MIT/LCS/TR-265

A DATAFLOW ARCHITECTURE WITH IMPROVED ASYMPTOTIC PERFORMANCE

Robert E. Thomas

A Dataflow Architecture with Improved Asymptotic Performance*

April 1, 1981

Robert E. Thomas

*This work was done while the author was at the University of California, Irvine.

CONTENTS

ACKNOWLEDGMENTS

I am indebted to Professors Arvind and Kim Gostelow, my advisors, who provided the support, encouragement, and enthusiastic interest in research which made it easy to select the topic of this dissertation. I would also like to thank the members of my committee for suggesting improvements to the manuscript.

I am deeply grateful for the financial support received through the National Science Foundation and the Department of Information and Computer Science, under whom I worked as a Teaching Assistant and later as a Research Assistant.

ABSTRACT OF THE DISSERTATION

A Dataflow Architecture with Improved Asymptotic Performance

by

Robert Eugene Thomas

Doctor of Philosophy in Computer Science

University of California, Irvine, 1981

Professor Kim P. Gostelow, Chair

Large scale integration presents a unique opportunity to design a computer comprising large numbers of small, inexpensive processors. This paper presents a design for such a machine based on the asynchronous and functional semantics of dataflow. Processors within the machine are interconnected by a packet-switched binary n-cube although a limited number of other networks may be substituted with predictable asymptotic effects on performance. Improved performance of the proposed machine over a previously reported dataflow architecture is predicted in terms of the computational time complexity of several example programs: matrix multiply, quicksort, and iterative solutions to partial differential equations. Although the example programs are numerical in nature, the machine is intended for general-purpose computation since programs are written in the high level dataflow language Id without knowledge of the number of processors or interconnections. New storage management and data communication methods are also presented which are necessary to obtain the improved performance. Experimental results from a simulated machine incorporating some of these methods are given to corroborate analytic results.

1.Ø INTRODUCTION

Large scale integration presents an opportunity to design a computer comprising hundreds or thousands of small, inexpensive processors. This opportunity is attractive for several reasons. First, signal propagation delay will eventually limit the performance of conventional sequential computers. Thus multiple execution units of some sort (e.g., arithmetic/logic units, processors) will eventually be necessary to increase performance further. Second, the current trend of rising software costs relative to hardware costs warrants, in many cases, trading inexpensive hardware for ease of software production. One of our approaches for realizing this tradeoff is to transfer the responsibility for processor and memory management from the programmer to the machine. We view automatic resource management as a potential source of additional parallelism which, if suitable exploited, would mitigate performance losses normally associated with dynamic resource management. A third reason multiprocessor computers are attractive is that redundant processors and communication links may be used to continue computation in the face of certain hardware faults. While error detection and control are beyond the scope of this paper, we believe the principles of dataflow underlying the architecture described here provide new opportunities for supporting high performance "fail-soft" computing.

1.1 Principles of Dataflow

Dataflow is a model of computation based on asynchrony and functionality. Asynchrony means a dataflow operation (e.g., a machine instruction) may begin execution any time after receiving its input operands. Functionality means every dataflow operation consumes a set of input values and creates a set of output values without side-effects. Asynchrony is the basis of concurrency in dataflow while functionality ensures concurrent operations do not interfere and therefore need not be artificially sequenced. Detailed descriptions of various dataflow computational models and their advantages have been presented elsewhere [3,12,13,22,40].

1.2 Dataflow Architectures and Complexity Analysis

Many dataflow architectures have been proposed [11,12,14,15,22,23,33,42] and a few prototypes have been constructed [11,12,23]. Evaluation of these architectures is an important task, but so far this evaluation has been more art than science. In an effort to improve the situation, time complexity analysis was used to design and evaluate the architecture presented here. Although the O-notation used in the complexity analysis is admittedly a rough tool (large constants may be hidden), such analysis can quickly determine the presence of bottlenecks in large computations. Some readers may also question the assumption used in the complexity analyses that an unbounded number of processors are available. Of course, any realizable machine is limited to a finite number of processors, communication lines, memory cells, etc. However, the necessarily limited resources of von Neumann machines has not lessened the value of complexity analysis for single processors. Therefore, in the same way that unbounded memory is assumed in conventional complexity analysis, unbounded memory and processors will be assumed in the parallel complexity analyses presented here.

An important practical issue in the application of dataflow and complexity results to real systems is hardware and software cost. All too often studies of parallel computing models do not intend that the model be implemented (i.e., it is a theoretical model only) or the analysis gives little or no aid to the programming of the proposed machine. At least one solution to this problem has been achieved and tested by simulation in a dataflow architecture [22]. There it was shown that general programs can be written in a high level language, Irvine dataflow (Id) [3]. Only one compilation is then required for multiple executions with various sized data. An important result is that parallel programs can be written without knowledge of the number or the interconnection of processors. The methods used to obtain these results are extended in this paper to a new dataflow architecture which has improved asymptotic performance over that described in [22].

Section 2 of this paper describes a simple parallel computer model used for the complexity analysis of some common algorithms. Section 3 describes how the theoretical model of Section 2 can be implemented in a dataflow environment. Section 4 presents experimental evidence (derived from executing real programs on a simulated dataflow machine) which lends support to selected results from Section 3.

2.0 AN ANALYTIC, PARALLEL COMPUTER MODEL

Complexity analysis requires an explicit model of the computing device concerned. Examples of models commonly used in complexity analyses are the Turing machine and the "random access machine" [1]. In this section, a simplified model of a parallel architecture is described for the purpose of complexity analysis. Implementation of the model will be discussed in Section 3.

Our parallel computer model comprises an unbounded number of processing elements (PEs) interconnected by a communication network. Intuitively, each PE may be considered a conventional processor directly connected to a private, unbounded memory. Network communication is assumed to operate on a "store and forward" packet basis. For purposes of analysis (as opposed to implementation), all PEs and communication links are synchronized by a central clock. This simplifies analysis by reducing the need for methods based on probabilities. Although the results thereby achieved apply only indirectly to the implementation discussed in Section 3, it is considered prudent to use this kind of analysis before more detailed analyses are conducted.

The communication network has considerable impact on the cost of implementing the computing device. For example the crossbar network, e.g. [45], has cost $O(N^2)$ where N is the number of nodes, and thus the communication network clearly dominates cost for large N. Although a number of networks costing less than $O(N^2)$ have been proposed [7,8,18,26,31,32,34,36,37,41,43], relatively little comparative information has been published about them. A good start in this direction is the work of Siegel [34], and Wu and Feng [44]. Wu and Feng discussed the equivalence of several networks while Siegel compared a small number of quite different networks in terms of the computational time complexity of simulating one network using another. Such comparisons are significant because they allow complexity results derived from one network to be applied to other networks.[1] This is one reason why one of the networks studied by Siegel, the binary n-cube, has been chosen as the network of our parallel computer model.

2.1 The Binary n-cube

A binary n-cube is an interconnection of $N = 2^n$ PEs placed at the corners of an n-dimensional cube. Each <u>edge</u> or <u>link</u> of the cube has two PEs; each PE has n

----

[1]Careful interpretation of Siegel's results is necessary since these results were developed for single instruction stream-multiple data stream (SIMD) computers [16] whereas the model described here is not a SIMD because each PE is assumed to have its own instruction stream. An example use of Siegel's results appears in Section 5: Conclusions.

bi-directional one-message-at-a-time (i.e., half-duplex) links connecting it to n other PEs. Examples of n-cubes are given in Figure 2.1. This paper assumes that at any given instant each PE can transmit or receive (but not both) on any one of the n links connected to it, although more concurrent implementations are also possible. The interconnections of the n-cube can be expressed formally using $p_{n-1} \ldots p_\emptyset$ to denote the binary address of an arbitrary PE and $\overline{p}_i$ as the complement of p_i. The ith function defining the n-cube interconnections is given by

$$\mathrm{cube}_i(p_{n-1} \ldots p_{i+1} p_i p_{i-1} \ldots p_\emptyset) = p_{n-1} \ldots p_{i+1} \overline{p}_i p_{i-1} \ldots p_\emptyset \qquad \emptyset \leq i < n$$

In the sequel, the notation for a particular communication link will be abbreviated from $\mathrm{cube}_i(x)$ to cube_i when the address x is obvious from context.

2.2 Binary n-cube Properties

The Hamming distance between two binary numbers (PE addresses) is the number of bit positions which differ in the two numbers. Let $\mathrm{bit}_j(z)$ denote the jth bit in address z. The following routing algorithm may be used to direct a message from PE x to PE y. Select any i such that $\emptyset \leq i < n$ and $\mathrm{bit}_i(x) \neq \mathrm{bit}_i(y)$. If no such i exists then the message has arrived; otherwise, transmit the message using function $cube_i(x)$ and repeat with the new address x. This has the effect of reducing the Hamming distance by one at each step of transmission, and since the largest Hamming distance is $n = \log N$, at most log N steps[2] are needed to transmit a message. This routing algorithm also implies that if two PEs are separated by Hamming distance m then m! distinct paths exist between the two PEs. Another interesting capability of the n-cube is the *most distant N message transfer* described by Sullivan and Bashkow [37]. In this transfer, each PE (concurrently with all other PEs) sends a single message to that PE at the greatest Hamming distance from it. The algorithm[3] is as follows:

Algorithm 2.1 [37].

```
For i from Ø to n-1 do
    Using links cube_i, all PEs transmit/receive
    each message with destination address which
    differs from the message's current address in
    the ith bit position;
```

---

[2]All logarithms will be taken to the base two.

[3]The notation "*For* i ... *do*" implies that every PE uses the same value *of* i at the *same* time in the order given.

To see how this works, note each PE initially contains one message at Hamming distance n from its destination. Since N/2 PEs are directly connected in an n-cube to the other N/2 PEs by links $cube_i$ for each i (i.e., there is a distinct partitioning for each i), N/2 messages can be exchanged in two time steps for each iteration thus bringing all messages one step closer to their destinations. Therefore, 2 log N steps[4] are required to complete the transfer. Given the assumption that at any given step a PE may service only one of the n links to which it is connected[5] this algorithm is optimal since:

1. At each step the Hamming distance of all messages transmitted is decreased by the maximum possible, i.e., by one;

2. Every step uses the maximum possible concurrency, i.e., N/2 transmissions.

The following four capabilities of the n-cube will be used extensively in the sequel. The first is the N-way

----

[4]In [37] only log N steps were required because full-duplex links were assumed.

[5]This assumption precludes "pipelining" of the transfer algorithm as discussed in [37]. If each PE could concurrently transmit/receive on all links to which it is connected, then another (new) message could be started immediately after the first message has departed from its source PE since a given link is used only once in the execution of the algorithm. This would allow each PE to send m distinct messages to the PE most distant from it in (m-1)+log N steps. However, with the assumptions used in this paper, 2m log N steps are required for this transfer since the algorithm must be repeated m times.

broadcast which distributes a single message from one PE to all other PEs in log N steps [37]. Assume the message to be broadcast is transmitted to the original broadcasting PE on a hypothetical link cube_{-1}. The algorithm is:

> Each PE that receives a broadcast message on link cube_i retransmits the message using (in order) links cube_j for $i<j<n$.

The first transmission (for $j=0$) from the original broadcasting PE using cube_0 can be thought of as splitting the original n-cube into two disjoint, identical cubes of size N/2. Two PEs (the original and the receiver on link cube_0) now have the message and each becomes the source to broadcast the message to the sub-cube in which that PE resides. This process is repeated until the resulting sub-cubes contain only one PE which terminates the broadcast.

The second important capability of the n-cube is the N(N-1) transfer where each PE transmits N-1 distinct messages, one to each of the other N-1 PEs. With this transfer, N PEs each send (N-1) messages for a total delivery of N(N-1) messages in N log N steps. (Note that the N(N-1) transfer is easily adapted to perform matrix transpose assuming each PE initially contains exactly one row of the matrix.) The algorithm for the N(N-1) transfer is the same as the one used in the most distant transfer, i.e., Algorithm 2.1.

<u>Theorem</u>: The N(N-1) transfer can be done in N log N steps.

<u>Proof</u> <u>by</u> <u>Induction</u>: The basis is trivially true for N = 2. <u>Inductive</u> <u>step</u>: Assume the N(N-1) transfer requires N log N steps for an n-cube of size N. Let the address of an arbitrary PE of an (n+1)-cube of size 2N be $p_n \ldots p_\emptyset$. Each PE of the (n+1)-cube starts with 2N-1 messages; N of these messages will have destinations d with $\text{bit}_\emptyset(d) = \bar{p}_\emptyset$ and N-1 will have destinations with $\text{bit}_\emptyset(d) = p_\emptyset$ since no PE sends a message to itself. Using Algorithm 2.1, for i=Ø each PE thus transmits N messages and receives N messages using a total of 2N steps. Exactly one of the messages received by each PE must be addressed to itself so each PE now contains 2N-2 = 2(N-1) undelivered messages which are addressed exactly like the messages of two N(N-1) transfers within each of the two sub-cubes defined by the set of addresses $p_n \ldots p_1$. Since links cube_i will not be used again, each of these two sub-cubes can act independently. By the inductive assumption the two N(N-1) transfers within each sub-cube require 2N log N steps. The total is 2N + 2N log N = 2N(log N + 1) = 2N log 2N steps. []

Again, this is optimal under the given assumptions for the same reasons as were given in the most distant N message transfer discussion.

The third important capability of the n-cube is the

N-1 linear transfer where a single PE sends a distinct message to each of the other N-1 PEs in O(N) steps. Although an algorithm seems to exist[6] for doing this transfer in exactly N-1 steps, the proof is nontrivial and its description is not needed for the purpose of this paper. The O(N) algorithm is simply transmit messages (in any order) at every other time step. This algorithm requires 2(N-1) steps for transmission from the source PE plus at most log N steps for the last message to arrive since no conflicts are possible. Thus O(2(N-1)+log N) = O(N) steps are required.

The fourth important capability of an n-cube is the ease and flexibility with which partitions may be defined. Some of these partitions are given in the following definitions. A $k_{n,m}$-partition is a set of m, $\emptyset \leq m \leq n$, distinct integers j, $\emptyset \leq j < n$, specifying the partitioning of an n-cube into k disjoint m-cubes. Each of these m integers represents a distinct bit position in the n-cube PE address $p_{n-1} \ldots p_{\emptyset}$.

Theorem 2.1. Let $k=2^{n-m}$. Then there are n!/(m!(n-m)!) distinct $k_{n,m}$-partitions of an n-cube. (See Figure 2.2a for example.)

----

[6]Send messages in order of decreasing destination Hamming distance and select links so that no one link is used twice in succession.

Proof: Consider for the moment that the n-m bits not in the partition are fixed to some arbitrary value. Then the m bit positions in the partition define a set of 2^m distinct PE addresses which can be re-labeled to integers j, $\emptyset \le j < 2^m$, by ignoring the other n-m bit positions. These re-labeled PEs and their connections satisfy the cube interconnection functions and thus define an m-cube. There are $k=2^{n-m}$ such m-cubes since the n-m fixed bits may assume 2^{n-m} different values. All m-cubes are disjoint because their original addresses are distinct and because an m-cube link must connect two PEs within the same m-cube. Finally, there are n!/(m!(n-m)!) distinct combinations of n bit positions taken m at a time each of which defines a distinct partition of an n-cube. []

The complement of a $k_{n,m}$-partition is the set $\{i \mid \emptyset \le i < n\}$ - ($k_{n,m}$-partition), i.e., all bit positions not in the $k_{n,m}$-partition.

Corollary 2.1a. Each (n-m)-cube specified by the complement of a $k_{n,m}$-partition shares exactly one PE with each m-cube specified by the $k_{n,m}$-partition (Figure 2.2b).

Proof: Follows immediately from the proof of Theorem 2.1 by reversing the bit positions which are fixed with those that are variable. []

2.3 Example Complexity Analyses

This subsection presents the complexity analysis of three numerical algorithms. The methods and results are intended to demonstrate the n-cube's capability for concurrent communication and to serve as a basis for generalizing the methods to non-numerical algorithms. Besides the already mentioned assumptions, the analyses assume the "uniform cost criterion" [1]. This means that primitive machine operations such as +, *, etc. are assumed to take constant time regardless of the size of the operands.

2.3.1 Related Work and Data Structure Assumptions -

The application of parallel processors to numerical problems has been studied for some time. For example, Squire and Palais give a program (without analysis) for matrix inversion on a proposed parallel machine incorporating a circuit-switched binary n-cube [35]. Many studies have been done for Illiac IV-like interconnections e.g., [19,25,30,39]. The advent of VLSI has further encouraged work in the area of "computation grids" [28]. The usual assumption made in these studies is that a PE works with a constant, usually small, number of data elements. One reason for this is the speed of the results obtained; for example, an O(log N) algorithm has been shown for NxN matrix transpose on the perfect shuffle network [36].

The difference between the current approach and the others cited is that here the PEs work with complete rows of data instead of single elements. Although this approach may result in an increase in time complexity (e.g., O(N) to O(N log N) for matrix multiply), aggregates of data larger than a single element are required for the implementation proposed in Section 3 which is intended to avoid one of the problems of "array computers": the exacting data layout and communication requirements that make such computers difficult to program. Furthermore, since the location of individual data elements is usually implicitly buried in the user's program, continuing operation with the loss of just one communication link or processing element becomes a difficult problem. An alternative approach using aggregates of data combined with dataflow allows the physical location of data to be divorced from the user's program as is shown in Section 3. The scheme proposed there allows the machine to function as long as at least one PE remains operational and sufficient memory is available (although time complexity may, of course, suffer).

2.3.2 NxN Matrix Multiply -

In this subsection, it is shown that two NxN matrices, A and B, can be multiplied in O(N log N) time ($N=2^n$ without loss of generality) using a 2n-cube (N^2 PEs) when the location of the input is favorably distributed. The average time required over all possible input distributions is unknown at this time but is conjectured to be O(N log N) assuming no PE begins or ends with more than a constant number of input or result rows.

2.3.2.1 One Possible Input Row Distribution -

Let $p_{2n-1}\cdots p_n p_{n-1}\cdots p_\emptyset$ denote a PE address in the 2n-cube. The N rows of matrix A are distributed over N distinct PEs such that the address of those PEs satisfy $p_{2n-1}\cdots p_n = p_{n-1}\cdots p_\emptyset$ (Figure 2.3a). Each of these PEs is an element of exactly one of the n-cubes (the front and back faces in Figure 2.3a) specified by the $N_{2n,n}$-partition, $\{i \mid \emptyset \leq i < n\}$. The N rows of matrix B are evenly distributed over the PEs of the n-cube defined by $x_{2n-1}\cdots x_n\emptyset\cdots\emptyset$ where x indicates a bit position which varies among PEs in the same n-cube (Figure 2.3c). This n-cube is one of the n-cubes specified by the partition $\{i \mid n \leq i < 2n\}$ which is, of course, the complement of the $N_{2n,n}$-partition above. By Corollary 2.1a, the n-cube in which B is distributed shares exactly one PE with each n-cube containing exactly one row of A; this is the "favorable" input distribution requirement.

2.3.2.2 The Matrix Multiplication Algorithm -

1. Transpose B to form B^t over n-cube $x_{2n-1}...x_n\emptyset...\emptyset$ using a N(N-1) transfer in N log N steps (Figure 2.3c).

2. N-way broadcast each row of B^t residing in PE $p_{2n-1}...p_n\emptyset...\emptyset$ to all PEs in the n-cube $p_{2n-1}...p_nx_{n-1}...x_\emptyset$ in N log N steps (Figure 2.3d). This can be done for all rows at the same time, since the n-cubes are distinct.

3. N-way broadcast each row of A residing in PE $p_{2n-1}...p_np_{n-1}...p_\emptyset$ to all PEs in the n-cube $x_{2n-1}...x_np_{n-1}...p_\emptyset$ in N log N steps (Figure 2.3b). This step can also be done for all rows at the same time.

4. Each PE now contains a row of A and a column of B and can form the inner product in O(N) steps (Figure 2.3e).

5. Using a "reverse" N-1 linear transfer, the N elements of each result row can be brought together within the same PEs which initially held a row of A in O(N) steps. This step can also be

done for all rows at the same time.

The total is (3N log N) + O(N) = O(N log N).

2.3.3 Quicksort of N Distinct Elements -

The analysis of quicksort presented here is simplified because there is no need to repeat the work of others. It is well known the average time complexity of quicksort on a single processor is O(N log N) while the worst case complexity is $O(N^2)$ [24]. Let the N (=2^n without loss of generality) distinct numbers to be sorted reside in an arbitrary PE of an n-cube. Assume this vector was transmitted to that PE on a hypothetical link $cube_{-1}$. The quicksort algorithm is:

```
For each PE receiving a vector to be sorted on link $cube_j$ do
    1. Let A be the input vector received by a
       particular PE on link $cube_j$;
    2. For i from j+1 to n-1 do
          a. Select the median of A which can be done
          in O(length of A) [1, p.97];
          b. Construct (within the same PE) a new
          vector A' using elements from A which are
          less than or equal to the median; transmit
          all other elements of A using link $cube_i$
          as a vector to be independently sorted;
          c. Let A = A';
    3. For i from n-1 downto j+1 do
          Concatenate A with the sorted vector
          received on link $cube_i$ to form a new A ;
    4. Transmit the resulting sorted vector A on
       link $cube_j$.
```

To see how this works, imagine the n-cube is split into two disjoint (n-1)-cubes. The source PE splits the input vector into two equal parts and transmits one of the parts to the other (n-1)-cube where it is independently sorted. Each N/2 element vector is then split again and one-half is sent to an (n-2)-cube to be independently sorted, and so on, until the length of each vector is one. No PE (or link) does more than O(N) work in this splitting phase. The vectors are then concatenated by reversing the above process starting with vectors of length one and ending with a vector of length N. Again no PE does more than O(N) work. Therefore, n-cube quicksort requires O(N) time.

2.3.4 Binary n-cubes and Mesh Connected Computers -

Before analyzing a portion of a partial differential equation problem, we will briefly explore the relationship between mesh connected computers (MCC) and n-cubes. A MCC is an interconnection of $N = 2^n$ identical PEs. The PEs are arranged in a q-dimensional $s_{q-1}X...Xs_0$ array, where each s_i is a power of two and $s_{q-1}*...*s_0 = N$. A PE address is expressed in standard coordinate indices as $PE(i_{q-1}, ..., i_0)$, $0 \le i_k < s_k$, $0 \le k < q$. Each $PE(i_{q-1}, ..., i_k, ..., i_0)$ is connected to its nearest two neighbors in each of q dimensions $PE(i_{q-1}, ..., i_k \pm 1, ..., i_0)$, $0 \le k < q$, provided they exist. PEs at the boundaries of the mesh have fewer than 2q connections unless the MCC is specified to have "wraparound connections". Each PE in a MCC with orthogonal wraparound (OW) is connected to exactly 2q neighboring $PEs(i_{q-1}, ..., ((i_k \pm 1) \underline{mod} s_k), ..., i_0)$, $0 \le k < q$. These definitions are due to Nassimi and Sahni who consider optimal routing on MCCs [30]; fast sorting algorithms on MCCs are discussed in [29,39]. Illiac IV is a two-dimensional MCC with slightly different wraparound connections called "propogating wraparound" [30].

An interconnection network can be represented by a directed graph denoted by {V, E} where V is a set of vertices (PEs) and E is a relation which is a subset of VxV representing edges (connections) between vertices. Since only bi-directional links are considered here, E will automatically be a symmetric relation, i.e., if $(v_1, v_2) \in E$ then $(v_2, v_1) \in E$. A network $\{V_1, E_1\}$ is a subnet of network $\{V_1, E_2\}$ if $E_1 \subseteq E_2$ (note the set of vertices is the same). Network $\{V_1, E_1\}$ is topologically equivalent to network $\{V_2, E_2\}$ if there exists at least one function f called a re-labeling function satisfying:

a) f is one-to-one and onto from domain V_1 to range V_2;

b) $\forall\ ((a, b) \in E_1)\ \ (f{:}a,\ f{:}b) \in E_2$;

c) $\forall\ ((x, y) \in E_2)\ \ (f^{-1}{:}x,\ f^{-1}{:}y) \in E_1$.

Theorem 2.2. An OW MCC of size N is a subnet of an n-cube of size N.

Proof: Consider the concatenation of the binary representations of the MCC coordinates $i_{q-1} \ldots i_\emptyset$ to be a binary PE address. Let $G_j(x)$ represent a j-bit Gray code mapping the integers x, $\emptyset \leq x < 2^j$, into the corresponding Gray code value. The re-labeling function f maps from MCC address $i_{q-1} \ldots i_\emptyset$ to n-cube address $G_{\log s_{q-1}}(i_{q-1}) \ldots G_{\log s_\emptyset}(i_\emptyset)$. f is clearly one-to-one and onto by a simple combinatoric argument. Next consider an arbitrary element of the MCC interconnection relation $((i_{q-1}, \ldots, i_k, \ldots, i_\emptyset),$ $(i_{q-1}, \ldots, i_k \pm 1, \ldots, i_\emptyset))$ which is mapped by f to $((G_{\log s_{q-1}}(i_{q-1}), \ldots, G_{\log s_k}(i_k), \ldots, G_{\log s_\emptyset}(i_\emptyset)),$ $(G_{\log s_{q-1}}(i_{q-1}), \ldots, G_{\log s_k}(i_k \pm 1), \ldots, G_{\log s_\emptyset}(i_\emptyset)))$.

This link is an element of the n-cube interconnection relation since, by definition of a Gray code, consecutive Gray code values vary in exactly one bit position which conforms to the definition of the n-cube interconnection functions. Thus under the re-labeling function f, an MCC is a subnet of an n-cube. []

Pease proved a similar result for a more general network, the "indirect binary n-cube" [31].

Corollary 2.2a. An OW MCC of size N such that $s_k = 4$, $\emptyset \leq k < q$, is topologically equivalent to an n-cube of size N (e.g., Figure 2.4).[7]

Proof: The number of OW MCC connections per PE is 2q and the number of n-cube connections per PE (log N) is the same when $s_k = 4$, $\emptyset \leq k < q$. By Theorem 2.2 (the OW MCC is a subnet of the n-cube) the two networks must be topologically equivalent. []

A re-labeling function is said to configure one network into another network. When the domain of the re-labeling function is physically part of a larger or more connected network, PEs and connections not included in the domain network may be ignored rather than physically deleted. A

----

[7]When all $s_k=2$, the OW MCC and n-cube are also equivalent since an n-cube is defined as a (q=n)-dimensional cube of side two.

single network may thereby be configured into a number of other networks by distinct re-labeling functions acting on possibly different subnets of the original network.

Corollary 2.2b. Let an n-cube be configured as a $s_{q-1}X...Xs_{\emptyset}$ MCC. Then this n-cube can be split into two disjoint (n-1)-cubes by a hyperplane bisecting the MCC in any one of the MCC's dimensions of size s_k provided $s_k \geq 2$ (e.g., see Figure 2.5).

Proof: Re-label the PEs from MCC addresses to n-cube addresses as in Theorem 2.2 but with special consideration given to the Gray code mapping coordinate index i_k where k is the index of the bisected dimension. Let LMB represent the left most bit position in the binary representation of i_k. Select the Gray code such that $bit_{LMB}(i_k) = \emptyset$ for $\emptyset \leq i_k < s_k/2$, and $bit_{LMB}(i_k) = 1$ for $s_k/2 \leq i_k < s_k$ (e.g., a standard reflected Gray code). Then the $2_{n,n-1}$-partition = $\{j \mid \emptyset \leq j < n\}$ - {the position of LMB in an overall n-cube address} specifies the desired partitioning of the n-cube into two disjoint (n-1)-cubes. []

Corollary 2.2b may be applied recursively to partition an n-cube configured as a MCC into many different m-cubes of various sizes (each a power of two). These m-cubes may be MCC hypersolids or hyperplanes ranging in size from half of the MCC to parts of individual rows of the MCC. In the sequel, MCC hyperplanes will be denoted by listing the fixed coordinates within parentheses e.g., $(i_1=5,\ i_\emptyset=\emptyset)$.

2.3.5 Partial Differential Equation Complexity -

The "nearest neighbor" connections of MCCs fulfill a major part of the communication requirements for iterative solutions to partial differential equations (PDE). Although convergence, stability, etc. complicate the issue [17], for simplicity the analysis here considers only nearest neighbor communication requirements. For this purpose, an n-cube of appropriate size is configured as an OW MCC in accordance with Theorem 2.2. Unlike most studies of PDE solutions which assign one data element to each PE of a MCC, here a row of data elements is assigned to each PE; again the motivation is to meet the requirements of the implementation to be presented in Section 3.

Consider a q-dimensional $s_{q-1}X\ldots Xs_\emptyset$ PDE problem where each s_k includes boundary data at indices $\emptyset$ and s_k-1. This problem may be mapped onto a q-dimensional MCC of size $s_{q-1}X\ldots Xs_1X\ O(s_\emptyset)$ such that each data row $(i_{q-1},\ldots,i_1)$ of size $s_\emptyset$ is placed in $PE(i_{q-1},\ldots,i_1)$ in MCC hyperplane $(i_\emptyset=\emptyset)$, e.g. see Figure 2.6. The computation then progresses along the dimension of size $O(s_\emptyset)$ such that MCC hyperplane $(i_\emptyset=k)$ contains the state of the problem at iteration k. An informal description of the data movement is as follows, where z is the actual value of $O(s_\emptyset)$ and a PE is called active when the necessary data are available:

```
For k from 1 until (convergence is achieved) do
  For j from 1 to $s_\emptyset-2$ do
    Each active PE sends the jth element of its row to
    each of its nearest 2(q-1) neighbors in MCC
    hyperplane ($i_\emptyset$=k) provided they exist. When the
    necessary data is present, each active PE computes
    an element for the next k-loop iteration, and
    passes this result along the dimension of size z to
    its neighbor in MCC hyperplane ($i_\emptyset$=(k+1) mod z),
    i.e., orthogonal wraparound connections are used
    when k=z-1;
```

For each element produced in a k-loop iteration, an interior PE will thus send 2(q-1) elements to neighbors, receive 2(q-1) elements, and send one result element to the next hyperplane. The PE computation for each such result is assumed to take at most O(q) time and thus the time required to produce the first elements for iteration k+1 is O(q). The PEs in hyperplane ($i_\emptyset$=k+1) which receive this data may then begin to exchange data and compute results as soon as the first few elements of each input data row arrive. The final computation for each row of size $s_\emptyset$ may thus be thought of as being carried out by a O($s_\emptyset$)-stage circular "pipeline". The time required is O($qs_\emptyset$) to initially distribute computation in the pipeline. All stages may then compute concurrently to finish in O(qT) time where T is the number of k-loop iterations required for convergence. The total is $O(q(s_{\emptyset} + T))$ for the overall PDE computation.[8]

---

[8] The time required for the PDE problem using a comparable number of PEs where each PE is assigned only a constant number of data elements (as in most array computer algorithms) is $O(qT)$.

3.Ø ONE IMPLEMENTATION OF THE PARALLEL COMPUTER MODEL

In the previous section, several complexity analyses were derived based on a simple parallel computer model. This simple model is not proposed as an implementation since machines based on centralized control often lack the flexibility, ease of programming, and extensionality desired for general-purpose computation.

The present section shows that the control of a machine based on the n-cube model can be decentralized with minimal effect, in the best case, on time complexities derived in Section 2. Of course, flexibility and ease of programming are quite subjective and no proofs can be presented for the claimed improvements. Instead, the following characteristics of the proposed machine are cited to support the claim:

1. The machine is to be programmed in the high level language, Id (Irvine dataflow) [3], instead of the assembly-like languages usually required for effective use of other multiprocessor computers. Id provides for transparent expression of parallelism (i.e., parallel operation is the default mode rather than the exception); Id is also side-effect free (functional) and shares many of the advantages of other applicative languages such as FFP [6], pure LISP [27], and LUCID [5];

2. Automatic memory management is provided along with a structured data type;

3. Id programs are independent of the number of processors or their interconnection.

Decentralized control has been demonstrated in a number of dataflow systems [12,14,15,22,42]. However, analyses of these systems has not yet produced time complexity results as good as those derived in Section 2. The sequel describes how the n-cube model and a dataflow system [22] can be combined to obtain the benefits outlined above. Dataflow is asynchronous by definition and thus each PE in the proposed machine will communicate asynchronously without centralized clock or control. Since the analyses in Section 2 depended on a central clock, the results of those analyses represent best cases for the asynchronous system. Hence, the complexity results derived in the sequel are not intended to prove an actual dataflow machine would attain these best case results because providing adequate scheduling may be difficult; rather the purpose of the analysis is to suggest that time complexity analysis is indeed a useful design and evaluation tool since by its use major bottlenecks in previous iterations of the architecture have been systematically identified and eliminated.

3.1 Overview of the Machine's Operation

As mentioned above, the dataflow machine is an asynchronously interconnected n-cube of N PEs. Although the distributed PE memory is organized as one address space each PE is solely responsible for managing its own random access memory. Program code is compiled into a data structure, so the following applies to both object code and program data. The location of each data structure is specified by a unique identifier (pointer) which may be passed anywhere in the machine. When actual data is required, the requesting PE forwards a message to the PE where the data is located; the receiving PE services the request by sending back the requested data.

The following is a brief summary of the execution of Id programs; details may be found in [3, 22]. A compiled Id program is a directed graph where each node represents an _operation_ and each link indicates that the result of one operation becomes the input to another. An operation can be any (side-effect free) function which consumes one set of inputs and produces one set of outputs. An execution instance of an operation is called an _activity_ and each activity is given a unique _activity name_. Each value resulting from an activity's execution is concatenated with the value's destination activity name into a packet called a _token_. Destination activity names are computed from the activity names of input tokens according to a set of rules located in each PE called the _U-interpreter_.

All input tokens to an activity must be directed to the same PE even though those input tokens may have been produced by many distinct PEs. The U-interpreter ensures that all tokens destined for the same activity have identical activity names. An <u>assignment function</u> is used by each PE to map the activity name of a result token to a physical PE address.[9] This address is then used to direct the token through the interconnection network to the PE holding the destination activity. Different assignment functions may be used concurrently in the machine so long as all PEs which are to send tokens to the same activity use the same assignment function.

A PE may contain many activities and each activity may be in any one of several stages of completion. The first stage commences when the activity's first input token arrives and lasts until sufficient input tokens are present to enable execution to begin. The activity then progresses through a series of stages which include operation code fetch, data fetch (if needed), operation execution, and output token generation and transmission. Activities are "multiprogrammed" within a single PE so that temporarily blocked activities (e.g., awaiting data fetch) need not monopolize execution resources. This capability allows flexibility in assigning processor resources since a single PE is sufficient to execute an entire program (assuming sufficient memory is available).

----

[9] A detailed discussion of several specific assignment functions and their effects may be found in [22].

3.2 Structures and Data Communication

The ability to specify computation on data types such as arrays, lists, records, etc. is often crucial for convenient expression of algorithms. In Id, such data types may be represented by structures. A _structure_ is a set of (selector, value) pairs where a _selector_ is an integer, and a _value_ is any value, including another structure [13]. < > denotes the empty structure while <i:x, j:v, ...> represents a structure with value x at selector i, value v at selector j, etc. Two functions are defined on structures. The _select_ function (denoted by x[i]) has two arguments, a structure x and a selector i, and yields the value at selector i. The _append_ function has three arguments: a structure, a selector, and a value to be appended to the given structure at the specified selector. Append does _not_ modify the given structure but instead makes a copy of it with the new selector and value placed appropriately. Various implementations of structures are discussed in [21]. For simplicity, only one of these implementations, the vector representation[10], is discussed in the sequel although generalization to the B-tree representation [21] has many advantages, e.g. the

---

[10] In [21] this implementation was called "array" representation.

representation of sparse arrays.

In this paper, the following implementation of structures is assumed. Each structure is associated with a unique address (pointer). In vector representation, a contiguous vector of memory cells is allocated to contain the elementary values or pointers to substructures which collectively comprise the elements of the structure. Select is implemented by indexing in the usual way. However, append in general requires a copy be made of the entire vector and the new value placed appropriately. Substructures need not be copied as is shown in [13,21]. Copying of the original vector can also be avoided when only one pointer exists which refers to that vector. In this case, append may safely update the vector in place.

Figure 3.1 shows a structure representing a NxN matrix where the (selector, value) set nearest the root is called the top level, and the collected substructure (selector, value) sets is called the bottom level. Level names can also be generalized to q-level structures where the top level is the first level, the next level from the root is the second level, and so on until the qth level. Hence a level name is derived from the path length from the root to the named level.

For the moment, the important problem of storage reclamation of structures will be ignored. We assume all object code and data aggregates are represented by structures, and furthermore that the programmer specifies (perhaps by type declaration) the largest number of elements each structure may hold so that sufficient space is initially allocated to contain it. The n-cube message transfers of Section 2 can then be implemented using structure operations as will be shown below.

A transfer begins with a _source message configuration_ and terminates with a _destination message configuration_. The _inverse_ of a transfer begins with the original destination message configuration, reverses each step of the original transfer, and terminates with the original source message configuration. Clearly the inverse transfer requires the same time as the original transfer. In a transfer based on structures, the values to be transmitted are grouped into a structure and a pointer to that structure is distributed to the PEs which are to receive one or more of the values. Each PE then sends a request (for each value it requires) to the appropriate PE which is holding that part of structure containing the required value. PEs receiving such requests service them by selecting and replying with the value requested. Each PE acts independently, but the collective effect is called a _request/acknowledge_ transfer. Each of the n-cube transfers can be implemented by the request/acknowledge mechanism without changing the best case order of time complexity.

Consider first the N-way broadcast and an additional implementation mechanism called a cache tree.[11] A cache tree is a distributed cache which automatically configures itself into the logical tree appropriate for each data item broadcast. The PE holding the value to be transmitted is the root of the broadcast tree; the tree also includes each requesting PE as well as all PEs in the paths from the requesting PEs to the root PE. The cache tree may be implemented by an associative memory table in each PE. An entry in the table consists of a two part key (a structure pointer s and a selector i) and a data field containing the value s_i if it is available; otherwise, the data field contains a pointer to a locally held list of requests received from other PEs for that same value s_i. When a PE receives a request, it looks up keys s and i in the cache. If the value is found, the PE replies as if it were the root PE with the value s_i; otherwise the request is added to the list of requests for that value. If the list was

----

[11]The cache tree was independently developed by Sullivan, Bashkow, Klappholz, and Cohn who called it a "conflict filter" [38]. However, Dr. Bashkow has indicated (by personal communication) that the cache tree is not included in current designs for the CHoPP machine, perhaps because of the difficulty of maintaining consistency in multiple copies of data. This problem does not arise in a functional environment such as dataflow because values are never modified; in practice this means cache values are read-only although redundant values other than the original source may be deleted, for example by a least recently used policy, without affecting correct operation.

previously empty, the request is also forwarded toward the root PE. When the response value s_i is received, the PE enters the value in its cache and sends a copy of the value for each of the requests in the list associated with keys s and i. An instance of a broadcast tree is thus dynamically constructed as requests filter toward the root PE and no PE need receive more than log N requests.[12] After the tree is constructed the lists of requests in the caches are used to direct the actual broadcast of the data item. In the n-cube network, any PE can be the root for a broadcast and many such broadcasts may be progressing simultaneously.[13] Clearly the best case order of time complexity for the cache tree N-way broadcast does not change over the broadcast of Section 2 since communication time is at most multiplied by a constant.

Next consider the N(N-1) transfer. (Discussion of the

----

[12]The order of request transmission must be carefully scheduled in the inverse broadcast phase to ensure no more than log N requests are actually received by any one PE. Achieving such optimum scheduling is difficult; however, scheduling policies approximating the desired behavior may prove to be adequate.

[13]Simultaneous transfers can be "timesliced" so that the total time required is the sum of the individual transfer complexities.

N-1 linear transfer will be omitted since a similar argument applies to it as well.) Let the N values to be transferred from each PE constitute one row of a NxN matrix which is represented by a structure as in Figure 3.1 where the top level may be located in an arbitrary PE of an n-cube. In the N(N-1) request/acknowledge transfer each destination PE requires a pointer to every row of the matrix. Hence in the first phase of the transfer, all PEs request via a cache tree a pointer to the first row requiring a best case time of O(log N), then the second row, and so on until the Nth row for a total best case time of O(N log N). In the second phase, each PE sends N requests for the N values it is to receive. These requests collectively form an (inverse) N(N-1) transfer. In the third phase each PE holding a row of values to be transmitted selects and sends the requested values collectively forming another N(N-1) transfer. Each PE receives N values and appends them into a vector; each such append requires only constant time because only one pointer to the structure being formed need exist. Pointers to each result row may be collected by an inverse N-1 transfer and appended together in O(N) time to produce a new NxN matrix represented by a structure. The total best case time required for the request/acknowledge N(N-1) transfer remains O(N log N) since each step requires no more than O(N log N) time.

3.3 Storage Reclamation

The above mechanisms are adequate to implement the transfer algorithms of Section 2 without affecting the order of time complexity if storage reclamation is ignored. Of course storage reclamation must be dealt with in a practical machine. The reference count scheme [1Ø] is often proposed [13,21] for storage reclamation in dataflow because structure definitions preclude circular references. In the reference count scheme, each structure has an associated non-negative integer called the reference count indicating the number of copies of the pointer referring to that structure. The reference count is incremented and decremented as copies of the pointer are respectively created and consumed. When the count is zero, the structure is no longer needed and may be reclaimed. However, in a distributed processor environment the classical reference count scheme incurs substantial communication overhead when copies of pointers are made since a request/acknowledge communication is required to update the reference count before the new pointer may be released.[14] This communication overhead is unacceptable as can be shown by considering the cache tree above. Suppose the value to be broadcast is a pointer to a structure. Then each time an internal node in the cache tree replicates this pointer, it must first send a request to the PE holding the structure to increase the reference count. Since N-1 copies of the pointer will be made in a N-way broadcast, the PE holding the structure requires O(N) time to process the reference count; thus reference count processing can increase broadcast time from O(log N) to O(N).

A generalization of classical reference counting called _weighted reference counting_ may be used to reduce such overhead [2]. In this scheme, an arbitrary positive integer called the _pointer reference weight_ (PRW) is associated with each instance of a pointer. Corresponding to the reference count in the classical scheme, each structure has an associated non-negative integer called the _structure reference weight_ (SRW) which is the sum of the PRWs of all pointers referring to that structure. (In the classical scheme, all implied pointer reference weights are equal to one and thus the SRW is the same as the reference count.) As in reference counting, when a pointer is destroyed its PRW must be subtracted from the referenced structure's SRW. However, when m copies of a pointer with a PRW equal to x are made, if $x \geq m$ then no change to the SRW is required. In this case the pointer's PRW may be "split"

---

14 The PE making the copy must wait for acknowledgment that the reference count was actually increased. Otherwise, the asynchronous operation of the machine could allow a reference count decrement to occur from the destruction of an otherwise unrelated instance of the pointer; this could lead to premature reclamation of the structure.

into arbitrary positive integers $x_1,...,x_m$ such that $x = x_1+...+x_m$ where x_i becomes the PRW on the ith copy of the pointer. (The original instance of the pointer is destroyed.)

Since changes to the SRW may be avoided when copies of a pointer are made, the reference weight scheme can dramatically reduce time overhead. For example, reconsider the problem of broadcasting a structure pointer to N PEs. If the PRW of this pointer is at least N then the broadcast can be done in O(log N) time because splitting the pointer's PRW at each internal node of the cache tree increases each PE's work by only a constant factor.

The cache tree can also reduce time overhead in a SRW decrease operation. Suppose N PEs each request the value s_i from structure s. Each request contains a pointer referring to s and since select destroys the pointer it receives as an argument, the PRW of the pointer referring to s in each of the N requests must be subtracted from the SRW of s. This can be done in O(log N) time by having each internal cache tree node accumulate the decrease in SRW as it services incoming requests for s_i. However, when an internal cache tree node has no a priori information about the number of s_i requests it will receive, it is not convenient to accumulate all such decreases in SRW before sending the first request for s_i to the root PE. This problem can be handled by delaying transmission of decrease

SRW messages until copies of s_i have actually been delivered to the PEs which originally requested s_i. These PEs then initiate an additional inverse broadcast to sweep together all decrease SRW messages into one decrease SRW value which is forwarded to the PE holding structure s. The total time required for the request/acknowledge data broadcast and a broadcast to accumulate the SRW decrease messages remains O(log N). Other policies with various tradeoffs in timeliness of storage reclamation versus concurrency potential are also possible using the reference weight scheme.

3.4 NxN Matrix Multiply and N Element Quicksort Complexity

The above mechanisms are adequate to implement the matrix multiply and quicksort algorithms of Section 2 without affecting the best case order of time complexity. (The PDE problem is considered in a later subsection.) An Id program for matrix multiply is given in Figure 3.2 where the more readable syntax "new x[i] ← v" represents "new x ← append(x,i,v)" for appends within loops and "x[i,j,...,m]" represents "select (... select (select(x,i), j)..., m)" within expressions. This program differs from a conventional matrix multiply program in two respects. First, the matrices are represented by structures (Figure 3.1) and thus A[i] returns a pointer referring to the ith row of matrix A. Second, since structure representation biases element access (i.e., in lexicographic order) the B matrix is first transposed before the multiplication is performed so that rows of B^t represent columns of B.

The Id program for matrix multiply (as well as all other programs considered in this paper) is independent of the size of the input. Thus the program can be distributed using a cache tree to N^2 PEs in O(log N) time; hence all PEs are assumed to hold a complete copy of the program although this would not necessarily be the case in a real machine. As was discussed earlier dataflow code is a collection of interconnected functions and copies of each function may begin execution when the necessary arguments arrive. However, for the purpose of complexity analysis it is sometimes helpful to view the initiation of function execution differently. A PE is said to initiate the execution of a function if it supplies all arguments required for that function. A part of each of the following complexity analysis determines the time for a single PE (the starting point for the computation) to initiate the various parts of a program using the transfer mechanisms of Section 2.

The following concerns the time complexity of matrix transpose (Figure 3.2). The rows of each matrix are assumed to be distributed as described in Section 2.3.2.1

while the top level of each structure may reside in any one of the PEs also holding a row of the same structure. The transpose procedure and its nested i-loop may be initiated in any PE holding a row of matrix B. Under the U-interpreter, the i-loop then initiates N copies of the j-loop each in a distinct PE holding a row of B in O(N) time using a N-1 linear transfer. The transpose can then be completed in O(N log N) time simply by using a request/acknowledge N(N-1) transfer.

In the multiply procedure, each i-loop initiates N copies of the j-loop in N distinct PEs and each j-loop then initiates N copies of the k-loop for a total of N^2 k-loop initiations in N^2 PEs all in O(N) time. The time for structure access is determined in the following. Since row_A is one of the inputs to a k-loop initiation, sufficient copies of pointers referring to rows of matrix A are made by the U-interpreter without increasing the order of time complexity of program initiation (assuming the reference weights of the original pointers are large enough). Each k-loop initiation then uses a single copy of one of these pointers N times to request the N elements of a row of A. The case for matrix B is slightly different since $B^t[j]$ occurs in the j-loop which generates a total of N^2 requests for pointers. These requests may be satisfied in $O(N \log N^2) = O(N \log N)$ time using a cache tree. Thus all requests for pointers may be satisfied in O(N log N) time. The actual data elements are then acquired by transfers as described in Section 2 but using request/acknowledge communication. Pointers referring to rows of the result matrix can then be appended together using a N-1 linear transfer in O(N) time. Thus matrix multiply can be done using structures to represent matrices, a high level Id program to represent the algorithm, and the pointer reference weight scheme to provide for storage reclamation without increasing the best case order of time complexity over that derived in Section 2.

An Id program for quicksort is given in Figure 3.3. Since the numbers to be sorted can be represented in a one-level structure, no special consideration for structure representation is needed. Thus the data transfer algorithm given in Section 2.3.3 is directly applicable in this case to give a best case time complexity of O(N).

3.5 I-structures

If an iterative solution to the PDE problem is programmed using a q-level structure, the pipelining and hence the degree of parallelism described in Section 2 will be lost. Consider the compilation[15] of a simple Id loop which builds a structure x (Figure 3.4). The L and L^{-1} boxes generate and strip away, respectively, context for each loop initiation much like conventional systems generate and strip away context for each initiation and return of a procedure. Similarly, the *D* and *D^{-1}* *boxes* generate and strip away a unique context for the values within each iteration of the loop. Such context changes are directed by the U-interpreter and need not be considered further here. Boxes with internal values such as < > or v produce that value when triggered by any input token. The $\otimes$ *operator* performs the identity function by passing each input token directly to its output port. The *switch operator* decides to which output port (T or F) each input token is to be sent based on the corresponding boolean valued token received at its side port. *Forks* in lines indicate that the token input to the fork is to be replicated so identical tokens are placed on each output line.

The point of interest in Figure 3.4 is that the output of the append box, *new* x, is circulated on each iteration of the loop and thus x does not appear on the loop *return* line until the loop terminates. This ordered construction of x is required by the semantics of structures as can be seen in the following example. Replace the third line of the program in Figure 3.4 with "*new* x $\leftarrow$ append(x, f(i),

----

[15]Id compilation is discussed in detail in [3].

g(i))". Since f(i) need not be a one-to-one function, the selectors used for the m appends need not be distinct. Hence the final value at each selector of structure x is not known until the loop terminates. The effect of these build-before-use structures on the PDE problem is to delay initiation of the computation for the next iteration of the outer loop until the current iteration is complete, and hence no pipelining between outer loop iterations is possible.

One solution to this problem is to use I-structures instead of structures.[16] _I-structures_ may be regarded as structures constructed in a restricted way. In the sequel only one operational semantics and implementation of I-structures is considered. Arvind and Thomas present a more complete theory of I-structures and compare I-structures with other functional data types [4]. The restriction on I-structure construction used here is that the value at each selector of a particular structure may be appended to at most once (termed the _single assignment_ rule for selectors).[17]

The single assignment rule suggests an I-structure implementation which allows values to be selected from an

----

[16]Another solution is to use a data type called "stream" [3] instead of structures. However, streams are often inappropriate for expression of numerical algorithms [4].

I-structure before that I-structure is complete. Figure 3.5 shows the compilation of the same program fragment as in Figure 3.4 but where x is considered an I-structure instead of a structure. The I-structure _pointer generator box_ allocates memory for the I-structure (the bounds of the I-structure must be supplied), initializes the value at each selector to the _not-present_ or _empty value_[18], and sends out two pointers referring to the I-structure. For error checking these pointers are marked "read-only" or "updateable" since in this simple model only the inside of the loop is allowed to append to the I-structure. This allows the _clean up box_ to convert the I-structure to a structure when the loop terminates by changing all empty values remaining in the structure to the _undefined value_. Note that the output of pointer x does not depend on termination of the loop.[19] Thus values at individual selectors of an I-structure may be selected from outside

----

[17]This rule differs from the single assignment rule for program variables [9] since the validity of the rule for program variables may be determined at compile time. The validity of the single assignment rule for structure selectors cannot in general be determined until execution is complete as was shown in the example above where function f(i) determined the selectors.

[18]Alternatively, memory could be mass initialized before allocation and then each "memory block" would be reinitialized on deallocation.

the loop as each value is appended to the I-structure within the loop.

Implementation of I-structures is similar to structures except a <u>presence bit</u> is associated with each I-structure selector. This presence bit is checked when a select is attempted from an I-structure. If the presence bit is on, select simply returns the value at that selector. However if the presence bit is off, the value at that selector is really a pointer to a list of <u>select requests</u> for that value. Each such request is delayed by adding the request to the list; the PE servicing that request may then go on to other tasks. When the value eventually becomes available through an append operation, each request on the list for that selector is satisfied by sending a copy of the selected value. Append also checks the presence bit when appending to an I-structure. If the presence bit is already on for the selector being appended to then the single assignment rule for that selector has been violated and an appropriate error message may be issued.

----

[19]Since termination of I-structure programs does not depend on termination of embedded loops, I-structure programs are more defined in the sense that an I-structure program may produce results when an otherwise equivalent structure program does not.

3.6 Partial Differential Equation Complexity

I-structures allow the pipelining desired in the PDE problem since I-structures need not be complete before values are selected. The following considers the time complexity of an I-structure program which solves for one variable of a q-dimensional $s_{q-1}X...Xs_{\emptyset}$ PDE (Figure 3.6). Assume the initial data is laid out in a $s_{q-1}X...Xs_1X\ O(s_{\emptyset})$ OW MCC as described in Section 2 but where the data is represented as a q-level structure $x_{\emptyset}$. Recall each s_i includes boundary data at indices $\emptyset$ and s_i-1, while the coordinate indices $i_{q-1},...,i_{\emptyset}$ indicate a PE address in a MCC, and a list of fixed coordinates in parentheses represents a MCC hyperplane. Assume the levels of structure $x_{\emptyset}$ are distributed such that $x_{\emptyset}[j_{q-1},...,j_m]$, $\emptyset<m\leq q$, is located in any PE in MCC hyperplane $(i_{q-1}=j_{q-1}, ..., i_m=j_m, i_{\emptyset}=\emptyset)$ where m=q means the only restriction is $i_{\emptyset}=\emptyset$, e.g. Figure 3.7. Thus the top level may reside in any PE in hyperplane $(i_{\emptyset}=\emptyset)$ while lower levels are restricted to the MCC hyperplanes from which most select requests for each structure will originate in the nearest neighbor access pattern of the PDE program (Figure 3.6). Data resulting from each succeeding k-loop iteration is to be constructed as a q-level I-structure to allow pipelining. If the placement of the I-structures for each of these k-loop iterations also meets the criterion above (except $i_{\emptyset}=k$ instead of $\emptyset$), then the computation can

proceed nearly the same as the PDE computation described in Section 2.3.5.

The first step in the complexity analysis of the PDE program is to determine the time for program initiation. Recall that T is the number of k-loop iterations required for convergence. Then the time for the program to unfold under the U-interpreter is $O(T + s_{q-1}+\ldots+s_{\emptyset})$ since the first PE will spawn T i-loops,[20] each of these will concurrently spawn s_{q-1} j-loops, etc. Initialization of the I-structures for <u>all</u> iterations can be done in time linearly proportional to program initiation time since the size of the vector to be initialized within the loop in each case equals the number of subloops to be initiated.[21] In addition, after each I-structure is initialized a pointer referring to that I-structure is released in constant time (Figure 3.5) and is returned as a value to

----

[2Ø]This analysis assumes the i-loop including its nested loops may be initiated before all inputs to the loop are available. Otherwise, initiations of these loops would require initialization of the I-structure x from the previous iteration of k. In theory, waiting for this initialization tends to negate the advantage of I-structures over structures. However, in practice the initialization could be made very fast relative to other operations in the machine. Such issues have been avoided in the analysis by assuming that not all inputs are required to initiate a loop.

[21]In practice such uncontrolled initiation of k-loop iterations and allocation of I-structures would probably waste memory without improving performance over a policy which delays initiation so that no more than one (or a small number of) concurrent k-loop iteration(s) exist per MCC hyperplane.

the I-structure within the next outer loop. Thus all I-structures may be completely constructed except for the elementary values at the bottom level in time linearly proportional to program initiation time. The computation may then proceed as described in Section 2.3.5 except that q levels of structure x must be traversed to access each element x[i,j,...,m]. The top level contains s_{q-1} selectors and by means of a cache tree all PEs in MCC hyperplane (i_0=k) may be sent the required pointer values in

$$O(s_{q-1} \log (s_{q-1}*...*s_1))$$

time since there are $s_{q-1}*...*s_1$ PEs in that hyperplane which is also a (log ($s_{q-1}*...*s_1$))-cube by Corollary 2.2b. Similarly, all requests for pointers in the next lower level can be serviced in

$$O(s_{q-2} \log (s_{q-2}*...*s_1))$$

time within each of s_{q-1} distinct (log ($s_{q-2}*...*s_1$))-cubes synonymous by Corollary 2.2b with the s_{q-1} MCC hyperplanes (i_{q-1}=j, i_0=k), $0 \leq j < s_{q-1}$, and so on for each structure level until the level one removed from the bottom which requires

$$O(s_1 \log s_1)$$

time. By expanding the log term in each of these equations to make them identical, the sum for all levels to satisfy these pointer requests simplifies to

$$O(s_{q-1}+\ldots+s_1 \log (s_{q-1}*\ldots*s_1))$$

Assume the values x[i,j,...,m] are concurrently requested by all PEs for all k-loop iterations, followed by a request for values x[i+1,j,...,m], etc. for a total of O(q) values per PE.[22] The time complexity of the overall PDE computation is

$$O(T + s_{q-1}+\ldots+s_\emptyset) + O(q(s_{q-1}+\ldots+s_1 \log (s_{q-1}*\ldots*s_1)))$$

plus the time to do the actual computation from Section 2.3.5,

$$O(q(s_\emptyset+T))$$

for a total of

$$O(q((s_{q-1}+\ldots+s_1 \log (s_{q-1}*\ldots*s_1)) + s_\emptyset+T))$$

If $N=s_{q-1}=s_{q-2}=\ldots=s_\emptyset$ then this equation reduces to $O(q^3 N \log N + qT)$. In comparison, the complexity of the

----

[22]Note that the caches would automatically tend to eliminate the need for actual traversal of all q levels for each of the 2q+1 values required to compute each result value. In addition, the PDE program could be modified to minimize redundant top level selects as was done in the matrix multiply program. For simplicity, these options were ignored in the analysis since they have little effect on the overall order of time complexity.

same problem on a q-dimensional array computer where each dimension is also of size N is O(qT) while on a sequential machine the complexity is $O(q\ N^q\ T)$.

4.0 EXPERIMENTAL RESULTS

Previous sections have analyzed network transfers, programs, and mechanisms in terms of computational time complexity. In this section analytic results on I-structures are supported with evidence from executing machine-compiled Id programs on the Irvine dataflow simulator [22].

4.1 The Irvine Dataflow Simulator

Although complete simulation of the architecture described in Section 3 would be desirable, so far this task has not been attempted. Instead the Irvine dataflow simulator was modified to independently test the utility of I-structures. Although the results are not directly applicable to the architecture described in this paper, complexity analyses indicate I-structures should reduce execution time of many programs on both architectures.

The following is a brief description of the simulated architecture; details may be found in [22]. The Irvine dataflow simulator is a detailed deterministic simulation of a particular interconnection of PEs. Some PEs called memory controllers (MC) are specialized to manipulate structures and perform memory management. The interconnection network is shown in Figure 4.1 where points A and A' are connected together to form two counter-rotating *token bus* rings. Each ring is partitioned into as many slots as there are PEs and each slot is either empty or holds one fixed-length token. Four PEs are connected together and to a memory controller by a *local bus* which carries structure access requests and responses. Each memory controller is directly connected to a private, conventional memory (M) organized as part of one unified address space. MCs are connected together by a *global bus* so every PE has indirect access to any data or code within the machine. A group of four PEs connected by a local bus to one MCC is termed a *physical domain*. The collection of PEs and MCs connected by the same counter-rotating token bus is called a *ring domain* which is the largest group now simulated. Assignment functions (Section 3.1) are chosen so closely connected activities (e.g., the activities comprising an instance of a procedure or loop body) are confined to the same physical domain. Since tokens are transmitted on that token bus which provides the shortest distance path to its destination, such assignment functions tend to reduce communication traffic between physical domains thereby promoting unimpeded local communication within concurrently operating physical domains.

4.2 Experiments: I-structures versus Structures

For simplicity all communication conflicts are ignored in the following complexity analysis intended to aid in understanding the experimental results (previous complexity results are not directly applicable to the simulated architecture). Consider once again a q-dimensional $s_{q-1}X\ldots Xs_{\emptyset}$ PDE program (Figure 3.6). As discussed in Section 3.5, if this program were implemented using structures the structure y and all of its substructures must be complete before the next k-loop iteration can begin. Since each k-loop iteration is dependent only on data from the previous k-loop iteration, recall that the q nested loops within the k-loop may unfold under the U-interpreter and thus the time required to complete each k-loop iteration is $O(s_{q-1}+\ldots+s_{\emptyset}+q)$. Therefore the total time required by the structure program is $O(T(s_{q-1}+\ldots+s_{\emptyset}+q))$ where T is the total number of k-loop iterations. For the one-dimensional planar hydrodynamics code executed in the simulator[23] this equation reduces to $O(Ts_{\emptyset})$. For ease in understanding experimental results, the convergence test was removed and the number of k-loop

---

[23]This code was donated to the University of California by the Lawrence Livermore Laboratory. The code is a declassified and simplified version of a program which simulates shock wave interactions by solving large PDEs.

iterations was artificially set to $s_\emptyset$ giving a complexity for the structure program of $O(s_\emptyset^2)$. By comparison, an equivalent I-structure program requires $O(T+s_\emptyset) = O(s_\emptyset+s_\emptyset) = O(s_\emptyset)$ since iteration k+1 may begin as soon as the first three values (in a one-dimensional problem) have been computed in iteration k, etc.

To test this analysis the compilation of structure variables in Id loops was changed from the structure schema (e.g., Figure 3.4) to the I-structure schema (e.g., Figure 3.5). The hydrodynamics source code was not modified in any way. A series of experiments was then conducted by varying the number of PEs in the machine for each problem size $s_\emptyset$. The minimum execution time from these experiments for each problem size was plotted in a _complexity graph_ showing the change in minimum execution time versus problem size (Figure 4.2). This graph illustrates substantial execution time reduction when I-structures are used even for small problem sizes. Although the I-structure curve appears almost linear, communication conflicts in this architecture would eventually cause the curve to bend upwards for larger problem sizes thus showing a real complexity higher than $O(s_\emptyset)$. However the coefficients for higher order terms in the actual complexity are small enough so that these terms do not appear in the experimental results when the simulator is in the "standard configuration" [22].

In the case of Gaussian elimination, similar complexity analysis (again ignoring communication conflicts) leads to $O(N^2)$ time to solve N equations with N unknowns by a structure program while the time for an I-structure program is O(N). These analyses are supported (although not as dramatically) by the experimental results given in Figure 4.3. Note that Gaussian elimination on a sequential machine requires $O(N^3)$ time.

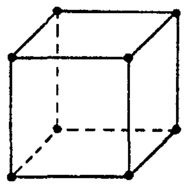
5.Ø CONCLUSIONS:

Our goal is to design a general-purpose computer comprising large numbers of cooperating processors to reduce execution time without increasing software costs. Toward this goal we chose to base the design of the architecture on dataflow since the traditional von Neumann model appears little suited for large scale multiprocessing [6,13,2Ø,22].

New computer architectures are often justified by their "cleverness quotient" or by listing concurrent operations. The difficulty with such evaluations is either extreme subjectivity or irrelevance since cleverness or concurrency gains little when unforeseen bottlenecks dominate execution time. Instead we suggest complexity analysis as a tool appropriate for designing and evaluating multiprocessor architectures since complexity analysis quickly uncovers major bottlenecks. However, the complexity analyses in this paper have two shortcomings. First, large constants may be hidden in the O-notation used to simplify analysis; hence small computations may perform relatively poorly. Second, only a few numerical algorithms were analyzed and thus more work is needed to show that the architecture is indeed suitable for general-purpose computation.

The asynchrony and decentralization present in the proposed architecture makes accurate analysis difficult and thus salient features were abstracted to derive a parallel computer model more amenable to analysis. This model is representative of a class of models since the binary n-cube on which it is based can be directly simulated by a number of other networks. For example, Siegel has shown that each of the n-1 sets of interconnections, cube_i $0<i<n$, of an n-cube can be simulated with a "SIMD perfect shuffle" network in n+1 steps while the set of interconnections, cube_0, requires only one step [34]. Since the parallel computer model allows simultaneous cube_i transmissions in one step (i.e., each transmitting PE may independently use a different link i) and there are n-1 such $i \neq 0$, the total for the SIMD perfect shuffle to simulate the parallel computer network is $(n-1)(n+1)+1 = n^2$. Thus complexity results derived for the parallel computer model are valid for the SIMD perfect shuffle when multiplied by at most n^2. In addition, the binary n-cube was found to have a number of properties facilitating parallel complexity analysis including regularity, concurrency potential, and ease of creating partitions. However, the centralized clock of the parallel computer model detracts from machine modularity and programming flexibility. Section 3 of the paper showed that the clock and other control could be decentralized with minimal effect on the best case order of time complexity where some of the mechanisms proposed to do this included request/acknowledge transfers, weighted reference

counting, cache trees, and I-structures.



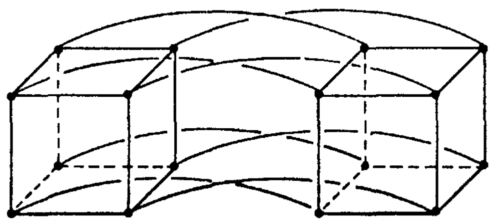


Figure 2.1
Examples of n-cubes

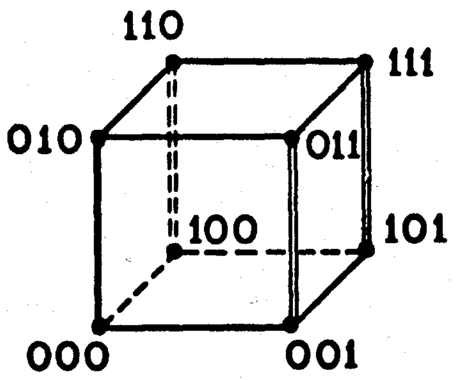


Figure 2.2a
The four **1-cubes** of one of the three (=3!/(1!2!)) possible $4_{3,1}$-partitions of a 3-cube are indicated by double lines (n=3, m=1, k=4, $4_{3,1}$-partition=$\{1\}$).

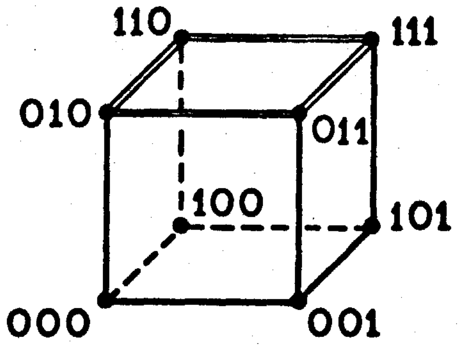


Figure 2.2b
One of the two 2-cubes sharing exactly one PE with each of the 1-cubes in Figure 2.2a is shown with double lines (complement of $4_{3,1}$-partition = $2_{3,2}$-partition = $\{2,0\}$).

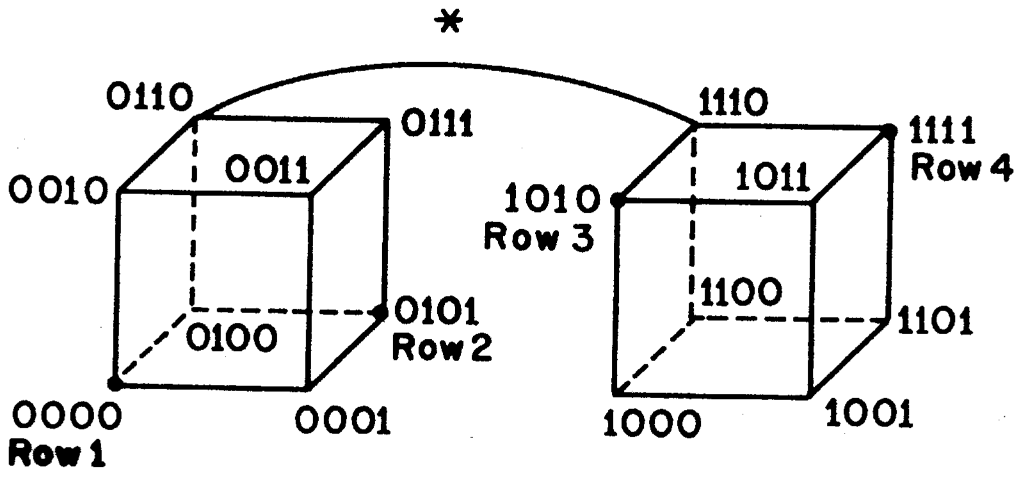


Figure 2.3a
Initial configuration of rows of matrix A in a 4x4 matrix multiplication.

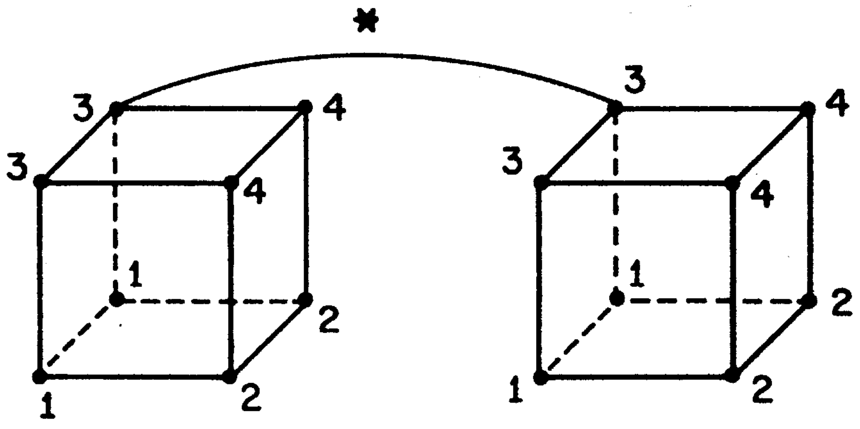


Figure 2.3b
Result of broadcasting rows of A over 4th dimension and front to back edges (i.e., the 2-cubes of partition { 2,3 }). The numbers indicate row numbers of matrix A.

*Only one of eight 4th dimension connections is shown.

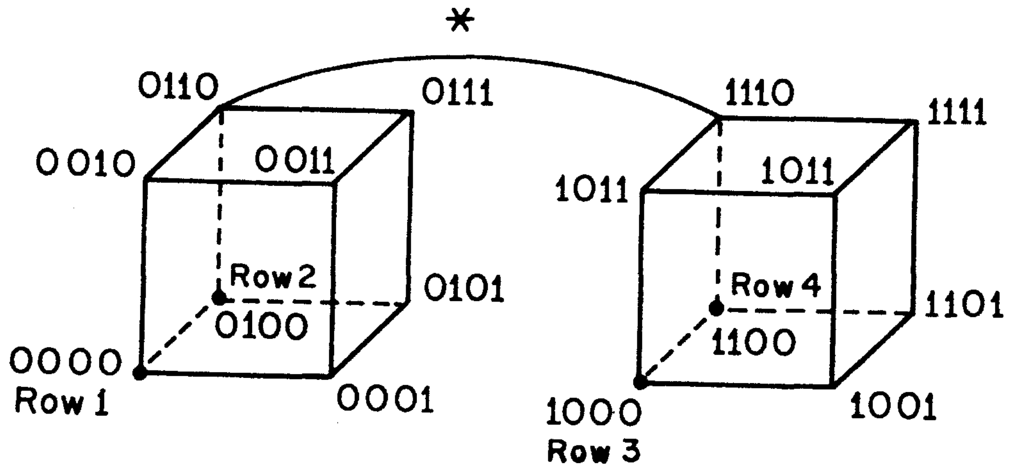


Figure 2.3c
Initial configuration of rows of matrix B and also the resulting configuration of B^t.

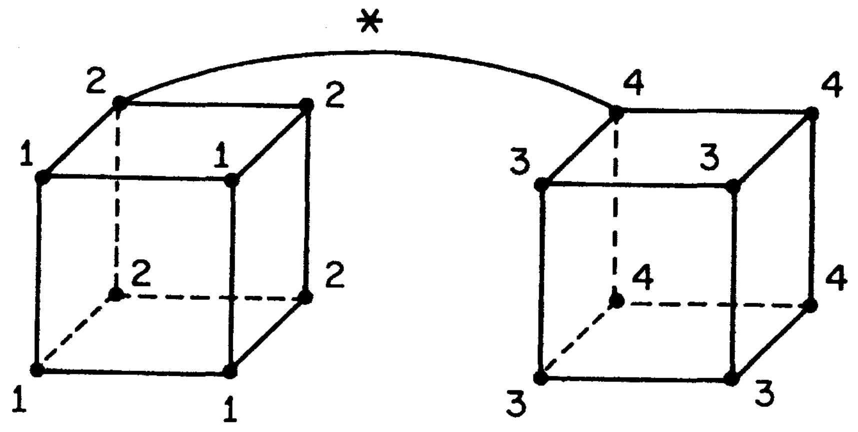


Figure 2.3d
Result of broadcasting rows of B^t over front and back faces (i.e., the 2-cubes of partition=$\{0,1\}$). The numbers indicate row numbers of matrix B^t.

*Only one of eight 4th dimension connections is shown.

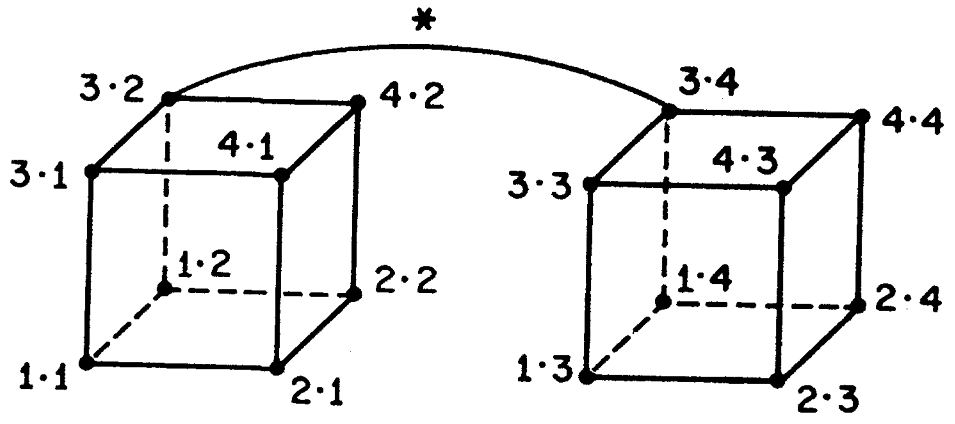


Figure 2.3e
Combined results of A and B^t broadcasts with inner products ready to be computed.

*Only one of eight 4th dimension connections is shown.

*Only one of eight 4th dimension connections is shown.

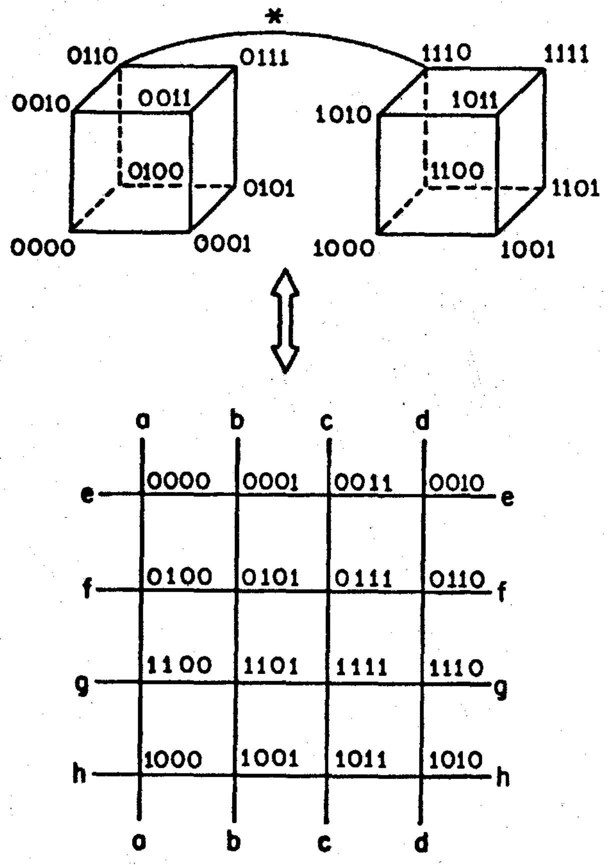


Figure 2.4
A 4-cube is topologically equivalent to a 4x4 MCC with orthogonal wraparound. (All labels represent corresponding n-cube addresses; MCC labels are not shown.)

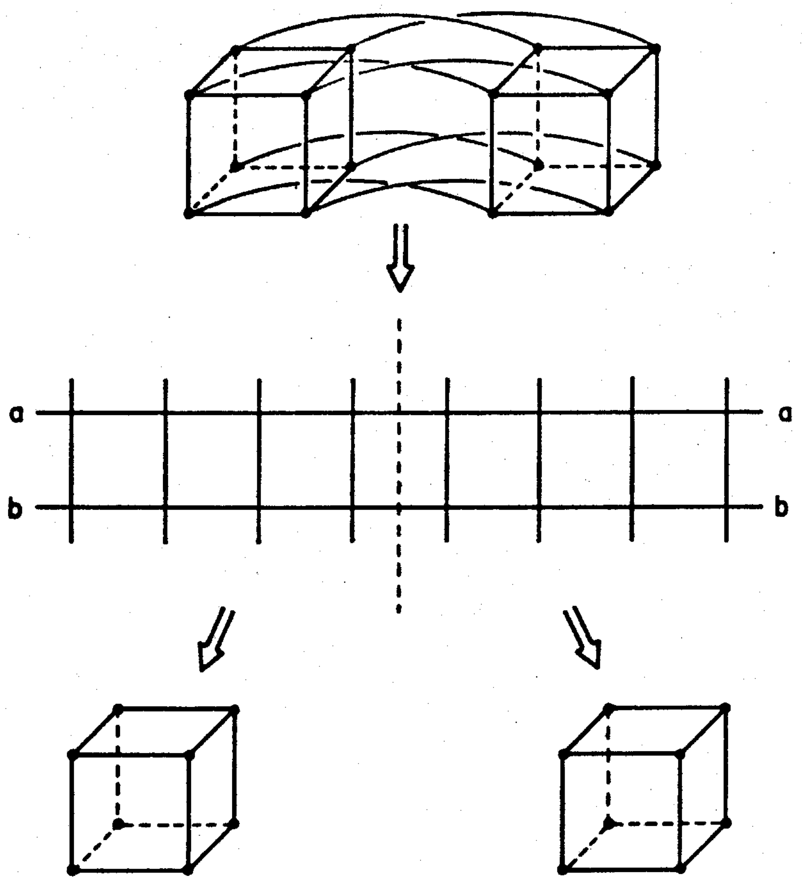


Figure 2.5
A 4-cube is configured as a 2x8 MCC (unneeded connections are ignored). A hyperplane (line) then bisects the MCC into two 3-cubes where some of the connections are recovered from those ignored during the original configuration.

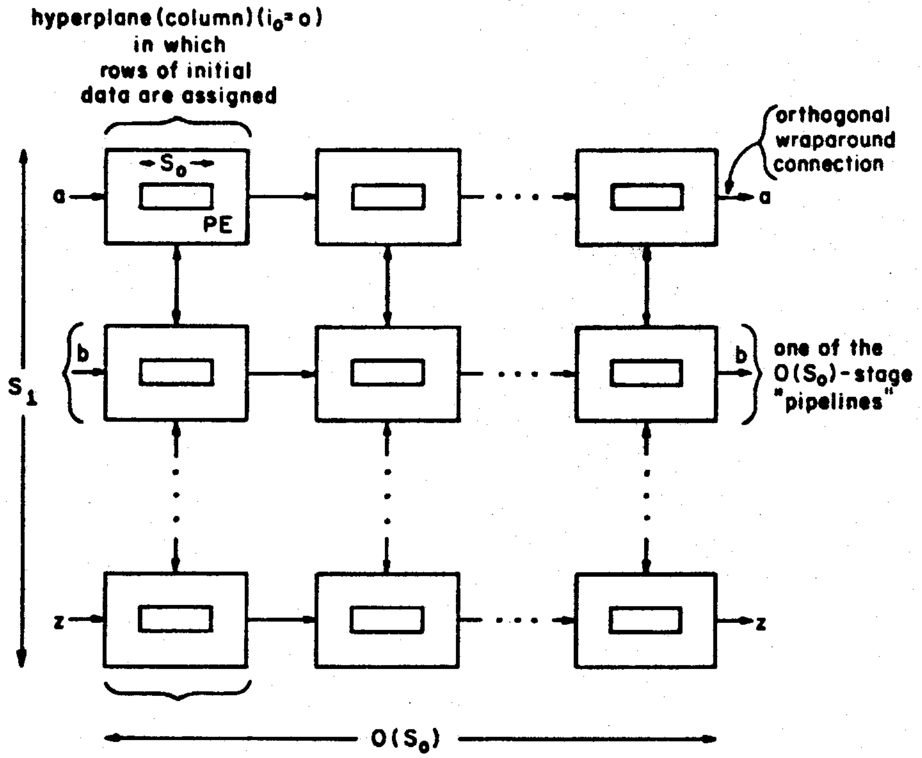

hyperplane (column) (i_0 = o) in which rows of initial data are assigned
S_0
PE
a
a
orthogonal wraparound connection
b
b
one of the O(S_0)-stage "pipelines"
S_1
z
z
O(S_0)

Figure 2.6
Example layout of a 2-dimensional $s_1 X s_0$ PDE problem

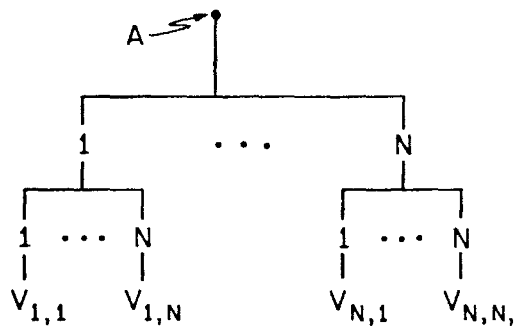


Figure 3.1
Structure representation of a NxN matrix

```
procedure transpose (B, N)
  (trans : structure[1..N] ;     ! see note* !
   initial trans ← < >
   for i from 1 to N do
     new trans[i] ← (
       row : structure[1..N] ;
       initial row ← < >
       for j from 1 to N do
         new row[j] ← B[j,i]
       return row)
   return trans) ;


procedure multiply (A, Bt, N)
  (C : structure[1..N] ;
   initial C ← < >
   for i from 1 to N do
     row_A ← A[i] ;
     new C[i] ← (
       row_C : structure[1..N] ;
       initial row_C ← < >
       for j from 1 to N do
         col_B ← Bt[j] ;
         new row_C[j] ← (
           initial inner_prod ← Ø
           for k from 1 to N do
             new inner_prod ← inner_prod + row_A[k]*col_B[k]
           return inner_prod)
       return row_C)
   return C)
```

Figure 3.2

The call "multiply(A, transpose(B,N), N)" returns the product of NxN matrix A and NxN matrix B.

*This Pascal-like declaration is not part of Id and is for illustrative purposes only.

```
procedure quicksort (A, N)
  (m ← N div 2 ;
   below, j, above ← (
     above, below : structure[1..N] ;     ! see note* !
     initial below ← < > ;  j ← Ø ;
             above ← < > ;  k ← Ø
     for i from 1 to N do
       (if i ≠ m
          then (if A[i] ≤ A[m]
                  then new below[j+1] ← A[i] ;
                       new j ← j+1
                  else new above[k+1] ← A[i] ;
                       new k ← k+1))
     return (if j > 1  then quicksort(below, j)
                       else below),
             j,
             (if k > 1  then quicksort(above, k)
                        else above))
   return (
     sorted : structure[1..N] ;
     initial sorted ← append(below, j+1, A[m])
     for i from j+2 to N do
       new sorted[i] ← above[i-j-1]
     return sorted))
```

Figure 3.3
Id procedure to sort N element vector A

*This Pascal-like declaration is not part of Id and is for illustrative purposes only.

```
(initial x ← < >
 for i from 1 to m do
   new x ← append(x,i,v)
 return x)
```

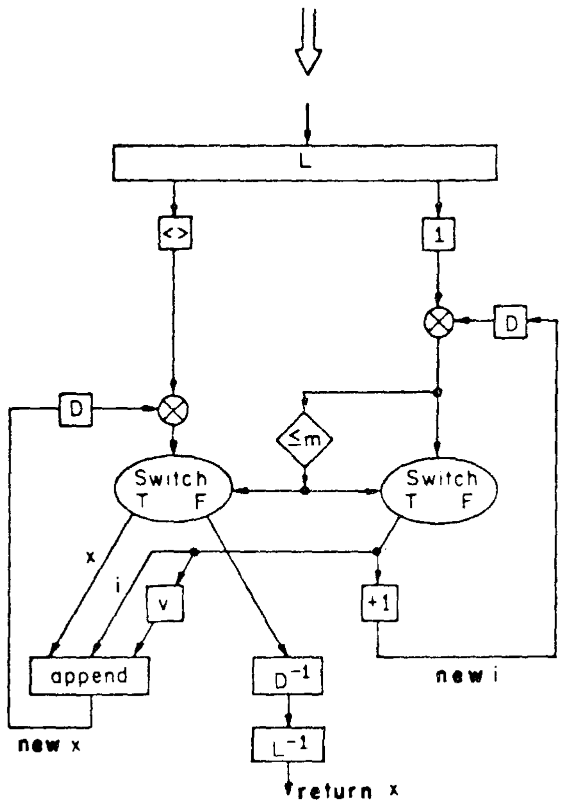


Figure 3.4
Example compilation of an Id loop expression

```
(initial x ← < >
 for i from 1 to m do
   new x ← append(x,i,v)
 return x)
```

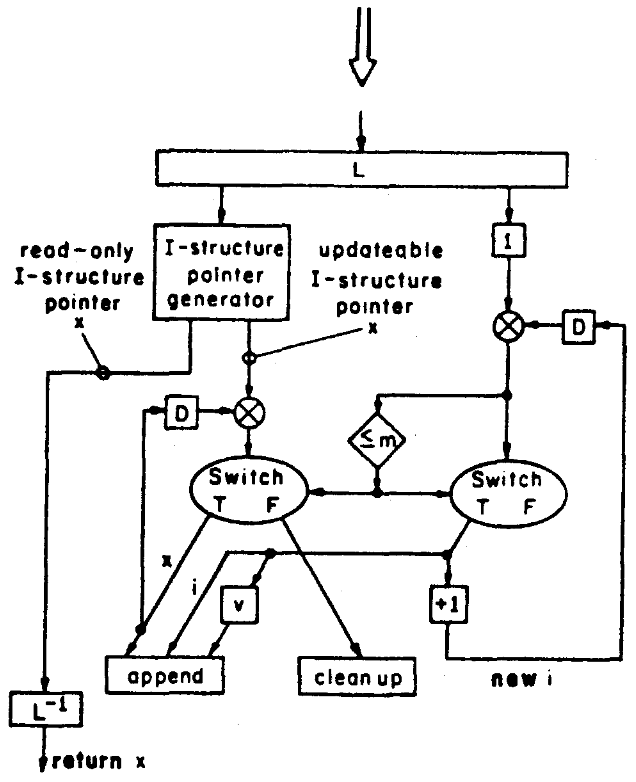

L
read-only I-structure pointer x
I-structure pointer generator
updateable I-structure pointer x
1
D
D
≤m
Switch T F
Switch T F
x
i
v
+1
append
clean up
new i
L^{-1}
return x

Figure 3.5
Compilation of the same program fragment as in Figure 3.4 but treating x as an I-structure rather than a structure

```
(initial x ← $x_Ø$    ! $x_Ø$ contains initial PDE data
                          represented by a q-level structure !

 for k from 1 until (convergence is achieved) do

   new x ← (
     y : I-structure[Ø..$s_{q-1}$-1] ;     ! see note* !

     initial y ← <Ø:x[Ø], ($s_{q-1}$-1):x[$s_{q-1}$-1]>

     for i from 1 to $s_{q-1}$-2 do

       new y[i] ← (
         y_sub_plane : I-structure[Ø..$s_{q-2}$-1] ;

         initial y_sub_plane ← <Ø:x[i,Ø], ($s_{q-2}$-1):x[i,$s_{q-2}$-1]>

         for j from 1 to $s_{q-2}$-2 do

           new y_sub_plane[j] ← (
                     .
                     .
                     .
               row : I-structure[Ø..$s_Ø$-1] ;

               initial row ←  <Ø:x[i,j,...,Ø], ($s_Ø$-1):x[i,j,...,$s_Ø$-1]>

               for m from 1 to $s_Ø$-2 do

                 new row[m] ← x[i,j,...,m]/2 +
                                (x[i+1,j,...,m] + x[i-1,j,...,m] +
                                 x[i,j+1,...,m] + x[i,j-1,...,m] +
                                         .
                                         .
                                         .
                                 x[i,j,...,m+1] + x[i,j,...,m-1])/(4*q)
               return row)
                     .
                     .
                     .
         return y_sub_plane)
     return y)
 return x)
```

Figure 3.6
Sample Id program to solve for
variable x (e.g., temperature) of a PDE

*This Pascal-like declaration is not part of Id and is for illustrative purposes only.

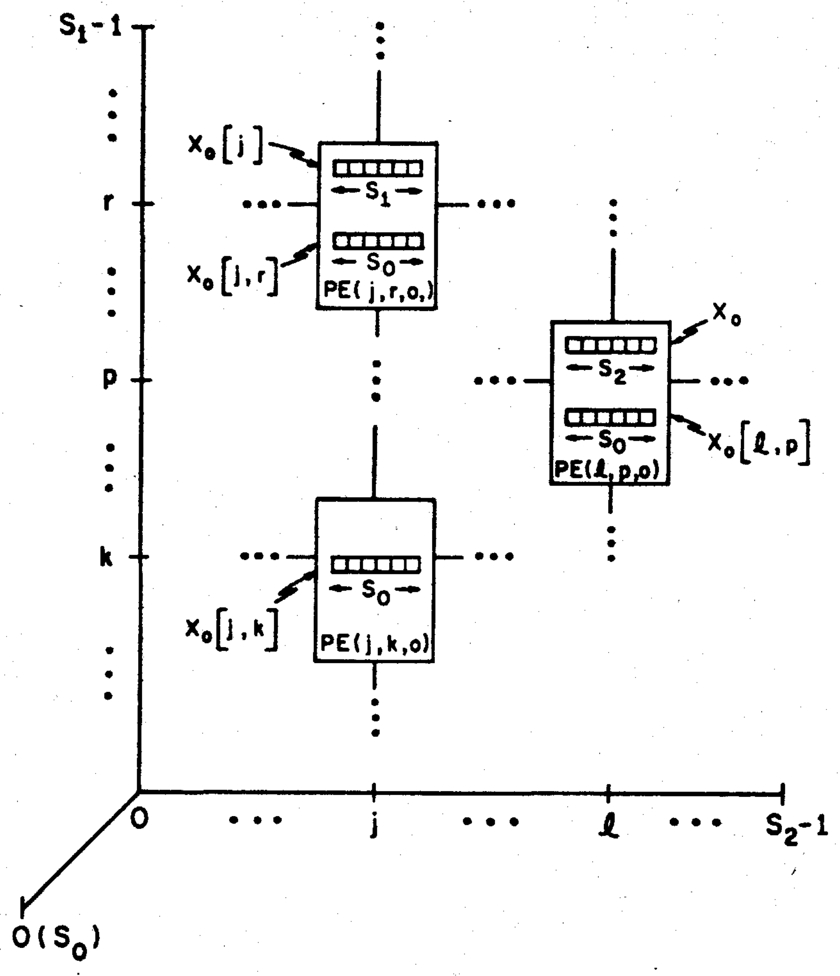


Figure 3.7
(q=3)-dimensional PDE initial data structure x_0 layout is shown for MCC plane (i_0=0). Top level of x_0 may reside in any PE (e.g., rightmost PE above) in MCC plane (i_0=0), x_0[j] may reside in any PE (e.g., top PE above) in MCC column (i_2=j, i_0=0), and data vector x_0[j,k] must reside in PE(j,k,0).

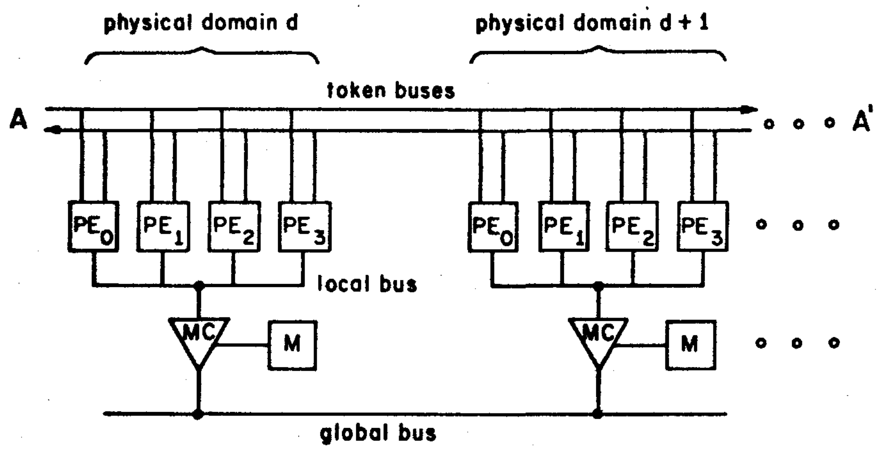


Figure 4.1
The interconnection of processors (PEs), memory controllers (MCs), and memories (Ms) in the Irvine dataflow simulator.

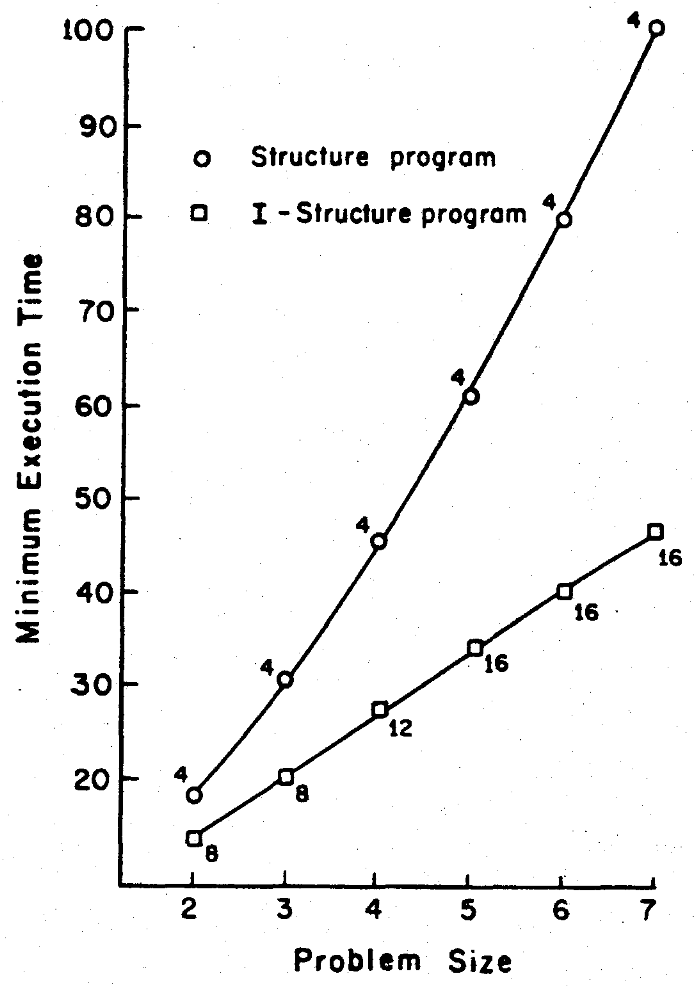


Figure 4.2
Execution time complexity curves for a one-dimensional planar hydrodynamics simulation. The curves show that I-structures reduce execution time through increased parallelism. The number of PEs used to achieve the minimum execution time appears adjacent to each point.

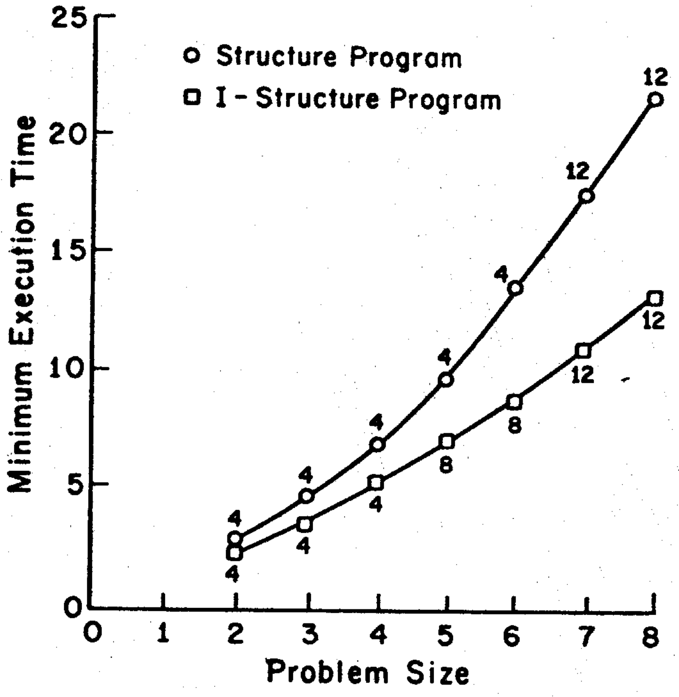


Figure 4.3
Execution time complexity curves for Gaussian elimination. The number of PEs used to achieve the minimum execution time appears adjacent to each point.

REFERENCES

1. Aho, A.V., J.E. Hopcroft, and J.D. Ullman, _The Design and Analysis of Computer Algorithms_, Addison-Wesley, Reading, MA, 1974.

2. Arvind, personal communication.

3. Arvind, K.P. Gostelow, and W.E. Plouffe, "An asynchronous programming language and computing machine," TR. 114a, Dept. of Inf. and Comp. Science, Univ. of Ca., Irvine, CA, Dec. 1978.

4. Arvind, and R.E. Thomas, "I-structures: an efficient data type for functional languages," (submitted for publication), July 1980.

5. Ashcroft, E.A., and W.W. Wadge, "Lucid, a nonprocedural language with iteration", _CACM_ Vol. 20, 7 (July 1977), 519-526.

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

7. Batcher, K.E., "The flip network in Staran," _Proc. International Conf. on Parallel Processing_, Aug. 1976, 65-71.

8. Bouknight, W.J., S.A. Denenberg, D.E. McIntyre, J.M. Randall, Sameh, A.H., and D.L. Slotnick, "The Illiac IV system," _Proceedings of the IEEE_, Vol. 60, 4 (April 1972), 369-388.

9. Chamberlin, D.D., "The single-assignment approach to parallel processing," _AFIPS FJCC Proc_. Vol. 39 (Nov. 1971), 263-269.

10. Collins, G.E., "A method for overlapping and erasure of lists," _CACM_, Vol. 3, 12 (Dec. 1960), 655-657.

11. Comte, D., and N. Hifdi, "LAU multiprocessor: microfunctional description and technological choices", _First European Conf. on Parrallel and Distributed Processing Proc_., (J.C. Syre, Ed.), Feb. 1979, 8-15.

12. Davis, A.L., "The architecture and system methodology of DDM1: a recursively structured data driven machine," _Proc. Fifth Symposium on Computer Architecture_, April 1978, 210-215.

13. Dennis, J.B., "First version of a data flow procedure language," Programming Symposium: Proc. Collugue sur la Programmation (B. Robinet, Ed.), Lecture Notes in Computer Science, 19, Springer-Verlag, NY, 1974, 362-376.

14. Dennis, J.B., C.K.C. Leung, and D.P. Misunas, "A highly parallel processor using a data flow machine language," CSG Memo 134-2, Lab. for Comp. Science, MIT, MA, 1980 (to appear in IEEE Transactions on Computers).

15. Farrel, E.P., G. Noordin, and Treleaven, P.C., "A concurrent computer architecture and a ring based implementation," Proc. Sixth Symposium on Computer Architecture, April 1979, 1-11.

16. Flynn, M.J., "Some computer organizations and their effectiveness," IEEE Transactions on Computers, C-21, 9 (Sept. 1972), 948-960.

17. Forsythe, G.E., and W.R. Wasow, Finite Difference Methods for Partial Differential Equations, Wiley, New York, NY, 1960.

18. Gecsei, J., "Interconnection networks from three-state cells," IEEE Transactions on Computers, C-26, 8 (Aug. 1977), 705-711.

19. Gentleman, W.M., "Some complexity results for matrix computations on parallel processors," Journal of the ACM, Vol. 25, 1 (Jan. 1978), 112-115.

20. Glushokov, V.M., M.B. Ignatyev, V.A. Myasnikov, and Torgashev, V.A., "Recursive machines and computing technology," Information Processing 74, J.L. Rosenfeld (Ed.), North-Holland, NY, 1974, 65-70.

21. Gostelow, K.P., and R.E. Thomas, "A view of dataflow," AFIPS NCC Proc., Vol. 48 (June 1979), 629-636.

22. _______, "Performance of a simulated dataflow computer", IEEE Transactions on Computers, Vol. C-29, 10 (Oct. 1980), 905-919.

23. Johnson, D., et al., "Automatic partitioning of programs in multiprocessor systems," COMPCON Spring 80 Proc., Feb. 1980, 175-178.

24. Knuth, D.E., The Art of Computer Programming Vol. 3, Addison-Wesley, Reading, MA, 1973.

25. Kuck, D.J., and A.H. Sameh, "Parallel computation of eigenvalues of real matrices," Information Processing 71, North-Holland, NY, 1972, 1266-1267.

26. Lawrie, D.H., "Access and alignment of data in an array processor," IEEE Transactions on Computers, C-24, 12 (Dec. 1975), 1145-1155.

27. McCarthy, J., "Recursive functions of symbolic expressions and their computation by machine, Part I", CACM, Vol. 3, 4 (Apr. 1960), 184-195.

28. Mead, C., and L. Conway, Introduction to VLSI Systems, Addison-Wesley, Reading, MA, 1980, Chap. 8.

29. Nassimi, D., and S. Sahni, "Bitonic sort on a mesh-connected parallel computer," IEEE Transactions on Computers, C-27, 1 (Jan. 1979), 2-7.

30. _______, "An optimal routing algorithm for mesh-connected parallel computers," Journal of the ACM, Vol. 27, 1 (Jan. 1980), 6-29.

31. Pease, M.C., "The indirect binary n-cube microprocessor array", IEEE Transactions on Computers, C-26, 5 (May 1977), 458-473.

32. Preparata, F.P., and J. Vuillemin, "The cube-connected-cycles: a versatile network for parallel computation," Proc. 20th Foundations of Comp. Science, Oct. 1979, 140-147.

33. Rumbaugh, J.E., "A dataflow multiprocessor," IEEE Transactions on Computers, C-26, 2 (Feb. 1977), 138-146.

34. Siegel, H.J., "A model of SIMD machines and a comparison of various interconnection networks," IEEE Transactions on Computers, C-28, 12 (Dec. 1979), 907-917.

35. Squire, J.S., and S.M. Palais, "Programming and design considerations of a highly parallel computer," AFIPS SJCC Proc., Vol. 23 (1963), 395-400.

36. Stone, H.S., "Parallel processing with the perfect shuffle", IEEE Transactions on Computers, C-20, 2 (Feb. 1971), 153-161.

37. Sullivan, H., and T.R. Bashkow, "A large scale, homogeneous, fully distributed parallel machine, I," Proc. Fourth Symposium on Computer Architecture, 1977, 105-117.

38. Sullivan, H., T.R. Bashkow, D. Klappholz, and L. Cohn, "CHoPP: interim status report 1977," Dept. of Electrical Engineering and Comp. Science, Columbia University, N.Y., NY, 1977.

39. Thompson, C.D., and H.T. Kung, "Sorting on a mesh-connected parallel computer," CACM, Vol. 20, 4 (April 1977), 263-271.

40. Treleaven, P.C., "Exploiting program concurrency in computing systems," Computer, Jan. 1979, 42-50.

41. Tripathi, A.R., and G.J. Lipovski, "Packet switching in banyan networks," Proc. Sixth Symposium on Computer Architecture, 1979, 160-167.

42. Watson, I., and J. Gurd, "A prototype dataflow computer with token labelling," AFIPS NCC Proc., Vol. 48 (June 1979), 623-628.

43. Wittie, L.D., "Efficient message routing in mega-micro-computer networks," Proc. Third Symposium on Computer Architecture, 1976, 136-140.

44. Wu, C., and T. Feng, "Routing techniques for a class of multistage interconnection networks," Proc. International Conf. on Parallel Processing, 1978, 197-205.

45. Wulf, W.A., and C.G. Bell, "C.mmp--A multi-mini-processor," AFIPS FJCC Proc., 1972, 765-777.