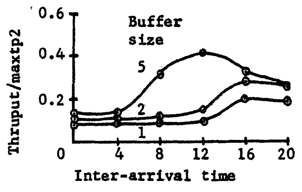


Fig. 2(b) Thruput/maxtp2 vs. inter-arrival time for an SEN(2,5,1) with variable buffer size.

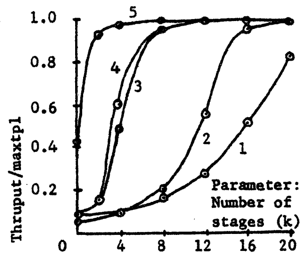


Fig. 2(c) Thruput/maxtpl vs. inter-arrival time for an SEN(2,5,k), $1 \leq k \leq 5$.

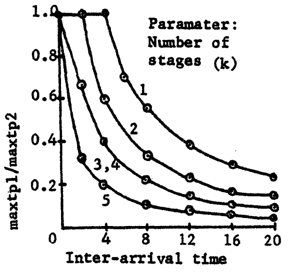


Fig. 2(d) maxtpl/maxtp2 vs. inter-arrival time for an SEN(2,5,k), $1 \leq k \leq 5$.

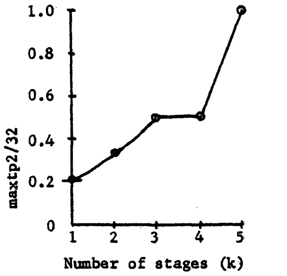


Fig. 2(e) maxtp2 vs. k for an SEN(2,5,k), $1 \leq k \leq 5$.

Notation:
Inter-arrival time: Average interarrival time of packets (exponentially distributed) after a buffer at a network input link becomes available.
Thruput: Average number of packets put out by the network in unit time.
maxtpl: Maximum thruput of a network for a given inter-arrival time.
maxtp2: Maximum thruput of a network at any inter-arrival time.

Parameters:
t_select= 0. t_pass= 1.0. t_test= 0.1. t_recovery= 1.0.
m= length of deadlock cycle. L= 32k for an SEN(2,5,k).
Deadlock detection time= $0.1(m + mL - m^2 - 1)$.

Fig. 2 Typical simulation results for an SEN(2,5,k), $1 \leq k \leq 5$.

A Layout for the Shuffle-Exchange Network

Dan Hoey
Charles E. Leiserson

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Abstract—This paper describes a technique for producing a VLSI layout of the shuffle-exchange graph. It is based on the layout procedure in [2] which lays out a graph by bisecting the graph, recursively laying out the two halves, and then combining the two sublayouts. The area of the layout is related to the number of edges that must be cut to bisect the graph.

For the shuffle-exchange graph on n vertices, we present a bisection schema for which the above procedure yields an $O(n^2/\lg n)$ area layout when $n = 2^k$ and k is a power of two. The bisection involves a mapping from vertices of the graph to polynomials, and the polynomials are subsequently evaluated at complex roots of unity. Incidental to this construction is a result on the combinatorial problem of necklace enumeration.

1. Introduction

The shuffle-exchange network has been shown to be an important communications structure for parallel processors. Stone [8] describes algorithms which use this structure to solve several problems, including the computation of the discrete Fourier transform and sorting bitonic sequences. The number of communications steps required by these algorithms is typically a polynomial in the logarithm of the number of nodes in the network, and the nodes themselves need only perform relatively simple operations.

VLSI designers often try to minimize the area used by a circuit subject to the requirements imposed by the fabrication technology on the minimum feature sizes of the components [5]. In [9] Thompson develops lower bounds on the growth of circuit area based on graph-theoretic properties of the communications structure. He shows in particular that any layout of the shuffle-exchange network on $n = 2^k$ vertices must use at least $\Omega(n^2/k^2)$ area. The arguments for Thompson's lower bounds are based on the *minimum bisection width* of a graph, which is the least number of edges that must be removed to separate the graph into two equal-sized subgraphs.

This research was supported in part by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-78-C-1551, the National Science Foundation under Grant MCS 78-236-76, and the Office of Naval Research under Contracts N00014-76-C-0370 and N00014-80-C-0236. Charles E. Leiserson receives support from a Fannie and John Hertz Foundation fellowship.

The concept of bisection width was extended by Lipton and Tarjan [3] to that of a *separator theorem* for a class of graphs closed under the subgraph relation. In essence, a separator theorem for a class provides upper bounds on the bisection widths of graphs in the class. Separator theorems allow the divide-and-conquer paradigm to be exploited in the design efficient algorithms for graph manipulation [4]. Recently, Leiserson [2] has used this approach to design area-efficient VLSI layouts.

In this paper a theorem similar to a separator theorem is proven for the shuffle-exchange graph on $n = 2^k$ vertices. We exhibit a *dissection* that shows how the shuffle-exchange graph may be bisected, how the resultant subgraphs may themselves be bisected, and so forth. We use this result to construct an $O(n^2/k)$ area layout for the case when k is a power of two, thereby improving Thompson's upper bound of $O(n^2/\sqrt{k})$. In our proof the vertices of the shuffle-exchange graph are mapped to a polynomial space, and then the polynomials are mapped to the complex plane. This construction also provides an asymptotic result on the combinatorial problem of necklace enumeration.

The next section formalizes the notions of bisection and dissection. Section 3 introduces the shuffle-exchange graph and describes its relationship to polynomials. In Section 4 we construct a bisection of the shuffle-exchange graph whose width is $O(n/k)$, and in Section 5 we extend this result to produce a dissection. In Section 6, the layout algorithm of [2] is applied to this dissection to produce an $O(n^2/k)$ area layout for the shuffle-exchange graph. Section 7 concludes by comparing this result with other work in the field.

2. Graph Dissection

In this section, we formalize concepts pertaining to the partitioning of a graph into smaller graphs by the removal of edges.

A *bisection* S of a graph $G = (V, E)$ into graphs $G' = (V', E')$ and $G'' = (V'', E'')$ is a disjoint partition of the vertices $V = V' \cup V''$ together with a disjoint partition of the edges $E = E' \cup E'' \cup E_S$ such that the cardinalities of V' and V'' differ by at most one. The cardinality of E_S is called the *width* of the bisection, and the edges in E_S are said to be *removed* by the bisection. The graphs G' and G'' are called the *halves* of the bisection.

Of course, any graph can be bisected by removing all its edges, but usually we are interested in removing as few edges as possible. The *minimum bisection width* of a graph is the smallest number of edges that must be removed to divide an n-vertex graph into a $\lceil n/2 \rceil$-vertex graph and a $\lfloor n/2 \rfloor$-vertex graph. Unfortunately, the problem of finding the minimum bisection width of an arbitrary graph is NP-complete [1].

It is sometimes the case that every graph in a class of graphs can be bisected by the same general mechanism. We define a *separator* for a class $\mathcal{G}$ of graphs to be a family $\mathcal{S}$ of bisections such that $\mathcal{S}$ contains a bisection of every nontrivial graph G in $\mathcal{G}$. Interesting separators are those that exhibit the *closure property.* A separator $\mathcal{S}$ for a class of graphs $\mathcal{G}$ has this property if for any graph $G \in \mathcal{G}$, the halves G' and G'' that are produced by a bisection of G in $\mathcal{S}$ are also in $\mathcal{G}$. Any separator with the closure property whose associated class contains a particular graph G is called a *dissection* of G.

A dissection $\mathcal{S}$ of G may be thought of as a complete binary tree that has G at the root, the halves of G from some bisection in $\mathcal{S}$ as its sons, and the halves of the halves as grandsons, and so forth to trivial graphs at the leaves. If G has n vertices, then the subgraphs at level j will have about $n/2^j$ vertices. Although there may be other graphs in the class $\mathcal{G}$ associated with $\mathcal{S}$, at the very least $\mathcal{G}$ must contain all of the graphs in the tree.

In [3] Lipton and Tarjan introduce *separator theorems* which use ideas similar to those presented here. In their work, however, the discussion is restricted to classes of graphs that are closed under the subgraph relation. (A class $\mathcal{G}$ is closed under the subgraph relation if every subgraph G' of a graph $G \in \mathcal{G}$ is also an element of $\mathcal{G}$.) We have departed from their approach because the results of this paper rely on properties of the shuffle-exchange graph that do not hold for all of its subgraphs.

3. The Shuffle-Exchange Graph

The *shuffle-exchange graph* on n vertices is defined only when n is a power of two. Each vertex of the $n = 2^k$ vertices can be identified with an element of the Cartesian product

$$\{0, 1\}^k = \{ b_{k-1} b_{k-2} \ldots b_0 \mid b_j \in \{0, 1\} \}.$$

Each vertex $v \in \{0, 1\}^k$ is incident on an *exchange edge* $(v, \varepsilon(v))$ and two *shuffle edges* $(v, \sigma(v))$ and $(v, \sigma^{-1}(v))$, where ε and σ are permutations defined by

$$\varepsilon(b_{k-1} b_{k-2} \ldots b_1 b_0) = b_{k-1} b_{k-2} \ldots b_1 (1-b_0), \tag{1}$$

$$\sigma(b_{k-1} b_{k-2} \ldots b_1 b_0) = b_{k-2} b_{k-3} \ldots b_1 b_0 b_{k-1}. \tag{2}$$

In the literature the vertices are usually identified with integers from zero to $n-1$ represented in binary notation. The shuffle permutation σ is then the permutation applied to a deck of n cards by a perfect riffle shuffle, in which case $\sigma(m) \equiv 2m \pmod{n-1}$. The exchange permutation ε is the permutation that exchanges pairs of adjacent elements of the vertex set, so that $\varepsilon(m) = m \pm 1$.

The shuffle-exchange graph is highly structured because of the shuffle permutation. From equation (2) we see that $\sigma(v)$ can be determined from v by rotating the indices of v to the left one position. The shuffle permutation partitions $\{0, 1\}^k$ into equivalence classes known as *necklaces* [7], where two vertices are equivalent whenever the indices of one are a cyclic permutation of the indices of the other. Since rotation by k positions yields the original vertex, the cardinality of a necklace cannot exceed k.

The properties that we shall use to dissect the shuffle-exchange graph are expressed conveniently in terms of the *characteristic polynomial,* which is defined for a vertex $v = b_{k-1} \ldots b_0 \in \{0, 1\}^k$ as

$$p_v(x) = \sum_{0 \le j \le k-1} b_j x^j. \tag{3}$$

It should be apparent that $p_v(2)$ is precisely the vector v considered as a binary number, as discussed above. The following lemma shows the relationship between the characteristic polynomial and the shuffle and exchange permutations.

Lemma 1: For all $v \in \{0, 1\}^k$,

$$p_{\varepsilon(v)}(x) = p_v(x) \pm 1, \tag{4}$$

$$p_{\sigma(v)}(x) \equiv x\, p_v(x) \pmod{x^k - 1}, \tag{5}$$

where the congruence (5) is taken over the ring $\mathbb{Z}[x]$ of polynomials with integer coefficients.

Proof. From the defining equations (1) and (2),

$$p_v(x) - p_{\varepsilon(v)}(x) = b_0 x^0 - (1-b_0) x^0 = 2 b_0 - 1,$$

$$x\, p_v(x) - p_{\sigma(v)}(x) = b_{k-1} x^k - b_{k-1} x^0 = b_{k-1} (x^k - 1).$$

The lemma follows from the fact that each b_j is either zero or one. □

The cyclic structure of necklaces is exploited in Section 4 to bisect the shuffle-exchange graph. This is done in such a way that most of the necklaces in the graph are bisected. When the number of vertices in a necklace is even, it turns out that the half-necklaces also have a cyclic structure. An *m-cycle* is defined to be an ordered sequence $(v_0, v_1, \ldots, v_{m-1})$ of m distinct vertices such that for $j = 1, \ldots, m-1$,

$$p_{v_j}(x) \equiv x\, p_{v_{j-1}}(x) \pmod{x^m - 1}. \tag{6}$$

The next lemma provides justification for calling such a sequence an m-cycle.

Lemma 2: Let $(v_0, \ldots, v_{m-1})$ be an m-cycle. Any sequence $(v_j, \ldots, v_{m-1}, v_0, \ldots, v_{j-1})$ formed by cyclically permuting $(v_0, \ldots, v_{m-1})$ is also an m-cycle. If d is a divisor of m, then the subsequence $(v_0, \ldots, v_{d-1})$ is a d-cycle.

Proof. This lemma can be proved by manipulating the congruence (6) in the definition of an m-cycle. The congruence can be iterated to yield

$$p_{v_{m-1}}(x) \equiv x^{m-1} p_{v_0}(x) \qquad (\text{mod } x^m - 1),$$

and since $x^m \equiv 1 \pmod{x^m - 1}$, it follows that

$$x\, p_{v_{m-1}}(x) \equiv p_{v_0}(x) \qquad (\text{mod } x^m - 1).$$

Thus (6) holds between the first and last vertices as well as between adjacent vertices, implying that the choice of a first vertex is immaterial. To prove the second part of the lemma, observe that congruence (6) modulo x^m-1 must also hold modulo its divisor x^d-1. □

Congruence (5) shows that a necklace of k vertices is a k-cycle. Lemma 2 establishes that when k is even, the necklace can be bisected to yield two $k/2$-cycles.

4. Bisecting the Shuffle-Exchange Graph

The concepts developed in Section 3 are applied in this section to construct a bisection of the shuffle-exchange graph on $n = 2^k$ vertices. The construction is obtained by evaluating the characteristic polynomials of the vertices at a complex kth root of unity, inducing a mapping from $\{0, 1\}^k$ to the complex plane. The complex plane is then divided to induce a bisection of the shuffle-exchange graph. A corollary of this construction is an asymptotic result on the number of necklaces.

Let $\omega = e^{2\pi i/k}$ be the principal primitive complex kth root of unity, and consider the mapping $v \mapsto p_v(\omega)$ from $\{0, 1\}^k$ to the complex plane. Figure 1 graphs the values of $p_v(\omega)$ for $k = 5$. The vertices are labeled with $p_v(2)$. The solid lines forming pentagons concentric about the origin represent shuffle edges, and the horizontal dotted arcs represent exchange edges.

Let us examine this figure in relation to Lemma 1. The occurrence of regular k-gons of shuffle edges can be explained by congruence (5). Since ω is a root of x^k-1, this congruence becomes the equality $p_{\sigma(v)}(\omega) = \omega\, p_v(\omega)$. Thus $p_{\sigma(v)}(\omega)$ is the point obtained from $p_v(\omega)$ by a counterclockwise rotation of $2\pi/k$ radians about the origin. The vertices in a necklace are mapped to k points equally spaced on a circle about the origin, unless the entire necklace is mapped to the origin. The fact that exchange edges are horizontal can be explained by equation (4) in Lemma 1. If vertices v and $\epsilon(v)$ are incident on an exchange edge, then they are mapped to complex numbers that have the same imaginary part and differ by one in the real part.

The bisection of the shuffle-exchange graph will be achieved by partitioning the vertices based on the imaginary part of $p_v(\omega)$, with tie-breaking when $p_v(\omega)$ is real. All edges that cross the real line will be removed, and it will be shown that there are at most $O(n/k)$ of these. This bound is easily shown for edges whose incident

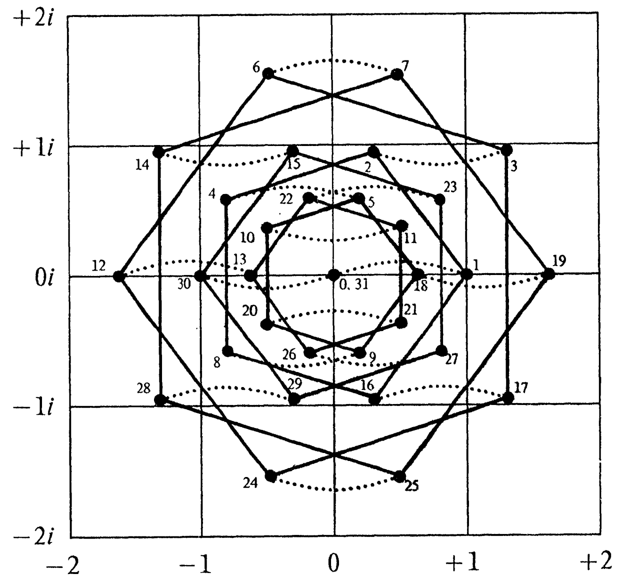


Figure 1: The shuffle-exchange graph on $32 = 2^5$ vertices mapped to the complex plane by $v \mapsto p_v(\omega)$. Vertices are labelled with $p_v(2)$. Dotted lines represent exchange edges, and solid lines represent shuffle edges.

vertices are not involved in the tie-breaking. Since there are n vertices in the shuffle-exchange graph, there are at most n/k regular k-gons of shuffle edges, and each of these k-gons crosses the real line twice. Since exchange edges are horizontal, they never cross the real line.

In order to define the bisection formally, we first partition the nonzero complex numbers as $\mathbb{C}^+ \cup \mathbb{C}^-$ where

$$\begin{aligned} \mathbb{C}^+ &= \{ z \in \mathbb{C} \mid \text{Im}(z) > 0 \} \cup \{ x \in \mathbb{R} \mid x > 0 \}, \\ \mathbb{C}^- &= \{ z \in \mathbb{C} \mid \text{Im}(z) < 0 \} \cup \{ x \in \mathbb{R} \mid x < 0 \}. \end{aligned}$$

The halves G' and G'' are defined by the regions to which vertices of the shuffle-exchange graph are mapped. The vertices for which $p_v(\omega) \in \mathbb{C}^+$ are assigned to V' and those for which $p_v(\omega) \in \mathbb{C}^-$ are assigned to V''. The remaining vertices, those for which $p_v(\omega) = 0$, are distributed arbitrarily but equally between V' and V''. Three types of edges are placed in E_S.

1. Exchange edges whose incident vertices are mapped to real numbers.
2. Shuffle edges whose incident vertices are mapped to the origin.
3. Shuffle edges between vertices v and v' such that $p_v(\omega) \in \mathbb{C}^+$ and $p_{v'}(\omega) \in \mathbb{C}^-$.

It can be seen by inspection that E_S is a superset of the set of edges that connect V' to V''. Edges not in E_S are allocated to E' or E'' according as their incident vertices are in V' or V''.

To see that $|V'| = |V''|$, consider for any vertex v the vertex $\mathbb{C}(v)$ obtained by complementing every index in the vector v. This relationship can be restated in terms of characteristic polynomials as

$$p_{\mathbb{C}(v)}(x) = (x^{k-1} + x^{k-2} + \ldots + 1) - p_v(x).$$

Because the sum of all kth roots of unity is zero, it follows that $p_v(\omega) = -p_{\mathbb{C}(v)}(\omega)$. Therefore, the correspondence $v \leftrightarrow \mathbb{C}(v)$ is a one-to-one correspondence between the vertices mapped to $\mathbb{C}^+$ and those mapped to $\mathbb{C}^-$. This proves that this partition is a bisection as was claimed. The cardinality of E_S is the width of the bisection and is bounded by the following theorem.

Theorem 3: For any positive integer k, there is a bisection S of the shuffle-exchange graph on $n = 2^k$ vertices such that the width of S is at most $6(n/k)$.

Proof. Let S be the bisection described above, and consider the three types of edges that compose E_S. We will bound each of the three types by the quantity $2(n/k)$.

Each of the type 3 edges is a shuffle edge incident on vertices mapped to nonzero complex numbers, and each such vertex belongs to a necklace of exactly k vertices which are mapped to nonzero numbers. Since the total number of vertices in the shuffle-exchange graph is n, there can be at most n/k such necklaces. The shuffle edges in each of these necklaces form a regular k-gon centered at the origin, and thus only two of these edges can cross the real line, in the sense of having one incident vertex mapped to $\mathbb{C}^+$ and the other to $\mathbb{C}^-$. Thus there can be at most $2(n/k)$ type 3 edges.

The same argument can be used to bound the number of type 1 edges. There are at most $2(n/k)$ vertices mapped to nonzero real numbers. Since every exchange edge whose incident vertices are mapped to real numbers has at least one of these vertices mapped to a nonzero real number, there can be no more than $2(n/k)$ type 1 edges.

Finally, the number of type 2 edges can be bounded by the number of type 1 edges by observing that for each shuffle edge $(v, \sigma(v))$ whose incident vertices are mapped to the origin, the exchange edge $(\sigma(v), \epsilon(\sigma(v)))$ is a type 1 edge. □

We now pause to examine an interesting by-product of these counting arguments, a result on the combinatorial problem of necklace enumeration. A necklace is a string of k pearls, where each pearl may be one of c colors. Two necklaces are considered equivalent if one can be rotated to form the other, but not if they are only reflections. It is well-known [7] that the number of necklaces of k pearls in c colors is

$$(1/k) \sum_{d\,|\,k} c^{k/d}\, \phi(d). \tag{7}$$

In this formula $\phi(d)$ is Euler's totient function, the number of positive integers not exceeding d that are relatively prime to d. Although it appears that the term for $d = 1$ in (7) might dominate the summation, it is not apparent that the contribution of the other terms is insignificant. However, the following corollary to Theorem 3 shows that this term is asymptotically dominant.

Corollary: The number of necklaces of $\{0, 1, \ldots, c-1\}^k$ lies between c^k/k and $((c+1)/(c-1))\,(c^k/k)$.

Proof. The definitions of the σ and ϵ permutations may be extended to $\{0, 1, \ldots, c-1\}^k$ as follows.

$$\epsilon(b_{k-1}\, b_{k-2} \ldots b_1\, b_0) = b_{k-1}\, b_{k-2} \ldots b_1\, (b_0{+}1 \bmod c),$$

$$\sigma(b_{k-1}\, b_{k-2} \ldots b_1\, b_0) = b_{k-2} \ldots b_1\, b_0\, b_{k-1}.$$

The characteristic polynomial is defined as before (notice that now $p_v(c)$ is the vector v considered as a number expressed in base c notation), and the argument of Theorem 3 can be adapted to show that the function $v \mapsto p_v(\omega)$ maps at most $2c^k/(c-1)k$ elements of $\{0, 1, \ldots, c-1\}^k$ to zero and that the remainder lie in necklaces of k elements. □

5. Dissecting the Shuffle-Exchange Graph

In the previous section, we presented a bisection of the shuffle-exchange graph on $n = 2^k$ vertices. In this section we will show that when k is even, the structure of the halves is similar to the structure of the original shuffle-exchange graph. This similarity is captured in the notion of an *m-cyclic subgraph* of the shuffle-exchange graph, and it is shown that the halves are $k/2$-cyclic subgraphs. The bisection from Theorem 3 can be modified to bisect m-cyclic subgraphs when m is even. Thus when k is a power of two, this approach can be used iteratively to construct a complete dissection of the shuffle-exchange graph.

An *m-cyclic subgraph* is a subgraph of the shuffle-exchange graph whose vertices are partitioned into disjoint m-cycles. Vertices not appearing in these m-cycles are also allowed, but such vertices must be isolated, not incident on any edge in the subgraph. If a shuffle edge $(v, \sigma(v))$ appears as an edge of the m-cyclic subgraph, it must occur between adjacent vertices of one of the m-cycles, and the exchange edge $(\sigma(v), \epsilon(\sigma(v)))$ must be an edge of the m-cyclic subgraph as well.

The reader should be warned that m-cyclic subgraphs are nothing more than a vehicle for extending the bisection of the shuffle-exchange graph to a dissection. The definition has been carefully crafted so that the proof of Theorem 3 will apply to them and so that their separator exhibits the closure property.

Lemma 4: When k is even, the halves G' and G'' produced by the bisection from Theorem 3 are $k/2$-cyclic subgraphs.

Proof. Without loss of generality, we show this for G' only. The vertices that are mapped to zero by $v \mapsto p_v(\omega)$ have no incident edges (are isolated), but every other vertex of G' occurs in some sequence $(v_0, \ldots, v_{k/2-1})$ that arose from cutting a necklace of k vertices in half. Since any necklace of k vertices is a k-cycle, and $k/2$ divides k, Lemma 2 ensures that this sequence is a $k/2$-cycle. Thus we have demonstrated the first requirement for G' to be an $k/2$-cyclic subgraph: every vertex not in an m-cycle is isolated.

We must now show that if a shuffle edge $(v, \sigma(v))$ appears as an edge in G', then it occurs between adjacent vertices of one of the m-cycles, and furthermore, that then the exchange edge $(\sigma(v), \epsilon(\sigma(v)))$ is also in G'. It is clear that the first condition is satisfied. The second condition can be demonstrated by observing that both v and $\sigma(v)$ are mapped to $\mathbb{C}^+$. Since the point $p_{\sigma(v)}(\omega)$ can be obtained from $p_v(\omega)$ by a counterclockwise rotation of $2\pi/k < \pi$ radians about the origin, it is impossible for $\sigma(v)$ to be mapped to the real line. The set of removed edges E_S contains only those exchange edges whose incident vertices are mapped to real points, which means that $(\sigma(v), \epsilon(\sigma(v)))$ must be in E'. □

When m is even, the bisection from Theorem 3 can be generalized to a bisection of an arbitrary m-cyclic subgraph. Let $\omega_m = e^{2\pi i/m}$ and consider the function $v \mapsto p_v(\omega_m)$. Since ω_m is a root of x^m-1, the congruence (6) between adjacent vertices of m-cycles becomes the equality $p_{v_j}(\omega_m) = \omega_m p_{v_{j-1}}(\omega_m)$. This means that if any vertex of an m-cycle is mapped to a nonzero complex number, all the m vertices of the m-cycle are mapped to distinct points evenly spaced on a circle about the origin. Equation (5) applies as before to show that vertices connected by an exchange edge are mapped to complex numbers which differ by one.

Let G be an arbitrary m-cyclic subgraph of a shuffle-exchange graph on $n = 2^k$ vertices, and suppose that m is even. In order to construct a bisection of G, the vertices of the m-cycles of G are assigned to V' or V'' according as they are mapped by $v \mapsto p_v(\omega_m)$ to $\mathbb{C}^+$ or $\mathbb{C}^-$. The remaining vertices of G are those vertices that are mapped to the origin and those that are isolated. These may be divided arbitrarily but equally between V' and V''. As with the bisection from Theorem 3, E_S consists of three types of edges.

1. Exchange edges whose incident vertices are mapped to real numbers.
2. Shuffle edges whose incident vertices are mapped to the origin.
3. Shuffle edges between vertices v and v' such that $p_v(\omega_m) \in \mathbb{C}^+$ and $p_{v'}(\omega_m) \in \mathbb{C}^-$.

The remaining edges are assigned to E' or E'' depending on whether their incident vertices are in V' or V''.

Unlike before, however, the correspondence $v \leftrightarrow \mathbb{C}(v)$ cannot be used to show that $|V'| = |V''|$, since v may be a vertex of G when $\mathbb{C}(v)$ is not. But because m is even, the equality $p_{v_j}(\omega_m) = -p_{v_{j+m/2}}(\omega_m)$ holds for vertices v_j and $v_{j+m/2}$ in the same m-cycle, and the correspondence $v \leftrightarrow v_{j+m/2}$ suffices to show that this partition is a bisection. The following lemma provides a bound for the width of the bisection.

Lemma 5: Let m be even, and let G be an m-cyclic subgraph on t vertices. There is a bisection S that bisects G into $m/2$-cyclic subgraphs and has width at most $6t/m$.

Proof. Let S be the bisection just described. Its width can be bounded by showing that there are at most $2t/m$ of each of the three types of edges in E_S. This bound holds for type 3 edges because there can be at most t/m disjoint m-cycles in G and no more than two type 3 edges per m-cycle. Since each type 1 edge has at least one incident vertex mapped to a nonzero real number, and there are at most two such vertices per m-cycle, the bound holds for these edges. Finally, for any type 2 edge $(v, \sigma(v))$, the edge $(\sigma(v), \epsilon(\sigma(v)))$ is a type 1 edge because G is an m-cyclic subgraph. Thus there can be no more type 2 edges than type 1 edges, and the bound on the width of the bisection is proved. It should be remarked here that the definition of m-cyclic subgraphs was specifically constructed in order to establish this correspondence between type 1 and type 2 edges.

To prove that the halves of the bisection are $m/2$-cyclic subgraphs, observe that the bisection S isolates those vertices that are in m-cycles mapped to the origin, and splits the other m-cycles into pairs of $m/2$-cycles. Since shuffle edges appear only between adjacent vertices of m-cycles, this adjacency is preserved in the $m/2$-cycles. The only exchange edges removed by the bisection are those whose incident vertices are mapped to real numbers, and hence the argument of Lemma 4 can be used to show that if $(v, \sigma(v))$ is in one of the halves, then $(\sigma(v), \epsilon(\sigma(v)))$ is also in the half. □

We are now ready to combine this bisection with the bisection from Theorem 3 into a dissection of the shuffle-exchange graph on $n = 2^k$ vertices for the case when k is a power of two. Recall from Section 2 that to dissect this graph, we need to find a class of subgraphs that has a separator with the closure property. The next theorem provides such a class.

Theorem 6: If k is a power of two, then there is a dissection $\mathcal{S}_n$ of the shuffle-exchange graph on $n = 2^k$ vertices such that any bisection in $\mathcal{S}_n$ which bisects an m-vertex graph has width at most

$$f_n(m) = \begin{cases} 6n/k & \text{if } m > n/k, \\ 0 & \text{otherwise.} \end{cases} \tag{8}$$

Proof. Let $\mathcal{G}_n$ be the class of subgraphs consisting of *i*) the shuffle-exchange graph itself, *ii*) its $k/2^j$-cyclic subgraphs that have $n/2^j$ vertices, for $j = 1, \ldots, (\lg k)-1$, and *iii*) its subgraphs that have no edges. Correspondingly, the separator $\mathcal{S}_n$ consists of *i*) the bisection of the shuffle-exchange graph from Theorem 3, *ii*) the bisections of its $k/2^j$-cyclic subgraphs from Lemma 5, and *iii*) arbitrary bisections of the totally disconnected subgraphs. To see that the closure property holds for $\mathcal{S}_n$, we first observe that the halves of the shuffle-exchange graph are $k/2$-cyclic subgraphs with $n/2$ vertices. For $j = 1, \ldots, (\lg k)-2$, the halves of the $k/2^j$-cyclic subgraphs with $n/2^j$ vertices are $k/2^{j+1}$-cyclic subgraphs with $n/2^{j+1}$ vertices. When $j = (\lg k)-1$ the bisection from Lemma 5 uses the mapping $v \mapsto p_v(\omega_2)$ to bisect 2-cyclic subgraphs. Since $\omega_2 = -1$, all vertices are mapped to real numbers, and thus the halves consist entirely of isolated vertices.

The bisection of the shuffle-exchange graph from Theorem 3 has width $6(n/k)$. For $j = 1, \ldots, (\lg k)-1$, the bisection from Lemma 5 bisects a $k/2^j$-cyclic subgraph of $n/2^j$ vertices with width $6(n/2^j)/(k/2^j) = 6(n/k)$. The totally disconnected graphs can be bisected with zero width. □

6. Laying Out the Shuffle-Exchange Graph

Given a dissection of an arbitrary graph, the divide-and-conquer technique of [2] can produce a VLSI layout whose area is related to the bisection widths of the graphs in the dissection. The VLSI model used is that of [9], and its important attributes are that wires have a minimum width and that only a constant number may cross at a point. In this section the results of Section 5 are applied to produce an $O(n^2/\lg n)$ area VLSI layout for an n-vertex shuffle-exchange network.

The technique of [2] constructs a layout for a general graph G by first bisecting G and laying out the halves recursively. The halves are then placed side-by-side, and the edges that were removed to bisect G are routed between the halves. The layout area can therefore be described as a recurrence in the area of the halves and the area required to route the edges removed by the bisection. This latter quantity is a function of the bisection widths in the dissection of G because the length and width of the layout increase by a constant amount for each edge routed.

The particulars of how the area recurrence arises from this construction are described more fully in [2]. Some solutions to the recurrence are also given in that paper, but the bisection width bound $f_n(m)$ from equation (8) fails to satisfy certain conditions that are assumed for those solutions. Therefore, we give the area recurrence from [2] without further justification, but present its solution in detail.

Let $A_n(m)$ be the area of the layout of an m-vertex graph in the dissection of Theorem 6 (thus $A_n(n)$ is the area of the original shuffle-exchange graph). We express $A_n(m)$ in terms of $f_n(m)$ from equation (8). For the initial condition of the area recurrence, $A_n(1)$ is a constant, and for $1 < m \leq n$,

$$A_n(m) = [\sqrt{2A_n(m/2)} + f_n(m)]^2. \tag{9}$$

The recurrence can be solved by taking the square root of both sides and then substituting $L_n(m)$ for $\sqrt{A_n(m)}$. For $1 < m \leq n$ this yields

$$L_n(m) = \sqrt{2}\, L_n(m/2) + f_n(m).$$

Iterating this recurrence and recalling that $n = 2^k$, we have

$$\begin{aligned} L_n(n) &= f_n(n) + \sqrt{2} f_n(n/2) + 2f_n(n/4) + \ldots \\ &\quad + \sqrt{2}^{\,k-1} f_n(2) + \sqrt{2}^{\,k} f_n(1) + \sqrt{n}\, L_n(1) \\ &\leq (6n/k)[1 + \sqrt{2} + \ldots + \sqrt{2}^{\,\lg k}] \\ &\quad + \sqrt{n}\, L_n(1) \qquad\qquad (10) \\ &= (6n/k)[(\sqrt{2}^{\,(\lg k)+1} - 1)/(\sqrt{2} - 1)] \\ &\quad + \sqrt{n}\, L_n(1) \\ &= O((n/k)\sqrt{k}) \\ &= O(n/\sqrt{k}). \end{aligned}$$

The reason the sum of the powers of $\sqrt{2}$ goes only as far as $\lg k$ in line (10) is that $f_n(m)$ is zero after this point. Since $A_n(n)$ is the square of $L_n(n)$, the area of the layout is $O(n^2/k)$.

This technique has been used in Figure 2 to lay out a shuffle-exchange network on 256 vertices. Only one fourth of the layout is shown, and the dissection that was used differs slightly from the one in Section 5. Instead of removing exchange edges, the arbitrary divisions among vertices mapped to zero are chosen so that $\varepsilon(v)$ is in the same component as v, and the two are placed together.

7. Conclusion

We have developed an extraordinary amount of machinery in order to construct an $O(n^2/k)$ area layout for the shuffle-exchange graph on $n = 2^k$ vertices, and indeed, we have only been able to show this upper bound for the case when k is a power of two. It may be that this bound holds when k is not a power of two, but we have not been able to prove this. For the time being, the best general upper bound seems to be Thompson's $O(n^2/\sqrt{k})$ bound.

In any event, a gap remains between either of these upper bounds and the best known lower bound of $\Omega(n^2/k^2)$ which is also given by Thompson. This lower bound is proved in [9] by showing that the minimum bisection width of the shuffle-exchange graph must be $\Omega(n/k)$ and that the area of any graph layout must be at least the square of the minimum bisection width of the graph. Theorem 3 shows that this $\Omega(n/k)$ lower bound for bisection of the shuffle-exchange graph can be achieved, even though the dissection based on this bisection does not achieve the $\Omega(n^2/k^2)$ lower bound for layout area. This is because the bisection width $f_n(m)$ does not immediately decrease as m decreases from n. It may be that an improved lower bound for the layout area will be based on the notion of a minimum dissection, where the width of every bisection in any dissection can be bounded from below.

On the other hand, it may be that an $O(n^2/k^2)$ area layout does exist for the shuffle-exchange graph, as does one for the *cube-connected-cycles* (CCC) network of Preparata and Vuillemin [6]. The CCC is the graph that arises from a boolean hypercube of d dimensions when each vertex is replaced by a cycle of d vertices. Many of the problems that can be solved quickly using the shuffle-exchange interconnection can also be solved quickly using the CCC. But despite the fact that a smaller layout is known for the CCC, descriptions of algorithms for the CCC tend to be more complicated. The discovery of an $O(n^2/k^2)$ area layout for the shuffle-exchange graph would therefore favor the shuffle-exchange graph as the network of choice and would allow the many algorithms already designed for this network to be applied directly in optimal VLSI implementations. But until such a layout is found —if ever one is found—the CCC will continue to have the edge.

In conclusion, we believe that characteristic polynomials provide a useful way of viewing the shuffle-exchange network, and we believe that this approach goes beyond the particular technical results presented here. Characteristic polynomials unveil properties of the shuffle-exchange graph that are obscured by the classical approach of relating the vertices to integers. We hope that the mechanisms we have developed to relate the topology of a particular graph to the algebra of polynomials will be exploited further.

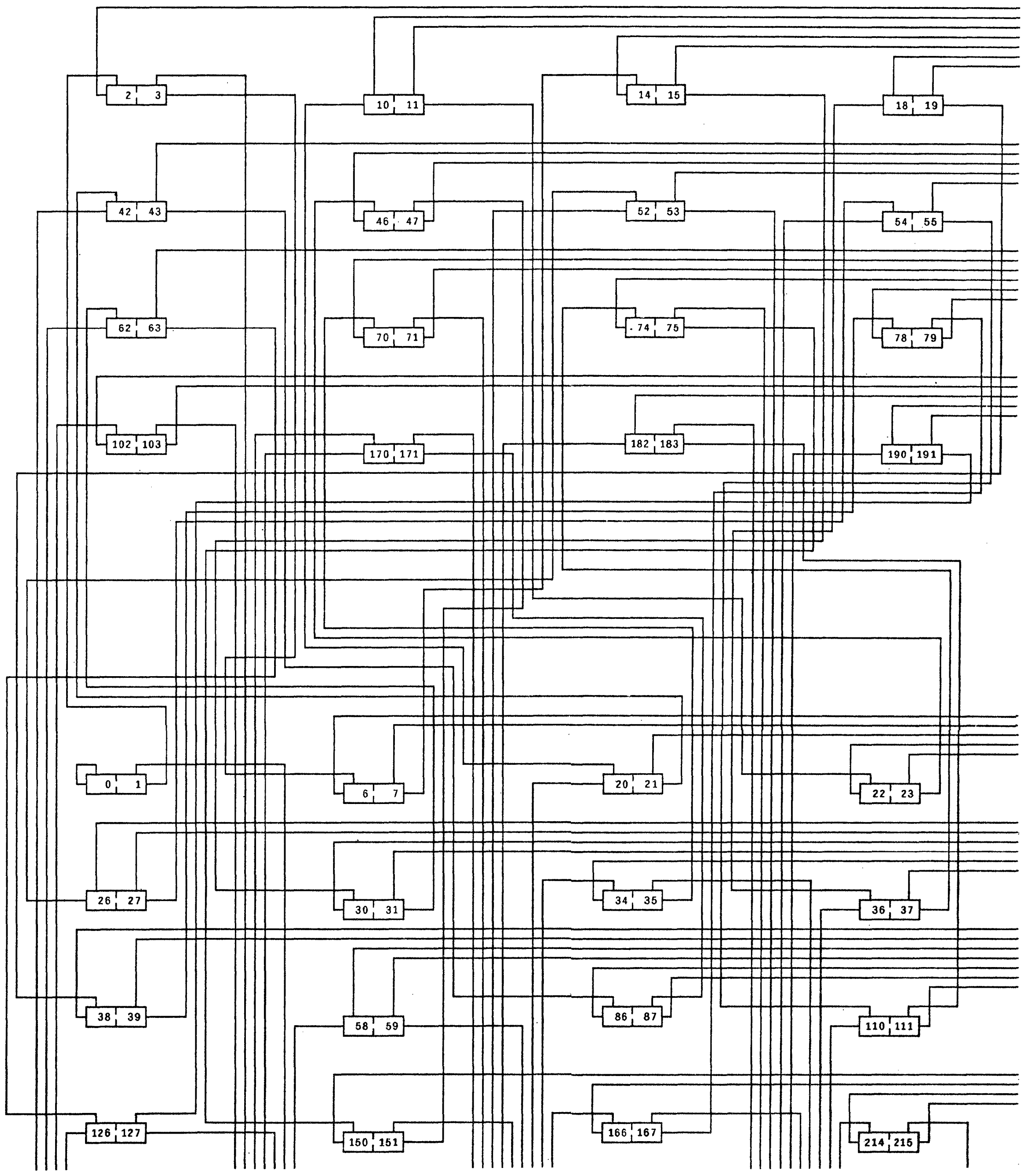


Figure 2: One fourth of a shuffle-exchange network

References

[1] M. R. Garey, D. S. Johnson, and L. Stockmeyer, "Some simplified polynomial complete problems," *6th Annual Symposium on Theory of Computing*, ACM, (April, 1974), pp. 47-63.

[2] C. E. Leiserson, "Area-efficient graph layouts (for VLSI)," *21st Annual Symposium on Foundations of Computer Science*, IEEE Computer Society, (October, 1980).

[3] R. J. Lipton and R. E. Tarjan, "A separator theorem for planar graphs," *A Conference on Theoretical Computer Science*, University of Waterloo, (August, 1977).

[4] R. J. Lipton and R. E. Tarjan, "Applications of a planar separator theorem," *18th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society, (October, 1977), pp. 162-170.

[5] C. A. Mead and L. A. Conway. *Introduction to VLSI Systems*, Addison-Wesley, (1980).

[6] F. P. Preparata and J. Vuillemin. *The cube-connected-cycles: a versatile network for parallel computation*, Technical Report 356, Institut de Recherche d'Informatique et d'Automatique, (June, 1979).

[7] J. Riordan, *An Introduction to Combinatorial Analysis*, John Wiley & Sons, Inc., (1958).

[8] H. S. Stone, "Parallel processing with the perfect shuffle," *IEEE Transactions on Computers*, C-20, 2, (February, 1971), pp. 153-161.

[9] C. D. Thompson, *A Complexity Theory for VLSI*, Ph.D. Thesis, Carnegie-Mellon University Computer Science Department, (1980).

TOWARD A GENERALIZATION OF TWO AND THREE-PASS MULTISTAGE, BLOCKING INTERCONNECTION NETWORKS

Abraham Shimor and Smil Ruhman
The Weizmann Institute of Science
Department of Applied Mathematics
Rehovot, Israel

Abstract

Blocking, multistage networks can realize only a fraction of the N! permutations (interconnections) possible. The minimum number of passes required to perform arbitrary permutations is an important parameter of every network.

We define four distinct classes of networks capable of performing any permutation in two passes, the lowest limit possible. These classes stem from a generalization of the Baseline network (known to be a two-pass network) by three of its main properties. Two of the classes are shown to be populated and examples of each are given. Whether the other two classes are empty is not clear, but this question is shown to be linked to another open question, namely the possibility of performing all permutations in two passes on the shuffle-exchange network. Using the lowest known bound for the shuffle-exchange, we define two classes of three-pass networks and demonstrate the existence of many members in each class. Finally, we show that some of the better known networks belong to the above classes. Beyond the results reported, questions and areas for additional research are identified.

I. Introduction

An important issue in the architecture of SIMD arrays is the choice of a flexible connection network for interprocessor (or processor-memory) communication. The requirement of cost-effectiveness along with high performance led to consideration of blocking multistage interconnection networks. Such a network of size N (N inputs x N outputs) consists of $\log_2 N$ stages, each comprising N/2 elementary 2 input x 2 output, two-state switches. Each stage is preceded or followed by a fixed wiring pattern that connects it to the adjacent stage or to the outside. Clearly, the maximum number of "admissible" permutations (realizable in a single pass) on such networks is $\sqrt{N^N} = 2^{\frac{N}{2}\log_2 N}$, a small part of the N! arbitrary permutations that exist.

A number of networks have been suggested [1, 4, 5, 6], each characterized by its set of exactly $\sqrt{N^N}$ admissible permutations. For the sake of flexibility, it is desirable to realize arbitrary permutations on blocking networks, even if this requires multiple passes. Wu and Feng [1, 2] suggested the Baseline network which is capable of realizing arbitrary permutations in two passes, the minimum possible. Recently, Parker [8] has proven that the shuffle-exchange network can perform arbitrary permutations in up to three passes, though it is not known whether this is the minimum upper bound. The same results had been shown by the authors [3] in two ways: a constructive proof by emulation of Beizer's [9] network, and a shorter algebraic proof based on the properties of the Baseline network [1, 2].

The wide range of networks proposed, and the seemingly unique characteristics of some, suggest the need for a general theory of blocking interconnection networks. Siegel [6] and Wu and Feng [1, 2] made significant contributions in this direction. In view of the two-pass property of the Baseline and the three-pass interim upper bound of the shuffle exchange, we raise the following questions:

(1) Are there other networks, different from either the Baseline or the shuffle exchange, which can perform arbitrary permutations in two or three passes?

(2) Given a selected subset of up to $\sqrt{N^N}$ permutations, is there a multistage, blocking network on which all the given permutations are admissible, while any other permutations can be realized in two or three passes?

In this paper we define several classes of two and three pass networks and demonstrate the existence of some of them. The second question remains open for the time being, but the treatment of question one may serve as a framework for additional research.

Notation

The number of input (output) lines of an interconnection network is denoted by $N=2^m$, where m is a positive integer. The input (and output) lines are numbered sequentially from 0 to N-1. This line number or address is denoted by a small letter, a, b, c ε {0,1,...,N-1}, whose binary expansion is given by $(a_{m-1}, a_{m-2}, \dots, a_0)$ and whose value is $a = \sum_{i=0}^{m-1} a_i 2^i$.

Permutations (interconnections) are designated by small letters p,q,s, where p(a) = b is a permutation that connects input line a to output line b. Superscripts indicate repetitive application of the permutation, the superscript -1 denoting the inverse permutation. Specific permutations used in the paper are defined below.

(1) Identity: $e(a) = a$
(2) Bit reversal: $\rho(a) = (a_0, a_1, \dots, a_{m-2}, a_{m-1})$
(3) Bit reversal excluding a_0 :
$r(a) = (a_1, a_2, \dots, a_{m-2}, a_{m-1}, a_0)$
(4) Perfect shuffle: $\sigma(a) =$
$(a_{m-2}, a_{m-3}, \dots, a_1, a_0, a_{m-1})$

A network and its set of admissible permutations

CH1569-3/80/0000-0337$00.75 © 1980 IEEE

will be denoted by a capital letter X, Y. Operations on networks are defined below.

$XY = \{pq \mid p \in X,\ q \in Y\}$

$X^{-1} = \{ p \mid p^{-1} \in X \}$

$sX = \{sp \mid p \in X\}$ and $Xs = \{ps \mid p \in X\}$

The group of all possible permutations on the integers $\{0,1,\ldots,N-1\}$ will be designated by S. Two specific networks of central interest in this paper are:

(1) The Baseline or Reverse Exchange network, B, is defined in [1]. A sketch of B for $N=8$ is given in Fig. 1.

(2) The Shuffle Exchange network, Ω, in which the wiring pattern preceding each stage is described by σ.

II. Two-pass networks

The Baseline interconnection network, B, introduced by Wu and Feng [1, 2] exhibits some interesting properties.

(a) $BB = S$, indicating that any arbitrary permutation can be performed in two passes through the network.

(b) $B = B^{-1}$, that is, $p_1 \in B$ implies $p_1^{-1} \in B$.

(c) $e \notin B$, the identity permutation (e) is not admissible on B, therefore B cannot contain any subgroup of S.

Of these properties the first one has particular significance from a practical point of view. Two questions are likely to arise with respect to the Baseline network and its characterisation by the above properties:

(a) Is the Baseline network unique among blocking multistage networks, or are there other different networks capable of performing any arbitrary permutation in two passes?

(b) Does the ability to perform any permutation in two passes imply either or both of the remaining properties?

In order to answer these questions, let us postulate the existence of four classes of "Baseline-like" networks, β_0 through β_3. X will be used to represent a connection network, as well as the set of admissible permutations on this network.

$$\beta_0 = \{X \mid XX = S;\ X = X^{-1};\ e \notin X\}$$
$$\beta_1 = \{X \mid XX = S;\ X = X^{-1};\ e \in X\}$$
$$\beta_2 = \{X \mid XX = S;\ X \neq X^{-1};\ e \notin X\}$$
$$\beta_3 = \{X \mid XX = S;\ X \neq X^{-1};\ e \in X\}$$

It is easily seen that these subsets are distinct: for every $i \neq j$; $i,j \in [0,1,2,3]$ $\beta_i \cap \beta_j = \emptyset$ (the empty set).

β_0 networks

Let X represent a connection network topologically equivalent to B. Applying the definition of topological equivalence introduced by Wu and Feng, this proposition implies that $X = p_1 B p_2$ or alternatively $B = p_1^{-1} X p_2^{-1}$ where $p_1, p_2 \in S$).

Let us concentrate on a particular subset of the networks topologically equivalent to B; we designate this subset as

$$\hat{\beta}_0 = \{X \mid X = p_1 B p_1^{-1};\ p_1 \in S\}$$

Lemma 1 $\hat{\beta}_0$ is a subset of β_0: $\hat{\beta}_0 \subseteq \beta_0$

Proof For every $X \in \hat{\beta}_0$

(1) $X^{-1} = (p_1 B p_1^{-1})^{-1} = p_1 B^{-1} p_1^{-1} = p_1 B p_1^{-1} = X$

(2) $XX = (p_1 B p_1^{-1})(p_1 B p_1^{-1}) = p_1 B B p_1^{-1} = S$

(3) Suppose that $e \in X$, since $B = p_1^{-1} X p_1$, this implies that $p_1^{-1} e p_1 = e \in B$ which leads to a contradiction, therefore $e \notin X$. Q.E.D.

Obviously $B \in \hat{\beta}_0$ ($p_1 = e$ in this case), therefore $\hat{\beta}_0$ is not an empty set. Furthermore, it is easily shown that $\hat{\beta}_0$ contains additional elements. Let us assume the opposite of this proposition. This would imply that for every $p_1 \in S$, $p_1 B p_1^{-1} = B$. Recalling the definition of a normal subgroup [7], the above means that B is a normal subgroup of S and this leads to a contradition, since (due to the fact that $e \notin B$) B cannot be a subgroup of S. Several examples of $\hat{\beta}_0$ class networks are described in Fig. 1.

β_2 networks

In [3] we have shown, that any arbitrary permutation $p \in S$, can be decomposed into the form $p_1 r p_2 = p$, where $p_1, p_2 \in \Omega$ (i.e. they are admissible on the shuffle-exchange network) and the permutation r (designated as $R_{(m)}$ in [3] is defined as,

$$r(a_{m-1},\ldots,a_0) = (a_1,a_2,\ldots,a_{m-1},a_0)\ .$$

Obviously $r = r^{-1}$. Alternatively we can state: $S = \Omega r \Omega$. Let E represent the network Ωr.

Lemma 2: $E \in \beta_2$

Proof: (1) $EE = \Omega r \Omega r = (\Omega r \Omega) r = S$

(2) $E \neq E^{-1}$ is proven by an example given in Fig. 2. We express E and E^{-1} in terms of B, using the identities given by Wu and Feng in [1], namely: $\Omega = B\rho$ and $\Omega^{-1} = \rho B$, together with the identity $r = \sigma\rho$. Hence $E = B\sigma^{-1}$ and $E^{-1} = \sigma B$.

(3) $e \notin E$, because if we assume the opposite, then the solution of the equation $e = qr$, $q \in \Omega$ yields $q = r$, but it can be shown from Lawrie's theorem 2 in [4] that $r \notin \Omega$. Q.E.D.

Similarly, we can show that $E^{-1} \in \beta_2$.

Let us now define set $\hat{\beta}_2$ as:

$$\hat{\beta}_2 = \{X \mid X = p_1 E p_1^{-1} = p_1 B \sigma^{-1} p_1^{-1};\ p_1 \in S\}.$$

Using the same method as was applied to $\hat{\beta}_0$ above, it can be shown that $\hat{\beta}_2$ is a non-empty subset of β_2. Several examples of $\hat{\beta}_2$-type networks are described in Fig. 3.

β_1 and β_3 networks

So far, no networks of these types have been identified, but neither has the possibility of their existence been disproved. Interestingly this so far undecided question is related to another unresolved problem: can any arbitrary permutation be performed in two passes on the shuffle-exchange network? The linkage between these problems arises from the following speculation.

If it is possible to perform all the permutations in two passes on the shuffle-exchange network, that is $\Omega\Omega = S$, then $\Omega \in \beta_3$, since

(1) $\Omega\Omega = S$

(2) $\Omega \neq \Omega$

(3) $e \in \Omega$

Furthermore, all networks defined as $p_1 \Omega p_1^{-1}$ would belong to β_3:

(1) $(p_1 \Omega p_1^{-1})(p_1 \Omega p_1^{-1}) = p_1 \Omega\Omega p_1^{-1} = S$

(2) $(p_1 \Omega p_1^{-1})^{-1} = p_1 \Omega^{-1} p_1^{-1} \neq p_1 \Omega p_1^{-1}$

(3) $e \in p_1 \Omega p_1^{-1}$, because $e = p_1 e p_1^{-1}$.

III. Three-pass Networks

It has been shown [3, 8] that the shuffle-exchange network can perform any arbitrary permutation in three passes. Following the generalization of the two-pass property of the Baseline network, the next logical step is to search for a new class of networks capable of performing any permutation in three passes. Let us designate this class as τ.

$$\tau = \{X \mid XXX = S\}$$

Parker [8] showed that $S = \Omega\rho\Omega$, and $\rho = \omega_2\,\omega_1$ so that $\omega_1, \omega_2 \in \Omega$ and $\omega_1 \Omega \in \Omega$. In [3] we had given a different factorization of ρ: $\rho = r_4 r_3$ so that $r_3, r_4 \in \Omega$ and $\Omega\, r_4 = \Omega$. Likewise, we also showed in [3] that $S = \Omega r \Omega$, where $r = r_2 r_1$ such that $r_1, r_2 \in \Omega$ and $\Omega r_2 = \Omega$. The permutations r_1, r_2, r_3, r_4 are defined below using a set of functions of the form $b_i = f_i(a, b_{m-1}, b_{m-2}, \ldots, b_{i+1}, b_{i-1}, \ldots, b_0)$ where $i = 0, 1, \ldots, m-1$. Also, $k = (m-1)\,\text{div}\ 2$ and $\ell = m\ \text{div}\ 2$, where div represents integer division.

$$r_1:\ b_i = \begin{cases} a_i \oplus a_{m-i} & \text{if } m \text{ even and } i > k+1 \\ & \text{or } m \text{ odd and } i > k \\ a_i \oplus b_{m-i} & \text{if } 0<i<k \\ a_i & \text{if } i = 0 \\ & \text{or } i=k+1 \text{ and } m \text{ is even} \end{cases}$$

$$r_2:\ b_i = \begin{cases} a_i \oplus a_{m-i} & \text{if } m \text{ even and } i>k+1 \\ & \text{or } m \text{ odd and } i>k \\ a_i & \text{otherwise} \end{cases}$$

$$r_3:\ b_i = \begin{cases} a_i \oplus a_{m-i-1} & \text{if } m \text{ is even and } i \geq \ell \\ & \text{or } m \text{ is odd and } i \geq \ell+1 \\ a_i \oplus b_{m-i-1} & \text{if } i<\ell \\ a_i & \text{if } m \text{ is odd and } i=\ell \end{cases}$$

$$r_4:\ b_i = \begin{cases} a_i \oplus a_{m-i-1} & \text{if } m \text{ is even and } i \geq \ell \\ & \text{or } m \text{ is odd and } i \geq \ell+1 \\ a_i & \text{otherwise.} \end{cases}$$

The subset of permutation $\overset{\leftarrow}{\Omega} = \{p \mid \Omega p = \Omega\ ;\ p \in \Omega\}$ including permutation r_2 and r_4, is worth some attention.

Lemma 3: $\overleftarrow{\Omega}$ is a subgroup of S.

Proof: (1) $\overleftarrow{\Omega} \subseteq \Omega$, therefore it is finite.

(2) Let $p_1, p_2 \in \overleftarrow{\Omega}$

$\Omega(p_1p_2) = (\Omega p_1)p_2 = \Omega p_2 = \Omega$

Hence $(p_1p_2) \in \overleftarrow{\Omega}$, in other words, $\overleftarrow{\Omega}$ is closed under multiplication.

Q.E.D.

(It is possible to show the existence of a similar subgroup $\overrightarrow{\Omega} = \{p \mid p\Omega = \Omega;\ p \in \Omega\}$; for example, Parker's ω_1 permutation is an element of $\overrightarrow{\Omega}$. Furthermore, $\overleftrightarrow{\Omega} = \overleftarrow{\Omega} \cap \overrightarrow{\Omega}$ is also a non-empty, non-trivial subgroup.)

The property associated with permutations in $\overleftarrow{\Omega}$ we call "right invariance", while $\overleftarrow{\Omega}$ is called the "right invariant subgroup". The scope of right invariance may be extended beyond the shuffle-exchange network.

Corollary 1. Any connection network X fulfilling the following two conditions has an associated right invariant subgroup, $\overleftarrow{X}$: (1) $e \in X$; (2) there exists at least one permutation p_1 so that $p_1 \in X$ and $Xp_1 = X$.

Let $\hat{\tau}_0$ represent a set of networks topologically equivalent to the Baseline network B, defined by:

$$\hat{\tau}_0 = \{X \mid X = p_1Bp_2\ ;\ p_1p_2 = p_2p_1 = \rho\}$$

$\hat{\tau}_0$ networks

Let $X \in \hat{\tau}_0$, then $X = p_1Bp_2$ where $p_1p_2 = p_2p_1 = \rho$. We can express B in terms of X: $B = p_1^{-1}Xp_2^{-1}$. Since $BB = S$, we obtain $X\rho X = S$. (A similar expression, $S = \Omega\rho\Omega$ led [3, 8] to the conclusion that the shuffle-exchange is a three-pass network.)

Lemma 4. All networks belonging to $\hat{\tau}_0$ are three-pass networks, that is $\hat{\tau}_0 \in \tau$.

Proof. We shall prove that ρ can be decomposed into a product of two permutations in the from $\rho = r_4'r_3'$ so that $r_3', r_4' \in X$ and $Xr_4' \in X$.

(1) Using the relationship between the Baseline and shuffle-exchange networks [1] we can modify the definition of X to $X = p_1\Omega\rho p_2$. Recalling the decomposition of ρ for the shuffle-exchange network ($\rho = r_4r_3$ where $r_4, r_3 \in \Omega$ $\Omega r_4 = \Omega$) we define: $r_4' = p_1r_4\rho p_2$ and $r_3' = p_1r_3\rho p_2$. Obviously $r_4', r_3' \in X$.

(2) $r_4'r_3' = (p_1r_4\rho p_2)(p_1r_3\rho p_2) =$
$(p_1((r_4(\rho(p_2p_1))r_3)\rho)p_2) = \rho$

(3) Let q' be an arbitrary permutation so that $q' \in X$. For every such q' there exists a permutation q such that $q' = p_1q\rho p_2$.

$q'r_4 = (p_1q\rho p_2)(p_1r_4\rho p_2) = p_1(qr_4)\rho p_2$

Since $qr_4 \in \Omega$ for any q (because $r_4 \in \overleftarrow{\Omega}$, hence $q'r_4' \in X$ for every $q' \in X$. Q.E.D.

$\hat{\tau}_1$ networks

The network $E = \Omega r \in \hat{\beta}_2$ is a two-pass network. We now define a new set $\hat{\tau}_1$ containing networks topologically equivalent to E (as well as to B, since $E = B\rho r$):

$$\hat{\tau}_1 = \{x \mid x = p_1Ep_2;\ p_1p_2 = p_2p_1 = r\}$$

From this definition, $E = p_1^{-1}Xp_2^{-1}$. Since $EE = S$, we conclude that $XrX = S$.

Lemma 5. All the networks belonging to $\hat{\tau}_1$ are three-pass networks: $\hat{\tau}_1 \subseteq \tau$.

Proof. As indicated earlier for the shuffle-exchange, r can be decomposed into a product of two permutations in the form $r = r_2r_1$ where $r_2, r_1 \in \Omega$ and $\Omega r_2 = \Omega$.

Following the proof of Lemma 4, it can be shown that the permutations $r_2' = p_1r_2rp_2$ and $r_1' = p_1r_1rp_2$ have the properties: $r_2'r_1' = r$; $r_2', r_1' \in X$; $Xr_2' \in X$. This proves the lemma.

Q.E.D.

It is easily shown that the shuffle-exchange network belongs to both $\hat{\tau}_0$ and $\hat{\tau}_1$:

$\Omega = e(\Omega\rho)eB\rho = e\epsilon\hat{\tau}0$, therefore $\Omega \in \hat{\tau}_0$

$\Omega = e(\Omega r)r = eEr$, hence $\Omega \in \hat{\tau}_1$.

Hence $\hat{\tau}_0 \cap \hat{\tau}_1 \neq \phi$; whether $\hat{\tau}_0 = \hat{\tau}_1$ seems to be a more difficult question. Other networks belonging to $\hat{\tau}_0$ may be found by the following lemma.

Lemma 6. $X \varepsilon \hat{\tau}_0$ implies $X^{-1} \varepsilon \hat{\tau}_0$

Proof. $X \varepsilon \hat{\tau}_0$ implies $X = p_1 B p_2$ where $p_1 p_2 = p_2 p_1 = \rho$.

Hence $X^{-1} = p_2^{-1} B^{-1} p^{-1} = p_2^{-1} B p_1^{-1}$. But $p_2^{-1} p_1^{-1} = p_1^{-1} p_2^{-1} = \rho$, therefore $X^{-1} \varepsilon \hat{\tau}_0$

Q.E.D.

Some examples of three-pass networks are given in Fig. 4. The equations $p_1 p_2 = p_2 p_1 = \rho$ and $p_1 p_2 = p_2 p_1 = r$ are worth some attention. They can be transformed to a more useful form, $p_2 \rho p_2^{-1} = \rho$ and $p_2 r p_2^{-1} = r$.

Let G_ρ and G_r represent the set of solutions to these equations respectively:

$$G_\rho = \{p_2 \mid p_2 \rho p_2^{-1} = \rho \; ; \; p_2 \varepsilon S\}$$

$$G_r = \{p_2 \mid p_2 r p_2^{-1} = r \; ; \; p_2 \varepsilon S\}$$

Lemma 7. G_ρ and G_r are subgroups in S.

Proof. (1) G_ρ and G_r are finite, since $G_\rho \subseteq S$ and $G_r \subseteq S$.

(2) Let $q_1, q_2 \varepsilon G_r$. Then $q_1 \rho q_1^{-1} = \rho$ and $q_2 \rho q_2^{-1} = \rho$.

Hence $(q_1 q_2) \rho (q_1 q_2)^{-1} = q_1 (q_2 \rho q_2^{-1}) q_1^{-1} = \rho$.

By similar treatment of G_r we conclude that both G_ρ and G_r are closed with respect to multiplication. Q.E.D.

We can establish a lower bound for the number of elements in these subgroups as follows: The permutation ρ acts as the identity permutation on those elements whose binary address is symmetric (e.g. 01011010 → 01011010). For a network of size $N = 2^m$ there are k_ρ elements with symmetric addresses, where

$$k_\rho = \begin{cases} 2^{\frac{m}{2}} & \text{if } m \text{ is even} \\ 2^{\frac{m+1}{2}} & \text{if } m \text{ is odd} \end{cases}$$

Therefore there are $(k_\rho)!$ permutations which act on the elements with symmetric addresses only, leaving the other elements undisturbed. It should be obvious that for any p_2 belonging to these $(k_\rho)!$ permutations, $p_2 \rho = \rho\, p_2$; hence $p_2 \rho p_2^{-1} = \rho p_2 p_2^{-1} = \rho$.

Similarly, for $p_2 \varepsilon G_r$

$$k_r = \begin{cases} 2^{\frac{m}{2}} & \text{if } m \text{ is even} \\ 2^{\frac{m-1}{2}} & \text{if } m \text{ is odd} \end{cases}$$

while the number of possible permutations is $(k_r)!$.

Another interesting question is how different are two unequal $\hat{\tau}_0$ networks? More precisely, let $X, Y \varepsilon \hat{\tau}_0$ and $X \neq Y$, what can we say about D, the number of permutations in the set $\{p \mid p \varepsilon X \text{ and } p \not\varepsilon Y\}$.

Lemma 8. For two unequal networks of size $N = 2^m$, D is not less than $\sqrt{N}^N / N^2$.

Proof. Since $X \neq Y$, there exists at least one connection of two input lines to two output lines which cannot be realized on Y , but can be realized on X . Naturally Y cannot realize any permutation which includes this connection. On the other hand, implementation of this specific connection of pairs on X requires the setting of no more than two exchange boxes (out of N/2) in each of the m stages. The maximum number of exchange boxes involved is therefore 2m , leaving the other $\frac{mN}{2} - 2m$ free. Therefore X can realize at least $2^{(\frac{mN}{2} - 2m)} = \sqrt{N}^N / N^2$ permutations each of which contains the connection-pair which cannot be realized on Y . Q.E.D.

Before we conclude our remarks on three-pass networks, let us return to right-invariant permutations. So far, we only made use of selected right-invariant permutations in proving the existence of classes $\hat{\tau}_0$ and $\hat{\tau}_1$. The practical significance of right- anf left-invariance lies in the fact, that the permutations having this property can be used to characterize classes of permuations performable in two passes. If $\overleftarrow{X}$ is the subset of right-invariant permutations on a network X , then all the permutations in the set $XX\overleftarrow{X}\cup X\overleftarrow{X}X$ can be performed in two passes. For example: $B = \Omega\rho = \Omega r_4 r_3 = \Omega r_3$ (because $r_4 \varepsilon \overleftarrow{\Omega}$) - the shuffle-exchange network can perform in two passes all the permutations admissible on the Baseline. In addition right- and left-invariance may be used to recognize or prove admissibility of a given permutation by its possible decomposition using the identity $X = X\overleftarrow{X}$. We proceed with some additional lemmas concerning right-invariance. Identical statements hold for left-invariance.

Lemma 9. The set of permutations admissible on a network belonging to $\hat{\tau}_0$ or $\hat{\tau}_1$ contains a right-invariant permutation subgroup.

Proof. The existence of a single right-invariant permutation for each network type has already been demonstrated: r_4' in $\hat{\tau}_0$ and r_2' in $\hat{\tau}_1$. In order to satisfy the conditions stated in Corollary 1, we have to show, that all $\hat{\tau}_0$ and $\hat{\tau}_1$ type networks contain the identity permutation. Let $X \varepsilon \hat{\tau}_0$, hence $X = p_1 \Omega \rho p_2$, where $p_2 p_1 = p_1 p_2 = \rho$. Assume $q \varepsilon \Omega$ such that $p_1 q \rho p_2 = e$. Then $q = p_1^{-1} p_2^{-1} \rho = e \ \varepsilon \ \Omega$. Similarly for $X \varepsilon \hat{\tau}_1$. Q.E.D.

Lemma 10. The right-invariant permutation subgroups of all networks in $\hat{\tau}_0$ are isomorphic to each other.

Proof. Let $X,Y \ \varepsilon \ \hat{\tau}_0$,where $X = p_1 \Omega \rho p_2$; $p_1 p_2 = p_2 p_1 = \rho$ and $Y = q_1 \Omega \rho q_2$; $q_1 q_2 = q_2 q_1 = \rho$

Since we are dealing with networks of the same size, there is a one-to-one mapping between X and Y: $Y = F(X) = q_1 p_1^{-1} X p_2^{-1} q_2$.

Let $s, s_1, s_2 \ \varepsilon \ X$. Let $s', s_1', s_2' \ \varepsilon \ Y$, defined as $s' = F(s)$, $s_1' = F(s_1)$, $s_2' = F(s_2)$. To prove that $\overleftarrow{X} \approx \overleftarrow{Y}$ ($\overleftarrow{X}$ is isomorphic to $\overleftarrow{Y}$) we must show that $F(s_1), F(s_2) \varepsilon \overleftarrow{Y}$ for any $s_1, s_2 \ \varepsilon \ \overleftarrow{X}$

$F(s_1)F(s_2) = F(s_1 s_2)$ [7].

(1) $s's_1' = q_1 p_1^{-1} s p_2^{-1} q_2 q_1 p_1^{-1} s_1 p_2^{-1} q_2 =$

$q_1 p_1^{-1} (ss_1) p_2^{-1} q_2$ '

because $p_2^{-1} q_2 q_1 p_1^{-1} = p_2^{-1} \rho p_1^{-1} = e$

The above is true for any $s' \ \varepsilon \ Y$, therefore, since $Xs_1 = X$ we conclude that $Ys_1' = Y$ or $s_1' \ \varepsilon \ \overleftarrow{Y}$.

(2) Similarly, we can show that $s_2' \ \varepsilon \ Y$.

(3) $s_1's_2' = F(s_1)F(s_2) = q_1 p_1^{-1} s_1 p_2^{-1} q_2 q_1 p_1^{-1} s_2 p_2^{-1} q_2 = q_1 p_1^{-1} (s_1 s_2) p_2^{-1} q_2 = F(s_1 s_2)$. Q.E.D.

By the same method we can prove:

Lemma 11. The right-invariant permutation subgroups of networks in $\hat{\tau}_1$ are isomorphic to each other.

Since $\Omega \varepsilon \hat{\tau}_0$ and $\Omega \varepsilon \hat{\tau}_1$, it follows directly from the last two lemmas that there is the same number of right-invariant permutations in any network belonging to $\hat{\tau}_0$ or $\hat{\tau}_1$.

IV. Summary and suggestions for further research.

By generalizing the Baseline network, four distinct subclasses of two-pass interconnection networks were defined. The existence of many different networks in two of these subclasses was proven and exemplified. It was also shown that many different networks exist capable of performing arbitrary permutations in no more than three passes, thereby generalizing the property that had been specifically proven [3, 8] for the shuffle-exchange network. Table 1 below shows that some of the most widely known multistage blocking networks are three pass networks.

We should emphasise several points about the class of three-pass netwroks. When performing arbitrary permutations in three passes, the middle (second) pass realizes a constant permutation specific to the network used but independent of the overall permutation being implemented. Hence the number of variable (permutation dependent) control bits is equal to that of two-pass networks. Furthermore, a large set of permutations can be implemented in less than three passes. Finally, it should be remembered that any of the three-pass networks (including the shuffle-exchange) may turn out to be two-pass, since it was defined as capable of performing an arbitrary permutation in no more than three passes.

Table 1. Classification of some well-known networks under individual switch control.

Network	Relation to B or E	Subclass	Number of passes
B, Baseline	eBe	$\hat{\beta}_0$	2
Ω, Shuffle-exchane (Omega)	eBρ , eEr	$\hat{\tau}_0$ $\hat{\tau}_1$	3
C^{-1}, Inverse Indirect Binary n-Cube	eBρ , eEr	$\hat{\tau}_0$ $\hat{\tau}_1$	3
F^{-1}, Inverse Flip network	eBρ , eEr	$\hat{\tau}_0$ $\hat{\tau}_1$	3
Ω^{-1}, Inverse SE (Inverse Omega)	ρBe	$\hat{\tau}_0$	3
C, Indirect Binary n-Cube	ρBe	$\hat{\tau}_0$	3
F, Flip network	ρBe	$\hat{\tau}_0$	3

The methods used in this paper and the lemmas proven lead to some additional interrelated questions:

(1) How many different networks are there in each of the subclasses $\hat{\beta}_0, \hat{\beta}_1, \hat{\tau}_0$ and $\hat{\tau}_1$?

(2) Are there two or three-pass netwoks not topologically equivalent to the Baseline network?

(3) Is every permutation admissible on some two or three-pass network?

(4) Is it possible to synthesise a two or three-pass network of size N which admits an arbitrary subset of S not exceeding $\sqrt{N}^N$ permutations?

(5) Are there iterative (single-stage recirculating) networks other than Ω and Ω^{-1}, capable of performing arbitrary permutations in two or three passes?

REFERENCES

[1] Wu, C., and Feng, T., "The reverse exchange interconnection network ", 1979 International Conf. on Parallel Processing, pp. 160-174.

[2] Wu, C., and Feng, T., "Routing techniques for a class of multistage interconnection networks", 1978 International Conference on Parallel Processing, pp. 197-205.

[3] Shimor, A., and Ruhman, S., "Emulation of universal interconnection networks with the shuffle-exchange", Report No. ASR2, Dept. of Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel, August 1979.

[4] Lawrie, D., "Access and alignment of data in an array processor", IEEE Trans. on Comput., Vol. C-24, No. 12, Dec. 1975, pp. 1145-1175.

[5] Pease, M.C., "The indirect binary n-cube microprocessor array", IEEE Trans. on Comput., Vol. C-26, May 1977, pp. 458-473.

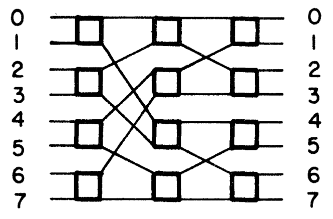
[6] Siegel, H.J., "The universality of various types of SIMD machine interconnection networks", Fourth Annual Symp. Computer Architecture, Mar. 1977, pp. 70-79.

[7] Herstein, I.N., "Topics in algebra", XEROX College Publishing, Lexington, MA. 1975.

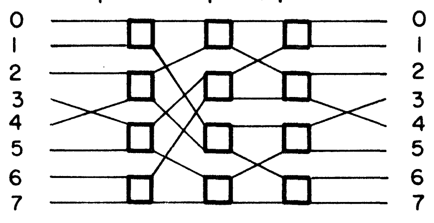
[8] Parker, D.S., "Notes on shuffle/exchange type switching networks", IEEE Trans. on Comput., Vol. C-29, No. 3, March 1980, pp. 213-222.

[9] Beizer, B., "The analysis and synthesis of signal switching networks", Proc. of the Symp. on Mathematical Theory of Automata, New York, April 1962, Polytechnic Press of the Polytech. Inst. of Brooklyn, pp. 563-576.

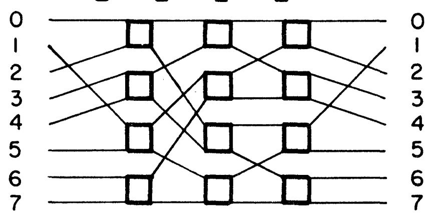
B



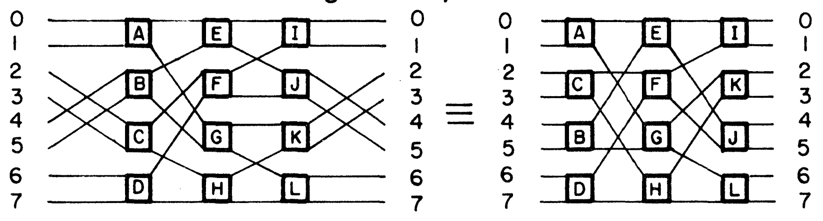
$X_1 = p\,Bp_1^{-1};\ p_1 = (34)$



$X_2 = p_2 Bp_2^{-1};\ p_2 = (1432)$



$X_3 = rBr\ ;\ \ r = (24)(35)$



$X_4 = p_3 Bp_3^{-1};\ \ p_3 = (0246)(1357)$

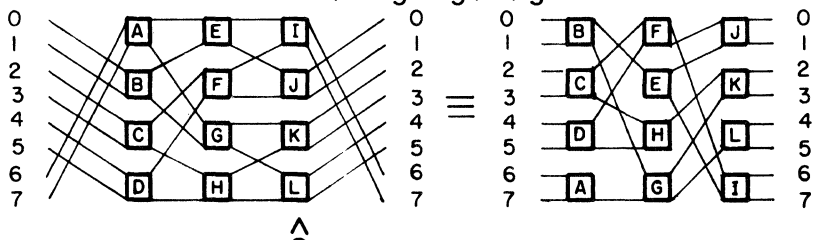


Table of Differences

Connection	Network Designation B	X_1	X_2	X_3	X_4
$0 \rightarrow 0$, $1 \rightarrow 1$	N	Y	N	N	N
$2 \rightarrow 0$, $3 \rightarrow 1$	N	N	Y	N	N
$3 \rightarrow 0$, $1 \rightarrow 1$	N	Y	Y	Y	N
$0 \rightarrow 0$, $2 \rightarrow 1$	N	N	Y	Y	Y

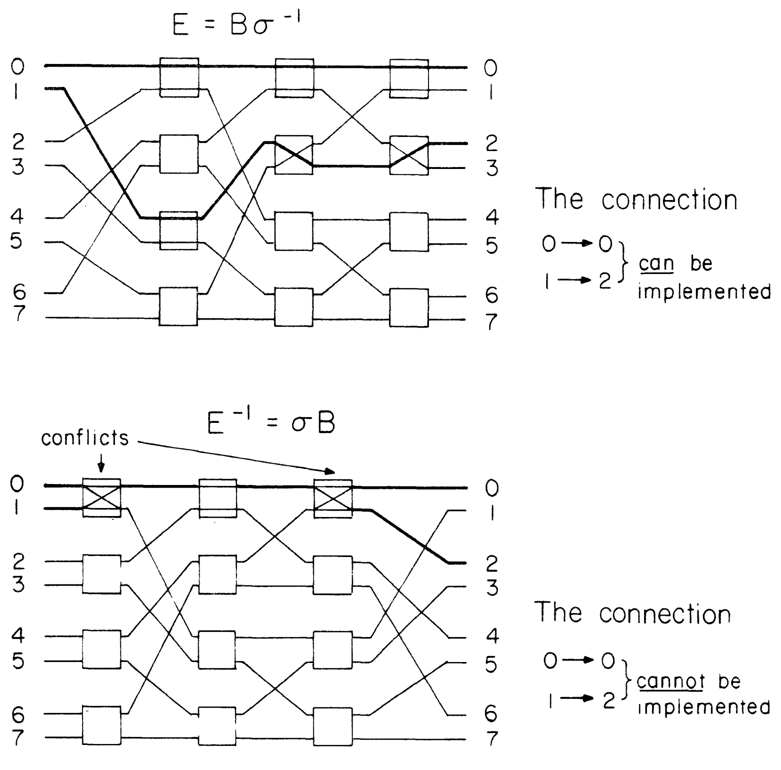
N = connection not possible

Y = connection possible

Examples of $\hat{\beta}o$ type networks

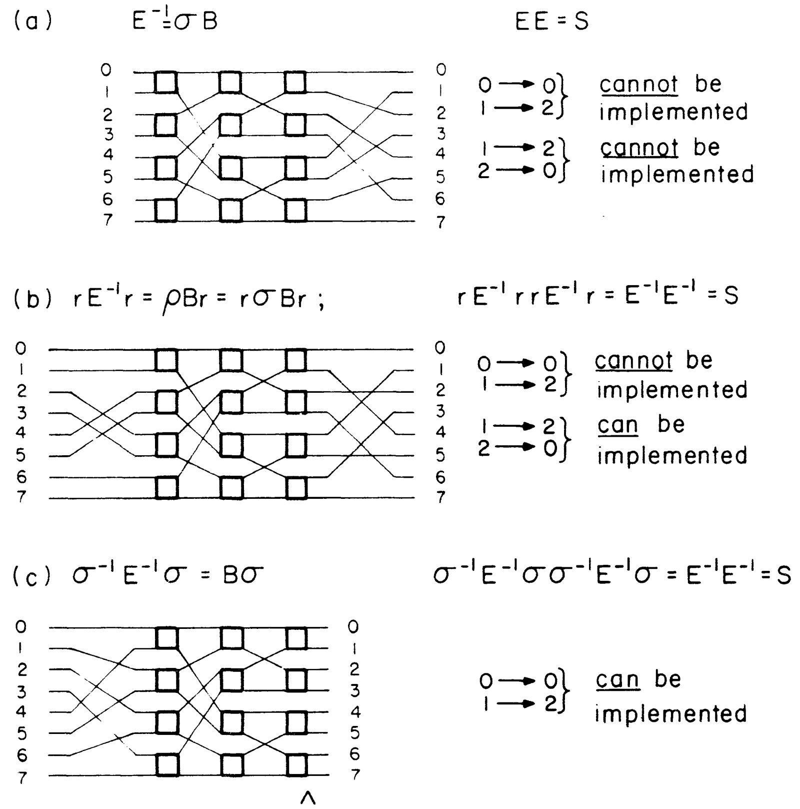
(permutations are given in cycle format)

Figure 1



An example proving that $E \neq E^{-1}$

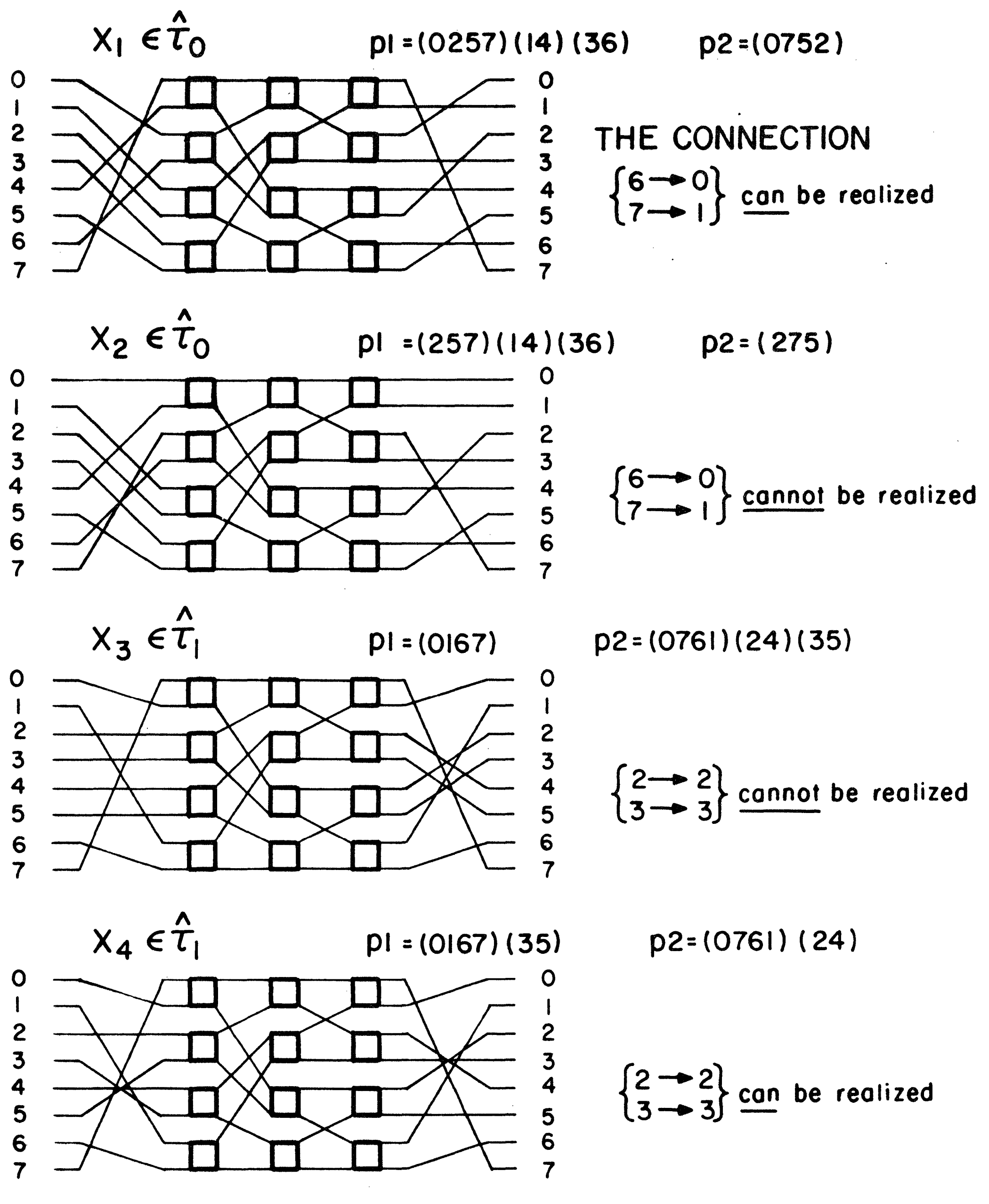
Figure 2



(a) $E^{-1} = \sigma B$
$EE = S$
$0 \to 0$, $1 \to 2$ cannot be implemented
$1 \to 2$, $2 \to 0$ cannot be implemented
(b) $rE^{-1}r = \rho Br = r\sigma Br$;
$rE^{-1}rrE^{-1}r = E^{-1}E^{-1} = S$
$0 \to 0$, $1 \to 2$ cannot be implemented
$1 \to 2$, $2 \to 0$ can be implemented
(c) $\sigma^{-1}E^{-1}\sigma = B\sigma$
$\sigma^{-1}E^{-1}\sigma\sigma^{-1}E^{-1}\sigma = E^{-1}E^{-1} = S$
$0 \to 0$, $1 \to 2$ can be implemented

Examples of $\hat{\beta}_2$ type networks

Figure 3


$X_1 \in \hat{\tau}_0$
p1 = (0257) (14) (36)
p2 = (0752)
THE CONNECTION
$\begin{Bmatrix} 6 \to 0 \\ 7 \to 1 \end{Bmatrix}$ can be realized
$X_2 \in \hat{\tau}_0$
p1 = (257) (14) (36)
p2 = (275)
$\begin{Bmatrix} 6 \to 0 \\ 7 \to 1 \end{Bmatrix}$ cannot be realized
$X_3 \in \hat{\tau}_1$
p1 = (0167)
p2 = (0761) (24) (35)
$\begin{Bmatrix} 2 \to 2 \\ 3 \to 3 \end{Bmatrix}$ cannot be realized
$X_4 \in \hat{\tau}_1$
p1 = (0167) (35)
p2 = (0761) (24)
$\begin{Bmatrix} 2 \to 2 \\ 3 \to 3 \end{Bmatrix}$ can be realized
0
1
2
3
4
5
6
7

Examples of $\hat{\tau}_0$ and $\hat{\tau}_1$ type networks

Figure 4

MODELLING CONTROL STRATEGIES FOR ARTIFICIAL INTELLIGENCE APPLICATIONS

A. Giordana [+] P. Laface [++] L. Saitta [+]

[+] Istituto di Scienza dell'Informazione
Università di Torino
Corso Massimo D'Azeglio 42
10125 - TORINO (Italy)

[++] C.E.N.S.
Istituto di Elettrotecnica
Politecnico di Torino
Corso Duca degli Abruzzi 24
10129 - TORINO (Italy)

Summary

Many problems in the domain of Artificial Intelligence (A.I.) require great computation power and are suitable for parallel processing [1]. This paper presents a modification of Hewitt's Actor System [6-8], oriented to these problems, in particular processing of data from the real world, such as continuous speech or visual images. In these cases various sources of errors affect the input data and also the a-priori knowledge is ambiguous and uncertain.
The described here model takes into account these peculiarities, devoting particular attention to the flow of messages and the scheduling philosophy.

In the last decade many efforts have been made toward the application of parallel processing to this kind of problems;the use of traditional programming techniques, which do not provide distributed and non-deterministic control structures, leads to complex implementations, unsuitable for formal description and rich of ad hoc solutions.
The approach we propose is based on the analysis of the specific characteristics of the said class of problems, in particular:

- Uncertainty and great number of data to be processed.
- Intrinsic concurrent nature of the decision algorithms.
- Complexity of the control structure and therefore need of introducing non-deterministic construct.
- Type of computations to be executed, simple but very frequent.
- Natural structuration of the data base and of the informations utilized by the algorithms, which are suitable to be distributed.

Last point, i.e. data base distribution, is one of the most relevant factor in order to obtain a high degree of parallelism [1,6]. The interest for parallel processing [1,5] is then quite obvious: many concurrent tasks, performed by asynchronous processors, will hopefully speed up the search of a solution. In fact many alternatives may be followed at the same time and results evaluated and compared. An heuristic search strategy selects a subset of possible alternatives, by taking into account information of various kind about the problem [4]. This information can be described by means of Knowledge Sources (KSs), which will help in solving the problem. To this aim, the KSs are hierarchically linked together in such a way that the search for a solution may be performed at different levels of analysis. Each KS can work at a given level, by utilizing the results obtained by those working at lower levels.The KSs cooperate to the emission of hypotheses about solutions,according to the paradigm "Hypothesize and Test", i.e. each KS can emit a partial hypothesis, which will then be verified by the KS itself or by others [1]. This hypothesis emission can be activated whether bottom-up (data driven) or top-down (model driven); bottom-up stimulation occurs when a nucleus of a hypothesis is drawn directly from the data; top-down invocation occurs when a KS calls another one at a lower level to verify a part of the hypothesis; each KS can also predict some part of the input data, when comprehension was not satisfactory. Therefore in such a system we have a continuous flow of information in the two directions. Moreover the relations between the elements contained in a KS (i.e. 'concepts' in a semantic network) are expressed by means of AND/OR graphs. This KS organisation and hypothesis formation process can be favorably implemented by means of non deterministic constructs [8,9]. In an OR node of an AND/OR graph, for example,progress can be evaluated on one basis of the first satisfactory verification without waiting for all the other components.
Various models of computation, based on the idea of communicating processes, have been recently proposed [6,9,10]; in particular Hewitt's Actor System has been developed as a general tool for modelling A.I. control strategies.The model we propose derives from the Actor System and has been designed taking into account the particular class of problems described.Fundamental objects of the Actor System are the Actors, potentially active pieces of knowledge, communicating among themselves by means of messages. In Hewitt's model, messages are also actors, but here we will refer to actor-messages simply as to messages.Messages contain data structures and possibly descriptions of other actors to be created. The receipt of a message by an actor is an Event; this activates the actor itself, which in turn processes the message, updating its local knowledge; moreover it may send new messages and eventually create new actors. An actor activation must always terminate; any other message arriving during this phase, must wait for the end of the current activation. Interference between messages is resolved by a fair arbiter.

As the activation of an actor can depend upon the random sequence of its events the Actor System includes a potential non-determinism (within the actor). This non-determinism can be favorably exploited by the control strategies previously described. Furthermore, the Actor System shows a dynamic and flexible structure, in that actors can be created and then removed from the system, when no longer needed. This last feature is fundamental for us, because it is impossible to have all the possible instantiations of the KSs a priori. On the other hand, other fundamental features for our applications are not explicitely included in Hewitt's model. First of all, in the Actor System the receipt of the messages by an actor is controlled by a fair arbiter, in order to avoid the starvation. In this way it is not possible to control the message flow outside the actors and any scheduling strategy must then be included in the actor itself. On the contrary, in our case processing of the most reliable hypotheses must always be preferred to the other ones, when competing with others for the same resource (e.g. the activation of the same actor). In fact the starvation of a bad hypothesis is not relevant. Thus in the said kind of applications, the direct implementation according to the Actor System, leaves the job of designing all the scheduling strategy supports to the user, complicating the programming task.

Another specific feature of our problem is the presence of two flows of information, i.e. bottom-up and top-down. As the flows may have not the same weight in different situations, it is better to have two autonomous control policies. The fundamental difference between our model and Hewitt's is the policy of the message reception; in particular:

- A set $\mathcal{C}$ of different classes of messages is defined.
- When an actor sends a message (to another), it assigns both a class identifier $c \in \mathcal{C}$ and a priority p.
- The actors receive the messages served by an arbiter which then orders and dispatches them according to a user definible function $f_s(c, p_m(c), Q)$ where c is the class identifier, $p_m(c)$ is the maximum priority of the waiting messages of class c, and Q is a parameter settable by the actor. In this way it is possible to specify f_s as function depending on Q in order to dynamically assign a preference to the messages of a particular class.

If a unique class of messages is defined and the same priority assigned to all the messages, the original Actor System fair scheduling is obtained. Furthermore, our model can be described in terms of Hewitt's model. In fact, the so defined actor can be considered as a compound of two A_{op} and A_s actors, where A_s is a Guardian [8], which fairly receives the messages and then dispatches them to A_{op} according to the described function f_s.

We will now describe the application of the computation model to the control of the semantic knowledge source for a Speech Understanding System [12]. The semantic network [11] consists of a graph, whose nodes represent concepts and whose arcs represent compatibility conditions among concepts. The graph is partitioned into subgraphs (Islands), which are, in turn, sets of correlated concepts with direct access to the input data. Fig. 1 shows the levelled hierarchy of the nodes in the graph: at each level the relationships among the nodes at the lower level are expressed by means of AND/OR relations [3]. Fig. 1 shows also the implementation scheme of the KS in terms of actors. Each α node (a memory actor which knows the AND/OR relations among a set of nodes at a lower level) is associated to a Controller actor C_α, that contains α in its acquaintances. (The acquaintances of an actor A are constituted by the set of all other actors which are known by A). The motivation for introducing C_α is the following: the same node may be called during a top-down process (by means of a message belonging to the class Messages) or during a bottom-up process (by means of a message belonging to the class Stimuli). An actor A can communicate with actor B only if it creates or is acquainted to B. In this case the two previously mentioned strategies would proceed independently, without intercommunication and would duplicate all actors called by both. On the contrary, to realize an effective strategy of cooperation and exchange of results, for each α node, you introduce the C_α controller actor, which is globally known and predefined at the time of system initialization. C_α receives the calls to α and coordinates the creation of the new actors, needed for developing the two strategies. We notice that, because of the great number of requests which, in general, C_α receives, the controller may make a wide use of the facilities introduced in the model: differentiation between the bottom-up and top-down processes, flexibility of the message receipt scheduling, consent to the starvation of some request. In fact this is how the controller hinders the proceeding of the bad hypotheses, thus limiting the number of computations to be executed.

Finally, an Initializer actor manages the access to the input data.

The actors described in Fig. 1 represent the a priori knowledge of the system. But, when actual data are processed, other actors will be dynamically (Fig 2) created. In fact, when a controller C_α receives a request for verifying a hypothesis, it develops, by creating a Producer actor P_α, the AND/OR graph contained in its acquaintances and produces the necessary AND/OR actors. The latter, in turn, will send (via IF actors) requests to the controllers of lower nodes and so on. The answers will return to C_α via the same path. When a C_α controller is stimulated again, it will not repeat the verification process, but send the previously found results. The experimental system based on this model is now being implemented on a DEC-10 using SIMULA 67.

References

[1] - R. Fennell, V. Lesser:"Parallelism in AI Problem Solving - A case study of Hearsay II", IEEE Trans. C-26, 98-111 (1977)

[2] - L. Kanal:"Problem Solving Models and Search Strategies for Pattern Recognition", IEEE Trans. PAMI-1, 193-201 (1979)

[3] - N. Nilsson:"Problem Solving Methods in AI", McGraw Hill, New York (1971)

[4] - P. Rovner, B. Nash-Webber, W. Woods:"Control Concepts in a Speech Understanding System", IEEE Trans. ASSP-23, 136-140 (1975)

[5] - V. Lesser, L. Erman:"An experiment in Distributed Interpretation", CMU-CS-79-120 (1979)

[6] - C. Hewitt: "Viewing Control Structures as Pattern of Passing Messages", Artificial Intelligence, 8, 323-364 (1977)

[7] - R. Atkinson, C. Hewitt:"Synchronization in Actor System", Proc. 4-th SIGPLAN-SIGAT Symposium, Los Angeles, (1977)

[8] - C. Hewitt, G. Attardi, H. Lieberman:"Specifying and Proving Properties of Guardians for Distributed Systems", MIT Memo 505 (1979)

[9] - C. Hoare:"Communicating sequential Processes" Comm. ACM 21, 666-677 (1978)

[10] - P. Brinch Hansen:"Distributed Processes- A concurrent Programming Concept", Comm. ACM 21, 934-941 (1978)

[11] - M. Coppo, L. Saitta:"Semantic Support for a Speech Understanding System", Proc. ICCS (Washington, 1976), pp. 520-524

[12] - R. De Mori, S. Rivoira, A. Serra:"A Speech Understadding System with Learning Capability", Proc. IJCAI (Tbilisi, 1975), pp. 468-475

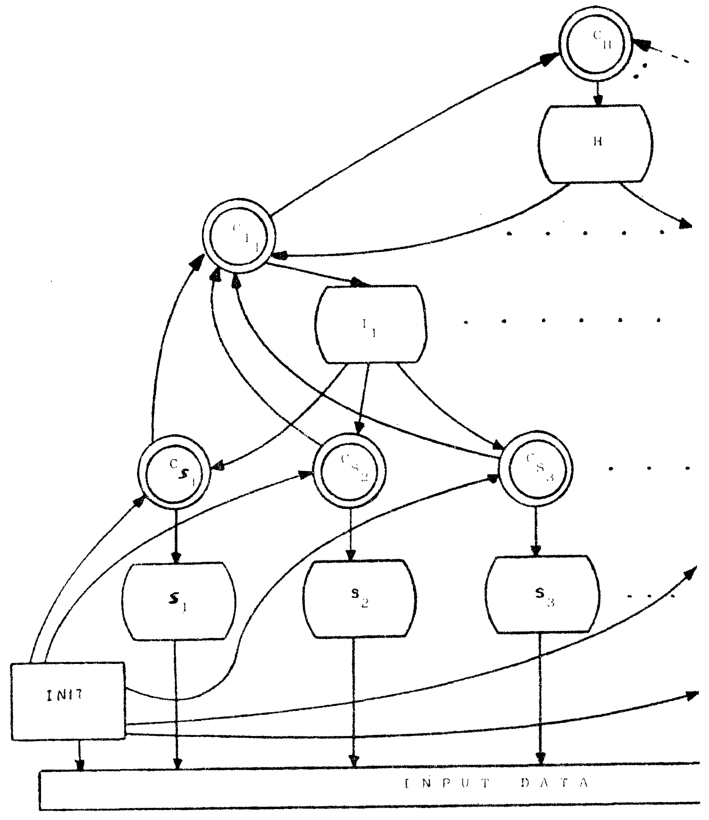


Fig.1 - Hierarchical organization of KSs.

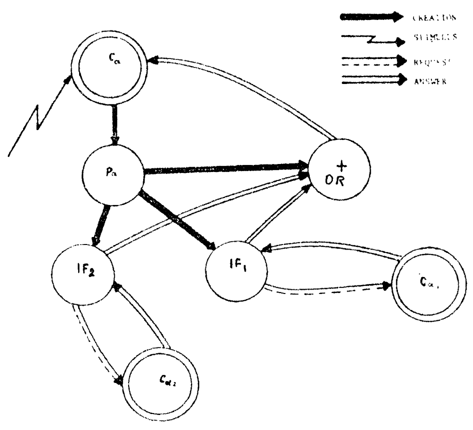


Fig.2 - Example of the expansion of an OR node.