UC Irvine
ICS Technical Reports

Title
Performance of a dataflow computer (revised)

Permalink
https://escholarship.org/uc/item/3sc0571v

Authors
Gostelow, Kim P.
Thomas, Robert E.

Publication Date
1979-10-01

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Performance of a Dataflow Computer*

by

Kim P. Gostelow **
Robert E. Thomas

Technical Report #127a

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

October 1979

*This work was supported by NSF Grant MCS76-12460: The UCI Dataflow Architecture Project.

**Authors appear in alphabetical order.

Copyright © 1979, by Kim P. Gostelow and Robert E. Thomas.

Abstract

Our goal is to devise a computer comprising large numbers of cooperating processors (LSI). In doing so we reject the sequential and memory cell semantics of the von Neumann model, and instead adopt the asynchronous and functional semantics of dataflow. We briefly describe the high-level dataflow programming language Id, as well as an initial design for a dataflow machine and the results of detailed, deterministic simulation experiments on a part of that machine. For example, we show that a dataflow machine can automatically unfold the nested loops of n-by-n matrix multiply to reduce its time complexity from $O(n^3)$ to $O(n)$ so long as sufficient processors and communication capacity is available. Similarly, quicksort executes with average $O(n)$ time demanding $O(n)$ processors. Also discussed are the use of processor and communication time complexity analysis and "flow analysis", as aids in understanding the behavior of the machine.

Index terms: dataflow, multiprocessor architecture, large-scale integration, asynchronous execution, parallel computer, distributed computer, concurrency, functionality, locality

1.0 INTRODUCTION

The ability of LSI technology to inexpensively produce large numbers of identical, small, yet complex devices should make possible a general-purpose computer comprising hundreds, perhaps thousands of asynchronously operating processors. Within such a machine each processor accepts and performs a small task generated by the program, produces partial results, and sends these results on to other processors in the system. Many processors thus cooperate, asynchronously, to complete the overall computation. A natural consequence of such behavior should be decreased time for problem solution as new processor modules are added to the machine. This paper describes results of simulation experiments on an initial design for such a machine based on the principle of dataflow.

1.1 Concurrency and the von Neumann Model

Several computers have been devised in attempts to synthesize a single large machine from a collection of smaller processors, e.g., Illiac IV [10], Cm* [16], and C.mmp [39]. However, multiprocessor machines have not yet achieved the ease of programming and level of performance sought. For example, the programmer should not be concerned with how a program is partitioned into concurrently executable pieces nor how these pieces coordinate; nor should the programmer have to consider the number or physical arrangement of the processors comprising the system. Instead, a program should automatically break apart into small pieces that are executed asynchronously with minimum interference from one another. Several

researchers [4,8,14,18] have concluded that this can be achieved only with significant changes to the widely accepted "von Neumann model".

For the machine we have in mind, two particularly troublesome attributes of the von Neumann model are

1. (centralized) sequential control
2. (shared) memory cells.

Sequential, one-instruction-at-a-time control is inappropriate because it prohibits the asynchronous behavior and distributed control we consider essential. It also burdens the programmer with the need to explicitly specify (or to employ an analyzer to determine) exactly where concurrency is to occur. The second attribute, the memory cell, presents a more subtle difficulty. A cell that is shared among several asynchronous program modules often requires complex synchronization controls to ensure its orderly use. Such controls are difficult to design into a highly distributed machine, may be costly in execution time, and are tedious for programmers to use.

The semantics of nearly every programming language so far devised[1] is closely related to the von Neumann model, a model we claim to be contrary to the nature of asynchronous multiprocessing. In contrast, we see <u>dataflow</u> as a more appropriate (though not necessarily the only) basis for general-purpose, asynchronous,

---

[1] Some exceptions are VAL [1], Id [6], LUCID [7], FFP [8], LAPSE [17], ISWIM [25], pure LISP [26], LAU [32] and SASL [37].

computation.

1.2 The Dataflow Model

The basic principles of dataflow semantics have been in existence for some time [9,14,21,22,29]:

1. A dataflow operation executes when and only when all required operands become available (asynchrony).

2. A dataflow operation is purely functional and produces no side-effects (functionality; no memory cells).

Operationally, these semantics are implied by the following model: A dataflow program graph (Figure 1.1) is a directed graph where each node (box or fork) is an operator with lines connecting an output port of one operator to an input port of another, provided that no two outputs are connected to the same input. Operators and lines may be labeled (e.g., operator s and line a in Figure 1.1). Values are carried by tokens which move along the lines between operators. Execution of a program graph is "data driven" in that an operator executes by absorbing exactly one token from each input, computing results, and producing exactly one result token for each active output. The function performed by each operator is indicated in the box; a fork performs the identity function and serves only to replicate values so that a single result may be sent (asynchronously) to many different operators. The requirement that no two outputs can be connected to a single input is the single-assignment rule, the

purpose of which is to exclude races. Computation in a dataflow program graph is determinate if the operators are determinate [3,27].

By adopting dataflow semantics in place of the von Neumann principles enumerated earlier, it is possible to realize the asynchronous execution of programs without the need for special parallel programming constructs (e.g., parbegin-parend, fork-join) or parallel program analysis of any kind. The remainder of this paper concerns a particular dataflow language, method of execution (Section 2), and architecture for a dataflow machine (Section 3). The results of simulation experiments (Section 4) demonstrate the degree to which we might actually realize that asynchrony.

Other dataflow machines have been proposed [13,15,20,23,30,32,38], and some prototypes [13,20,32] have been built. Our work, however, differs from each of the above in at least one of the following three important respects: First, the underlying interpretive mechanism is more asynchronous [3]; second, our high-level dataflow programming language Id (for Irvine dataflow) [5,6], is more complete than other dataflow languages and has been used to describe distributed database systems [5] and a basic operating system as well as the smaller kinds of problems we will discuss here. Third, there are important differences in goals. Our goal is not just to devise a machine that can execute programs rapidly. We are also interested in how dataflow can help in solving general system problems. For example, user protection [12] and exception handling facilities [28] are also being integrated into the design.

2.0 DATAFLOW

Programs for our system are written in Id. An example of an Id program is the expression

mmt(a,transpose(b,m,n),ℓ,m,n)

that utilizes the two Id procedures shown in Figure 2.1 to multiply an ℓ-by-m matrix a by an m-by-n matrix b. Id programs are not executed directly but are first compiled into dataflow program graphs which are then executed on the machine. In this paper we do not discuss the compilation or execution of program graphs.[2] Rather the discussion and analyses that follow will be from the Id programmer's point of view. The goal of the present section is to acquaint the reader with Id and its execution behavior. We ask the reader's indulgence in our claim that the principles which make matrix multiply and the various test programs utilized in Section 4 exhibit their asynchrony extend to other programs as well.

2.1 Values in Id

There are two classes of values in Id[3]: elementary and

---

[2] The interested reader may find the details of compilation in [4,6,19] and an introduction to the execution of dataflow graphs in [36].

---

[3] A third kind of value called a "stream" [6] is also defined but is not discussed here.

structured. An elementary value is an integer, real, boolean, string, etc. and needs no further discussion. A structured value, or structure, is a tree where the branches from any given node are uniquely labeled with a selector (or subscript). Both x and y in Figure 2.2a are structures; Figure 2.2b shows a more complex structure representing a 2×3 matrix in row major order. The empty structure is denoted Λ.

Two functions on structures are select and append. Let the values carried by tokens on lines x and y be the structures x and y, respectively, from Figure 2.2a. Also, let the values carried by tokens on lines i, j, and z be the respective values i, j, and z. Then select(x,i), written x[i] in Id, outputs the value x_i if it exists, otherwise an error value is produced. Somewhat more complex is append(x,j,z) which creates a new structure value identical to x except that selector j is made to refer to value z. We emphasize that the creation of this new structure by append does not modify the value on line x; rather, a new (logical) copy x' of x is first made, then the value z is associated with selector j in x', and finally the value x' is produced as the result. Thus append(append(Λ,1,x),2,y) gives the structured value γ shown in Figure 2.2b.

2.2 The Matrix Multiply Program

In this subsection we consider in some detail the execution of the previously introduced program call

```
mmt(a,transpose(b,m,n),ℓ,m,n)
```

for multiplying the two matrices a and b of dimensionality ℓ-by-m and m-by-n, respectively.

Procedure transpose (Figure 2.1) comprises two nested loops, an i-loop and a j-loop. Concentrating on the i-loop, we note that a loop in Id is strictly an expression with inputs (the initial values, and any values referenced but not defined within the loop -- i.e., b, m, and n), and outputs (the return values). We represent this by the diagram in Figure 2.3a. The phrase "for i from 1 to n" in the i-loop is a generator that "unfolds" n copies or iterations of the loop body (the statements between do and return) and sends successive values of i to each such iteration. All such interations are independent except when interconnected by a so-called "new variable". In the case of the i-loop the statement "new trans← ..." induces the connection between the successive iterations as shown in Figure 2.3b. In this fashion the result computed by one iteration becomes an input value to the next. The first iteration receives the initial value of trans while the value of new trans from the last iteration becomes the result of the entire i-loop. Each i-loop iteration in Figure 2.3b contains a j-loop, that is, each i-loop iteration produces inputs for and receives a result from an entire j-loop. But all j-loops are independent (i.e., none depends on the variable trans) and so may compute concurrently. In essence, the rows of the transposed matrix are being computed concurrently by the j-loops. These rows are then gathered together by the i-loop to form the matrix in row major order.

The above analysis may be expressed concisely in time complexity terms.[4] The i-loop unfolds in time linearly proportional to n, i.e.,

---

[4] A brief introduction to complexity theory may be found in [11].

in O(n) time. Each j-loop instance created by the i-loop requires O(m) time, but all j-loop instances may execute concurrently (assuming O(n) processors) giving procedure transpose O(m+n) time complexity rather than O(mn) as would be the case on a sequential machine. A similar analysis of procedure mmt reveals its complexity to be $O(\ell+m+n)$ (assuming $O(\ell n)$ processors) rather than $O(\ell mn)$ time as required for a sequential machine. In the case of procedure mmt each j-loop instance unfolds to create a total of ℓn concurrent k-loop instances. The overall complexity of the original expression, mmt(a,transpose(b,m,n),ℓ,m,n) is thus $O(\ell+m+n)$ to multiply two matrices.

The above analysis ignores communication costs and assumes unbounded resources. The purpose of conducting an analysis under such ideal conditions is simply to describe the potential concurrency in a particular program. (Section 4, however, will again analyze the same program but with the inclusion of communication costs and will compare the results of that analysis with simulation results.)

Finally, we note that Id loops and procedures are very similar in that both can give rise to independent and asynchronous computations by creation of instances of nested loops and procedure applications. Because of this similarity, each loop or procedure instance generated during execution is called a <u>logical domain</u>. For example, procedure

transpose comprises n+2 logical domains generated at run-time -- one for the procedure itself, one for the i-loop generated by the procedure, and n for the n j-loops generated by the i-loop. Each logical domain has a unique name, u, generated when the domain is created. Within a loop logical domain each iteration is further qualified with the name u.i ($i \geq 1$) where i is the number of the iteration. (Procedures always have i=1.) Logical domains may also be broken down into smaller operations. For example, the compiler breaks every computation into labeled operators such as +, *, etc. to form the nodes of a dataflow program graph (e.g., Figure 1.1). Each loop or procedure is then encoded by the compiler into a structure α such that the value $\alpha[s]$ is the encoding of the operator labeled s. Thus each instance of execution of an operator, called an _activity_, may be uniquely identified by an _activity name_ of the form $u.i.\alpha.s$ specifying the operator labeled s in procedure or loop α executing in the i_th_ iteration of logical domain u. Activities are central to the operation of the machine described in the following section since large numbers of activities can be created dynamically during dataflow program execution.

3.0 THE ARCHITECTURE

3.1 Principles

Three basic principles have guided the design presented below. The first is concurrency achieved through distribution. This is the most basic behavior we are trying to achieve in activity execution, token transmission, and structure access. However, distribution should be tempered by a second principle, locality, meaning that activities logically close together should be executed physically close together. We have selected the logical domain (Section 2.2) to be the unit of localization. Each logical domain is confined to some small sub-section of the machine since the activities within a logical domain are more likely to communicate with one another than with activities outside that domain. A third principle, redundancy, can affect both concurrency and locality. For example, the memory system may keep multiple copies of the same structure value in disjoint areas of the machine to allow concurrent access to local copies of information.

We wish to emphasize that the design discussed here is intended to help discover how dataflow programs behave, and to test some ideas for exploiting that behavior. It is not intended to be a final design. With that in mind, we mention two important design goals that we feel are more easily met in a dataflow system but which we have not as yet attacked. These include:

- modularity: The machine should be constructed from only a few different component types, regularly interconnected, but internally these components will probably be quite complex (e.g., a processor).

- reliability and fault-tolerance: Components should be pooled, so removal of a failed component may lower speed and capacity but not the ability to complete a computation. New opportunity in this area is evidenced by the use of redundant values in the memory system which may prove useful in case a copy of data is lost through component failure.

3.2 Description of the Machine

3.2.1 Units of Measure -

We have experimented with various machine configurations and component speeds in a detailed, deterministic simulation.[5] The following paragraphs describe the machine in detail according to a standard configuration. Unless otherwise stated for any particular experiment, each parameter assumes its standard configuration value. Time is measured in terms of time units.[6] Physical capacities, such as storage words or queue lengths, have no physical limit in the simulation. Although finite working storage at various points in an actual machine can, if exceeded, lead to deadlock, we feel that work on detailed deadlock avoidance schemes at the architecture level is premature. (Deadlock at the Id program level can be detected at compile-time.)

---

[5] The simulator itself comprises a 4500 line program written in SIMULA and runs on a PDP-10.

---

[6] When necessary, for example, to determine the feasibility of a device operating in x time units, we equate one time unit with 100 nanoseconds. All component speeds used in the experiments are considered feasible, though no detailed design has been done.

3.2.2 A Ring Domain -

A ring domain (Figure 3.1) comprises an interconnection of processing elements (PE), memory controllers (MC), and memory boxes (M). All PEs within a ring domain are connected to a pair of shift-register token buses. The buses are connected at their ends (points A and A' in Figure 3.1) to form a pair of counter-rotating 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. Physically, each token carries a <value, destination activity name> pair, as well as a physical PE destination address (explained later). We assume this plus other control information totals 100 bits per token. The rings shift in unison, so each shift brings two new token slots to the ring interface port of each PE. If the physical destination address on the token at a PE's ring interface port matches the PE's address, the PE removes the token from the ring replacing it with an empty slot. A PE may fill any empty ring slot at its interface port with an output token. For the standard configuration we have assumed a token bus shift requires 4.0 time units, or equivalently a maximum of one token in and one token out of each PE every 2.0 time units.

The basic unit of computation is the activity. When a result token with its (logical) destination activity name is being produced, the PE evaluates an assignment function that maps the logical destination activity name onto a physical PE address. Any two PEs intending to send a token to the same activity must use the same assignment function. Since there is a fixed number of PEs but an unbounded number of activities, more than one activity may be assigned

to the same PE for execution. Thus each PE must accept all tokens that are sent to it and sort those tokens into groups by activity name. When all input tokens for an activity have arrived, the PE may execute that activity. Tokens output by an activity are queued in the PE to await empty token bus slots.

To reduce communication load, structure values are not explicitly carried by tokens but rather are kept in a memory, so a token need only carry a pointer to the structure that it logically transmits. (We emphasize again that this memory system is <u>not</u> seen by the Id programmer.) In the standard configuration, four PEs are connected together and to a memory controller by a one-message-at-a-time <u>local bus</u>. Each memory controller is a fairly sophisticated machine that controls the random-access memory box (assumed to be interleaved arrays of 32-bit words) associated with it. All memory controllers in a ring domain are themselves interconnected by a one-message-at-a-time <u>global bus</u> so that every PE has (indirect) access to any structure value held in the machine. Although the memory of the dataflow machine is distributed over the memory boxes, it is organized into one unified address space to facilitate sharing. For example, say a PE is to execute an activity that performs a select on structure α. The PE must ask the memory system to do the operation by sending a <u>request</u> over the local bus to the memory controller to which that PE is attached, called the <u>local memory controller</u>. If α is available in the local controller's memory box, the controller can carry out the select request on α and return a <u>response</u> to the requesting PE. If α is not local then the local memory controller must forward the select request to the proper <u>distant memory controller</u> for servicing. The

distant controller then returns its response to the local controller. Both the request and response messages traveling between memory controllers move on the global bus. Finally, whether α was local or distant, when the local memory controller has the result it is returned to the PE that initiated the original request.

3.2.3 Inside a PE -

Figure 3.2 shows details of a PE organized as a pipelined processor.[7] Each box in the figure is a unit that performs work on one item at a time drawn from FIFO input queue(s). Logically, tokens enter the PE from the two token bus rings at the top of the figure while new tokens are output to the (same) rings at the bottom. The local memory bus connection is shown at the left.

Functional Operation of a PE

The function of the sorter is to group tokens by activity name. The sorter requires 4.Ø time units to process each token with the aid of an associative table keyed on activity names. When an activity name is presented, the table returns a pointer to the list of tokens already gathered for that activity (kept in a "fast" local scratch pad memory). The token is then added to that list. Each token carries a number specifying the total number of input tokens required to complete an activity. If the newly arrived token completes the activity, an activity item pointing to the list of input tokens is

---

[7] The PE architecture described here was pipelined to simplify the simulator. The degree of concurrency appropriate within a PE has not yet been determined.

created and sent to the code fetch box.[8] Each successive box in the figure will include more information in the activity item until processing is completed.

Upon receipt of an activity item with name u.i.α.s, the code fetch box is responsible for retrieving the operation code α[s]. To speed operation, the code fetch box employs a local cache to hold previously fetched dataflow code. If the needed code is already present in the cache, no time is charged, and the code is immediately attached to the activity item which then moves to the next stage in the PE. If the code is not present in the cache, a code structure select request is placed in the local bus input queue and the activity item is held in the code fetch box until the selected item is returned. (This does not prevent the code fetch box from initiating work on the next activity item in its input queue.) Responses from the memory system to the PE are returned over the local bus. These responses are queued and then serviced in FIFO order by the appropriate box within the PE. A response to a code fetch request contains the activity's operation code and the information necessary for the PE to construct the output tokens' destination activity names. The code fetched is also entered into the PE cache with keys α and s. Note that the order in which activity items leave the code fetch box is not necessarily the order in which they enter.

After code fetch, the activity item moves to one of two boxes.

---

[8] This description of code fetch corresponds to the simulator implementation. Alternatively, code fetch could be initiated as soon as the first token for an activity arrives.

The data fetch box issues memory requests and receives memory responses for structure operations (select and append); the arithmetic/logical unit, or ALU, carries out all dataflow operations not requiring the memory system (such as +, *, etc.). The data fetch box operates similar to the code fetch box, sending requests and receiving responses from the memory, except that there is no cache. The time an activity item remains in the data fetch box is determined solely by the response time of the memory to each request. On the other hand, each ALU operation is fixed at 10 units of time.

After proceeding through either the data fetch or ALU boxes, an activity item (with the result of the particular operation attached to it) moves to the output box. Tokens are manufactured by the output box at a rate sufficient to match the token bus, which in the standard configuration is a maximum of two tokens every 4 units of time. During this time the box must copy the result, assign a destination activity name (Section 2.2)[9], and map the activity name to a physical PE address by evaluating the assignment function. The output box then selects the token ring that gives the shortest path from the present PE and places the token in the appropriate output queue. From there tokens move in FIFO order into empty token slots as they appear at the PE's ring interface port.

The Assignment Function

The assignment of activities to physical PEs is very important.

---

[9]This is true except when a new logical domain is created. In such cases we have assigned this work to the ALU and charge 10 units of time.

A good assignment function promotes concurrency and locality, while a poor one can destroy machine performance. (In Section 4 we demonstrate some results on different assignment functions.)

Concurrency is achieved by distributing the activity workload over the PEs of the ring domain. Locality is promoted by mapping all activities within a single logical domain onto the same physical domain, defined to be the set of PEs directly attached to the same memory controller. Thus a ring domain with 32 PEs and four PEs per memory controller has eight physical domains. When the number of logical domains created exceeds the number of physical domains, several logical domains will be assigned to the same physical domain and compete for PE resources. (The competition is actually at the level of the activities within each logical domain.)

The following is a simple assignment function which promotes both locality and concurrency: The jth logical domain to be created, in time order[10], is assigned to physical domain (j mod q) where q is the number of physical domains (numbered Ø through q-1) in the ring domain. Within a physical domain, activity u.i.α.s is mapped onto PE number (s mod 4) as there are four PEs (numbered Ø through 3) per physical domain. Since PEs select the token ring bus which provides the shortest distance path for each output token, tokens in adjacent physical domains do not intermingle, save for the passing of arguments and returning of results. Figure 3.3 represents the concurrency in

---

[10] This implies the existence of a centralized memory cell or other resource to keep track of the latest value of j. We chose this method for ease of simulator programming though we envision using a distributed method (with similiar effect) in an actual machine.

execution and in token transmission that can be achieved when physical domains are active at the same time.

3.2.4 Inside a Memory Controller -

To discuss the operation of a memory controller, we must first discuss data representation. Program code is a special case of structures so the following covers both data and program representation.

Representation of Structures in Memory

Each level of a structured value ν may be represented in the system in either vector or 2-3 tree form, each of which implies its own space requirements and time complexity for select and append operations. Table I gives these requirements for a structure with n selectors at a given level. The contiguous vector representation allows for quick access and is essentially the technique used for one-dimensional arrays in FORTRAN, ALGOL, and other von Neumann languages. Select is straightforward and requires only constant time. However, since dataflow values are never modified, append in general requires that the vector first be copied[11] and the new value inserted at the correct position to create the result. An important exception occurs when there is only one pointer referencing ν (which can be determined using a reference count or reference weight [35] scheme).

---

[11] Only the "top level" of the original structure need be copied in an append operation. Sub-structures, if any, need not be physically copied as shown in [14,19].

Here the input value v will no longer be used after the append anyway, so v can be updated in place to give the result directly. In this case append requires only constant time (assuming sufficient contiguous space is available for the new value if a new selector is being appended). An alternative representation for structures is a 2-3 tree which requires O(log n) time for append and select as discussed in [19,33].

The standard configuration uses vector representation for program code. Vector representation is also used for each input data structure until the first append, at which point it is automatically converted to 2-3 tree format. The structure then remains in 2-3 tree form in anticipation of further appends. However, in many algorithms the automatic conversion on append is best overridden to reduce the time complexity of subsequent structure accesses. For example, in procedure mmt (Figure 2.1) only one pointer to a given row of the result matrix exists at any given time during its formation. Thus if enough contiguous space is allocated[12] when computation of a row begins, then each append can be done in constant time resulting in a matrix in vector format. Hence, ignoring conflicts, memory access need not increase the time complexity of matrix multiply.

The Memory Cache System

To increase concurrency and locality we have devised a cache system (independent of the PE cache) wherein each memory controller acts as a cache to the rest of the memory system. Assume that a

---

[12] This is analogous to dynamic allocation of vectors in ALGOL and could be specified by the programmer or perhaps by compiler analysis.

memory controller receives a select(α,i) request from a PE and that α is not local. The local controller then requests the distant controller to send a copy of the top level of structure α to the local controller, rather than have the distant controller do the select. When the copy is received by the local controller, it makes an entry with key α in an associative table which points to where the copy α' of α is locally held. Any subsequent operations on α can then be carried out on α' independent of those carried out on α. (Recall that dataflow structures are never modified.) Also, the internal representations of the structure at α and at α' need not be the same.

Functional Operation of a Memory Controller

Figure 3.4 shows a detailed view of a memory controller and associated memory box. The transmission of an entire level of a structure resulting from a copy request can require significant global bus time. For this reason, and to make sender-receiver coordination easier, there is a separate copy processor and memory port provided for copy data transmissions. Thus when a distant memory controller's request processor services a copy request, it gives the request to the copy processor which then transmits the structure over the global bus to the copy processor at the local controller. In the standard configuration, the sending copy processor transmits only the leaves of 2-3 trees which are then converted back to 2-3 format by the receiving copy processor. Structures in vector format are transmitted and stored without conversion.

The time required by a memory controller to process a request depends upon several factors including data representation, memory

speed, and memory controller speed. In general, the simulator charges 6 units of overhead time for each request message, plus the time to do the actual operation. Select and append require the number of operations specified in Table I multiplied by 1.5 time units per word. Overhead for a minimal amount of storage management activity such as reference weight manipulation is included in this figure. More extensive "storage reclamation" is assumed to be accomplished during memory controller "idle periods" which typically range from 30 to 60% with the optimum number of processors. Each select and append request message is four words and requires 0.4 units of transmission time per word. Structure copy bus transmission time is as specified in Table I multiplied by 0.4 time units per word.

4.0 MACHINE PERFORMANCE

A full scale machine will contain a large number of ring domains (Section 3) although we have limited this initial study to just one. Our intent was to answer some simple questions: Does the asynchrony of the unfolding mechanism actually provide for increased speed of execution as more processors are added to the pool? And to what extent do our working hypotheses -- the anticipated relationships between locality, distribution, concurrency, and redundancy -- actually operate?

The experiments involved running dataflow programs on a simulated machine which monitored the programs' executions. All programs[13] were

---

[13] The programs used were matrix multiply (procedure mmt), optimal binary search tree generation, Gauss-Seidel linear equation solver, Gaussian elimination, recursive quicksort, and fast Fourier transform (both an iterative and a recursive version).

written in Id and then machine compiled and loaded into the simulator for execution. Many experiments were repeated on more than one type of dataflow program, though due to cost not all experiments could be repeated on all programs. Also, only the mmt procedure part (Figure 2.1) of matrix multiply was used in the matrix multiply experiments presented here. (The presence of procedure transpose in matrix multiply has no effect on the overall time complexity since transpose requires only O(m+n) time. This prediction was confirmed by test cases as was the physical distribution of the transposed matrix to ensure satisfaction of procedure mmt's input assumptions.) Finally, we have concentrated in this paper on the matrix multiply program for two reasons: it places a heavy asynchronous load on the machine, and it is easy to analyze.

4.1 Speedup Experiments

The graphs of Figure 4.1 are speedup graphs. Each curve plots the execution time (y-axis) of a particular dataflow program against the number of PEs (x-axis) in the standard configuration. (Although experiments were conducted by varying the number of PEs in a ring domain, we anticipate an actual machine would have fixed-size ring domains.) The percentage efficiency of ALU utilization is indicated on the graphs at each experimental point, where

$$\text{efficiency} = \frac{\text{cummulative actual busy time of all ALUs}}{\text{cummulative potential busy time of all ALUs}} * 100$$

i.e., efficiency is directly related to the mean ALU duty cycle.

Figure 4.1a shows, for example, how procedure mmt performs on two 7×7 matrices when run on machines with PE resources varying from 1 to 120 PEs. This curve demonstrates two important points:

1. For even this small (though highly asynchronous) computation, a significant number of PE resources can be usefully employed. For example, execution time was reduced by a factor of almost 14 (43468 time units for a system with 1 PE versus 3123 units for 60 PEs).

2. Small increments in the size of the domain (number of PEs) are effective; in fact, they are very effective when scarce, while increments beyond the optimum result in very slow performance fall-off.

Such speedup graphs are produced without changing the dataflow programs -- only a single parameter is altered to tell the simulator the number of PEs available. In the Introduction we noted such behavior would be desirable because it demonstrates independence of physical processor configuration (both size and shape) from the programs executed. Moreover, even the small degree of performance fall-off that was present can be blamed on an unsophisticated assignment function that forces computations to be distributed over the ring domain even when such distribution is inappropriate. The result is under-utilized PEs and increased communication delays since there is an increase in mean token distance and a decrease in the probability that any given structure resides in a memory local to the PE needing it.

The other curves in Figure 4.1 show similar, if not as dramatic, results for other programs except for the iterative fast Fourier transform (FFT), Figure 4.1e, which did not do well at all. Although the behavior of iterative FFT is not completely understood, it appears

to be a combination of several factors. These factors include scheduling anomalies and unwanted synchronization imposed by the append operation in constructing structured values.[14] Recursive FFT performs well because it uses a "divide-and-conquer" method, and because the size of the data structures progressively decreases at each recursive call. Under dataflow, pairs of divide-and-conquer recursive calls are done asynchronously (i.e., a new pair of logical domains is created). This means that recursion is often faster than looping.

4.2 Complexity Experiments

Recall from the analysis of matrix multiply in Section 2.2 that the processor time complexity (ignoring communication complexity) was O(r) for procedure mmt on two r-by-r matrices. To experimentally determine execution time complexity, we performed the speedup experiment for all problem sizes from r=2 through r=8 . The results appear in Figure 4.2 where each point is the minimum execution time for each problem size; the number of PEs used to achieve that minimum appears adjacent to each point. The bottom curve represents the standard configuration and shows that execution time was indeed O(r). In addition, when processor efficiency (ALU duty cycle) is accounted for, processor utilization is $O(r^2)$ as predicted by the analysis. To explain the other curves in Figure 4.2 we must first discuss time

---

[14] In some programs, this synchronization can be removed by using a new pipelined or stream append operation. Although we have not tested the method, we expect it to significantly improve performance.

complexity a bit more.

Processors are just one of the resources being demanded by a program. We must also consider other resources -- memory controllers, the global memory bus, and the token bus -- and their effect on execution time complexity. Consider the memory controllers. Exactly one copy of each input matrix exists initially, the rows of which are distributed over the available memory boxes. (Matrix multiply produces a result in this configuration.) The analysis in Section 2 assumed that the number of PEs available, and thus memory controllers, is $O(r^2)$ while the number of input rows is $O(r)$. Thus there are plenty of controllers and memory boxes. However, procedure mmt requires access to all elements of each row r times since each row participates in r inner products. So each of as many as 2r memory controllers (the number of rows) sees $O(r^2)$ accesses, giving a memory controller time complexity of $O(r^2)$. (Note that it is irrelevant to the memory controller complexity analysis whether a row is copied from a distant to a local memory controller or not, even though such copying is often highly relevant to actual elapsed time as will be shown in Section 4.4.)

The global memory bus experiences an even heavier demand than the memory controllers. The $O(r^2)$ accesses to each of the possible 2r memory controllers must traverse the single global bus. Thus the global memory bus time complexity is $O(r^3)$. (This might not be the case were there more than one ring domain.)

Finally we consider the bi-directional shift register token bus with its intensive intra-logical domain communication. To determine

the token bus time complexity, note that both the number of logical domains and the physical domains to which they are assigned is $O(r^2)$, since for these experiments it is assumed that $O(r^2)$ PEs are available. Since the intra-logical domain communication on the bus is concurrent (Figure 3.3), then these domains are non-interfering and have essentially constant token communication time within each domain. (This agrees with experimental results where the overall mean token communication time was always between 4.0 and 8.0 units when the standard assignment function was used.) However, all r^2 inner product domains (instances of k-loops) were originally produced from a single initial domain (the outer i-loop). This means that a chain of tokens must have passed from this initial domain to each of the r^2 inner product logical domains distributed along the bus. But the length of the bus is directly proportional to the number of processors -- $O(r^2)$. Thus the length of the longest token path from the initial logical domain to the last of the r^2 inner product logical domains is $O(r^2)$ -- the token bus time complexity.

By the above analysis the global memory bus is the limiting resource and constrains performance to $O(r^3)$. Nevertheless, the bottom curve in Figure 4.2 is (almost) linear because the constants in the global memory bus time complexity term do not allow it to become dominant when $r \leq 8$. For $r > 8$ the above analysis predicts that the apparently straight line will eventually become a cubic. Due to constraints in the simulator on the PDP-10 we were not able to go beyond $r=8$; but to verify the expected behavior, we instead unbalanced the machine and lowered the memory system speed by the factors indicated in the other two plots in Figure 4.2 . Lowering

memory system speed increases the constants in the global bus and memory controller time complexity terms, causing the machine to reach more quickly the predicted execution time complexity of $O(r^3)$.

Another example of a time complexity experiment is Figure 4.3a which shows measured execution time for both the recursive and iterative versions of FFT. Processing time complexity analyses for these two programs give O(n) and O(n log n) respectively. These results are borne out by the experimental curves. On a sequential machine both times are O(n log n). Figure 4.3b shows time complexity graphs for a Gaussian elimination algorithm to solve simultaneous linear equations. The time complexity for a single processor is $O(n^3)$ while the processing time complexity for the dataflow system is $O(n^2)$ as demonstrated by the experimental results. Similarly, recursive quicksort has an average time complexity of O(n) on the dataflow machine as shown in Figure 4.3c ; on a sequential machine quicksort has O(n log n) average time complexity.

Time complexity analysis is a useful tool in understanding dataflow machine behavior and in aiding selection among design alternatives. Overall execution time complexity has been shown to include not only processing complexity but also token bus, memory bus, and memory controller time complexities which together represent a "communication time complexity factor" not explicitly present in algorithmic analysis on standard von Neumann systems. However, it is probable that communication time complexity will be the dominant term in future systems and algorithmic design, a conclusion similar to that reached by Sutherland & Mead in their speculative article [31].

4.3 Flow Analysis

A major difficulty in evaluating a system is devising adequate measures. Section 4.2 has shown that time complexity can be a useful tool for understanding system behavior. The usual measures such as queue lengths, time to execute a program, and the duty cycle of various units are also helpful. However, flow analysis[15] was most useful in determining resource balance and the location of bottlenecks (imbalances).

When flow analysis is applied to the actions related to the execution of activities, it is referred to as activity flow analysis. Let the term item refer to both a token and to an activity item, and let the block diagram of a PE (Figure 3.2) represent a sequence of stations through which items must pass (each queue is also interpreted as a station distinct from the station it serves). Activity flow analysis consists of measuring the mean time spent by all items at each station, and interpreting this time as the time spent at a station by some hypothetical "mean" item. For instance, Table II shows an activity flow analysis for two runs of procedure mmt at r=7 where column (a) is the result for the standard configuration. The mean token time in the sorter queue over all PEs was measured as Ø.81 units. Thus we say that the hypothetical token spent Ø.81 time units waiting in the sorter queue. The other measures are taken in a similiar manner, except that no single activity item passed through both a data fetch box and an ALU box. Here we weight the means and

---

[15] Flow analysis is a generalization of "longest path" analysis discussed in [34].

say the hypothetical mean item spent (d/n)*td of its time in a data fetch box (where d of the n total activities passed through a data fetch box with measured mean time td), and ((n-d)/n)*tALU of its time in an ALU box. For example, in the case of Table IIa 15.6% of the activity items passed through data fetch boxes with mean service time of 21.79 units; this yields a data fetch time of 3.4 units for the hypothetical mean item. The sum of the times for all boxes and queues listed is the <u>cycle time</u> of a hypothetical activity. Table IIb shows a flow analysis for the same program but on a machine differing from the standard configuration only in the speeds of the local and global buses. These analyses pinpoint the resulting system imbalances.

<u>Memory request flow analysis</u> is concerned with the memory system portion of activity flow analysis, i.e., the time a hypothetical mean memory request spends in the memory system until its originating PE receives and processes the corresponding hypothetical response. The fact that not all messages require the global bus or a distant memory controller is accounted for by weighting the measured times by the fraction of requests that did access the global bus and some distant memory controller. Table III presents the memory request flow analyses for the same runs that produced the respective activity flow analyses of Table II. Again the imbalances in resources are immediately evident.

We consider the weighted means given by flow analysis to be more indicative of overall performance for an asynchronous (dataflow) machine than the corresponding raw means. Flow analysis factors out "waiting time" such as the time between the arrival of the first token

of an activity and the last token. Although waiting time is important in considering buffer requirements, it appears otherwise to have little effect on average machine performance. For comparison with a more traditional measure, Table IV shows the duty cycles of the various units for the same runs that produced Tables II and III.

4.4 Locality, Concurrency, Distribution and Redundancy

A primary effect we wish to achieve is concurrency of execution induced by distributing (more or less) independent activities over many processors. But activities should not be distributed indiscriminantly -- program locality should be considered. Locality is evidenced in token and memory communication distances and is largely determined by the assignment function used.

Figure 4.4 shows a speedup curve for procedure mmt with r=7 utilizing four different assignment functions called A, B, C, and D. Functions A, B, and C promote token communication locality (as illustrated in Figure 3.3) by mapping onto physical domain d the jth logical domain that is created according to the formula

$$d = j \ \underline{\text{mod}}\ q$$

where q is the number of physical domains in the machine. This confines all activities in logical domain u to physical domain d, regardless of how large that logical domain might be.

Assignment function A (described previously) maps activity u.i.α.s onto PE p <u>within</u> physical domain d by the formula

$$p = s \underline{\bmod} 4$$

Consider what happens when recursive procedures or nested loops are present in a program. In assignment function A, distinct initiations of the same procedure or loop (which are assigned to the the same physical domain) are assigned identically within that physical domain. The same PE then executes the same operators within those logical domains resulting in very effective use of the PE's code fetch cache.

Assignment function B is used for the standard configuration:

$$p = (s+j) \underline{\bmod} 4$$

and is similar to A except that distinct initiations of the same procedure or loop assigned to the same physical domain do not necessarily have their activities assigned identically within that physical domain. The result is a "wider" distribution of activities, and a lessening in cache effectiveness, i.e., reduced locality. As evidence of locality reduction in the case of procedure mmt with 6Ø PEs, the mean code fetch hit ratio in the PE cache was reduced from Ø.93 for assignment function A to Ø.82 for assignment function B.

A third assignment function C distributes activities within a physical domain to a greater extent than either functions A or B by including the term i in mapping $u.i.\alpha.s$ to PE p within physical domain d :

$$p = (s+j+i) \underline{\bmod} 4$$

The fourth assignment function D is present for the purpose of comparing assignment functions A through C with a function which

distributes activites without regard to physical domain. Function D is

$$p = (s+j) \underline{mod} (q*4)$$

where (q*4) is the number of PEs in a ring domain and the PEs are numbered Ø through q*4-1.

Another view of the comparison among the four assignment functions is offered by the activity flow analyses of Table V taken at the 6Ø PE point from each of curves A-D (Figure 4.4). Note the effect of locality on code fetch, output, and token bus times. If locality were not present, then average token bus time would be half way around that ring bus which gives the shortest path to the destination, or Ø.5 × (Ø.5 × 6Ø) × 4.Ø time units = 6Ø time units. (This demonstrates that even assignment function D promotes considerable locality; this is due to favorable statement numbering by the compiler.) Performance differences for the locality-exploiting assignment functions A-C are slight compared to the clear performance loss of assignment function D. Although reasons for performance differences among A-C are interesting to hypothesize, we have not yet conducted sufficient experiments to explain the differences in detail.

The following experiments concern the structure copy mechanism discussed in Section 3.2.4, whereby a local controller acts as a cache to the other memory controllers. Redundancy of data provides for improved performance potential in two respects. First, copies of data in disjoint memories allow concurrent access where not previously possible. Second, congestion may be reduced because only one request

and response (rather than many) have required the global bus and have suffered the various queueing and processing delays. Figure 4.5 demonstrates how data redundancy affects execution behavior for the case of procedure mmt. Here we have included the three time complexity curves from Figure 4.2 for comparison with a system differing from the standard configuration only in that the structure copying mechanism has been inhibited. That is, all requests for select and append on non-local structures are forwarded to the proper distant controller for execution.

5.0 CONCLUSIONS

Our eventual goal is to design a system that exploits the full potential of LSI technology. To achieve this, we have adopted the semantics of dataflow as the basis for a machine and programming language since it allows us to avoid many of the problems that confront current multiprocessor systems.

This paper has shown the results of experiments on a simulated version of a particular architecture executing programs written in a high-level dataflow language. Our purpose in experimenting with this machine was not to show that it was fast in any absolute sense, but rather to answer some basic questions about dataflow and its feasibility as the basis of a machine. In particular, we demonstrated that the system can generate large numbers of activities, and that the independence of these activities allowed for increased execution speed as additional processors were made available to the system. We verified several expectations concerning locality, distribution, and

redundancy, and their effects on the concurrency achieved in the machine. We also confirmed our analyses of program time complexity and concluded that communication time complexity is at least as important as processor time complexity. In general, we feel complexity analysis is a useful tool for designers of such systems as is flow analysis for uncovering bottlenecks and resource imbalances.

Of course, much work remains to be done. In particular, we plan to revise several aspects of our initial design (e.g., the busing systems) and to extend the machine from one to many ring domains. Also planned is further research into the assignment and scheduling of activities and determination of the proper size or "grain" of an activity, aspects which are certain to have significant impact on machine performance. Another area scheduled for investigation is the incorporation of streams to avoid the unnecessary synchronization of dataflow structures.

In summary, the results appear encouraging. The highly asynchronous behavior we hoped to observe was indeed found in many programs to a degree suggesting that dataflow may be one way to utilize the power apparent in LSI technology, while also giving the programmer a clean and useful semantic basis [5,12].

ACKNOWLEDGEMENTS

We would like to acknowledge that part of the work (reported elsewhere) of Arvind and Wil Plouffe on the design of Id, the base language, and the unfolding mechanism. We also acknowledge the excellent PDP-10 SIMULA system implemented by the Swedish National

Defense Research Institute. We would like to thank Shirley Rasmussen for typing the manuscript, and the UCI Computing Facility for providing computer support.

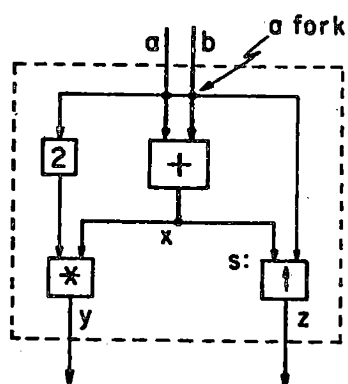


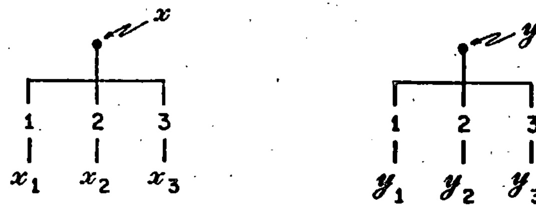
Figure 1.1

An example dataflow graph

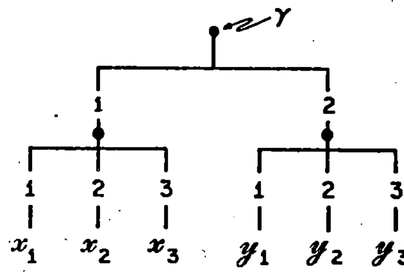
```
procedure transpose (b, m, n)
   (initial trans ← Λ
    for i from 1 to n do
        new trans ← append(trans,i,(
            initial row ← Λ
            for j from 1 to m do
                new row ← append(row,j,b[j,i])
            return row))
    return trans) ;

procedure mmt (a, bt, ℓ, m, n)
   (initial c ← Λ
    for i from 1 to ℓ do
        rowa ← a[i] ;
        new c ←append(c,i,(
            initial rowc ← Λ
            for j from 1 to n do
                colb ← bt[j] ;
                new rowc ← append(rowc,j,(
                    initial innerprod ← Ø
                    for k from 1 to m do
                        new innerprod ← innerprod + rowa[k]*colb[k]
                    return innerprod))
            return rowc))
    return c)
```

Figure 2.1

The call mmt(a, transpose(b,m,n),ℓ,m,n) is the product of the ℓ-by-m matrix a and the m-by-n matrix b.



(a)



(b)

Figure 2.2

Examples of structured values

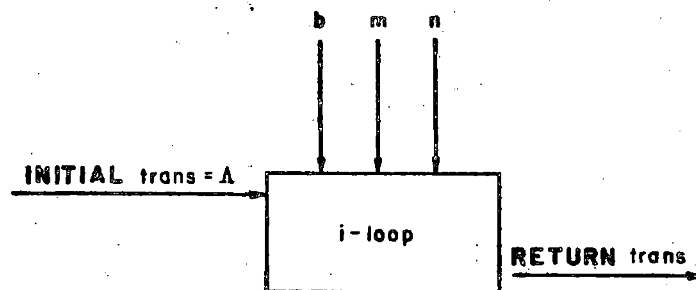


Figure 2.3a

A loop is an expression

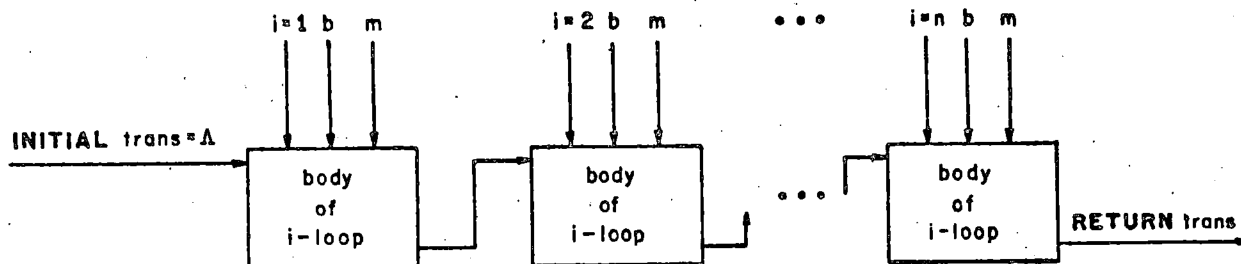


Figure 2.3b

A loop unfolds and produces many iterations, i.e., instances of its body, which might execute concurrently

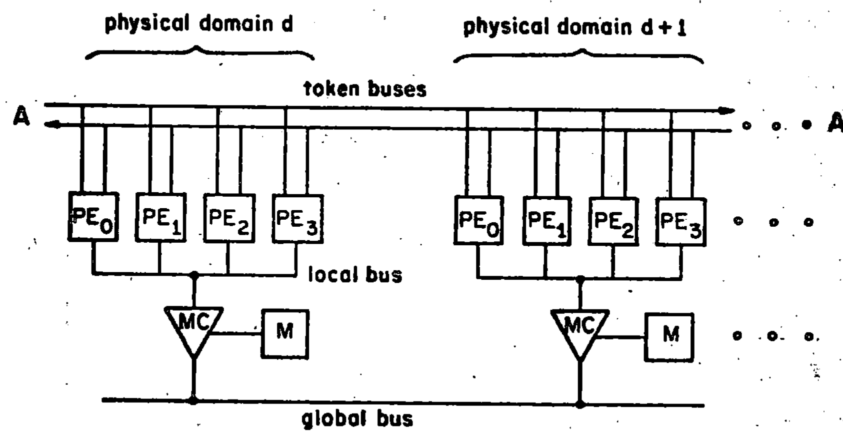


Figure 3.1

A ring domain

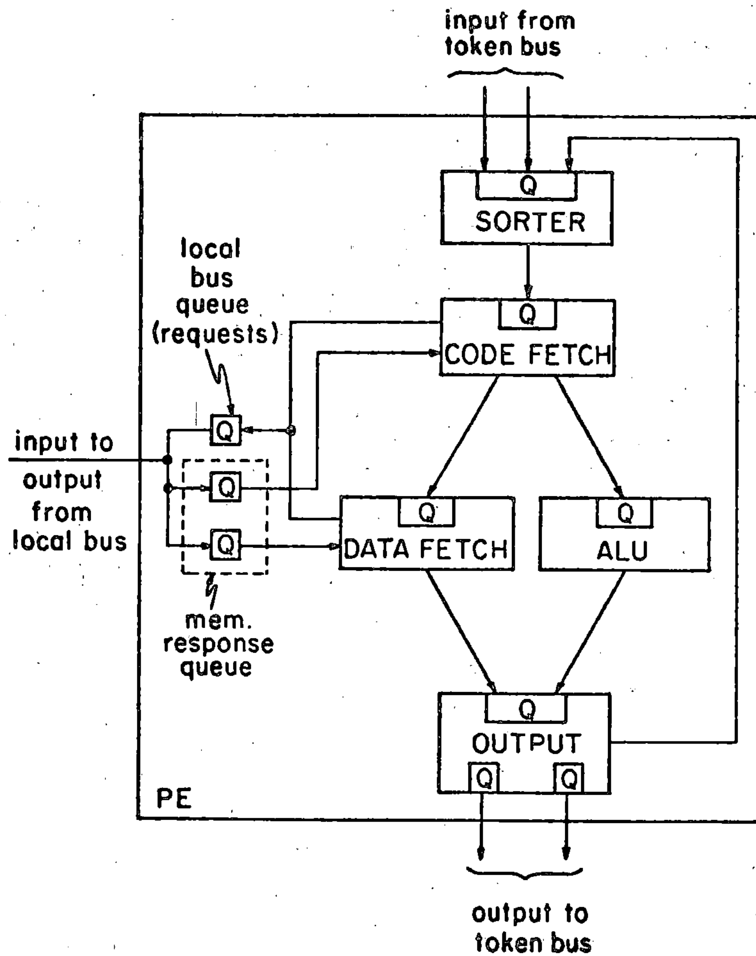


Figure 3.2

A processing element (PE)

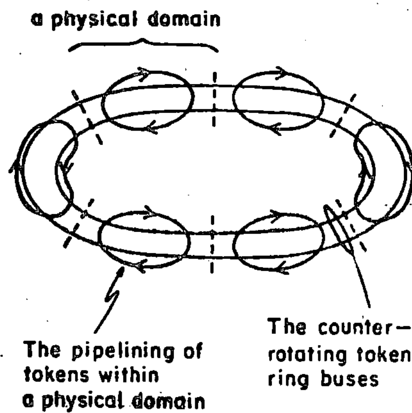


Figure 3.3

Physical domains operating concurrently

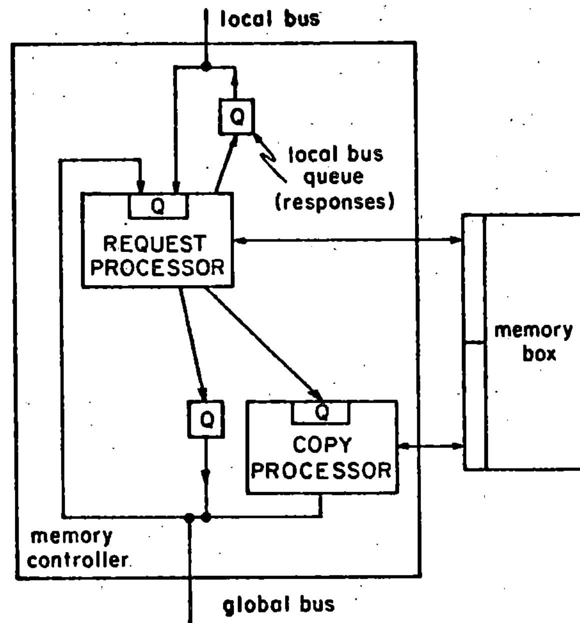


Figure 3.4

A memory controller and
attached memory box

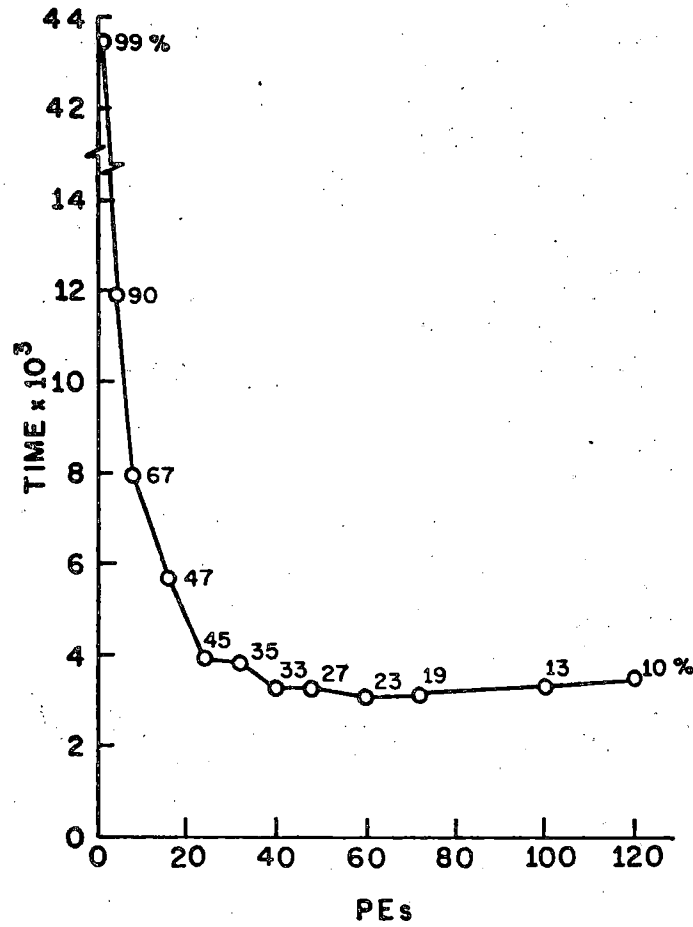


Figure 4.1a

Matrix multiply (7×7)
The efficiency of ALU utilization appears at each experimental point.

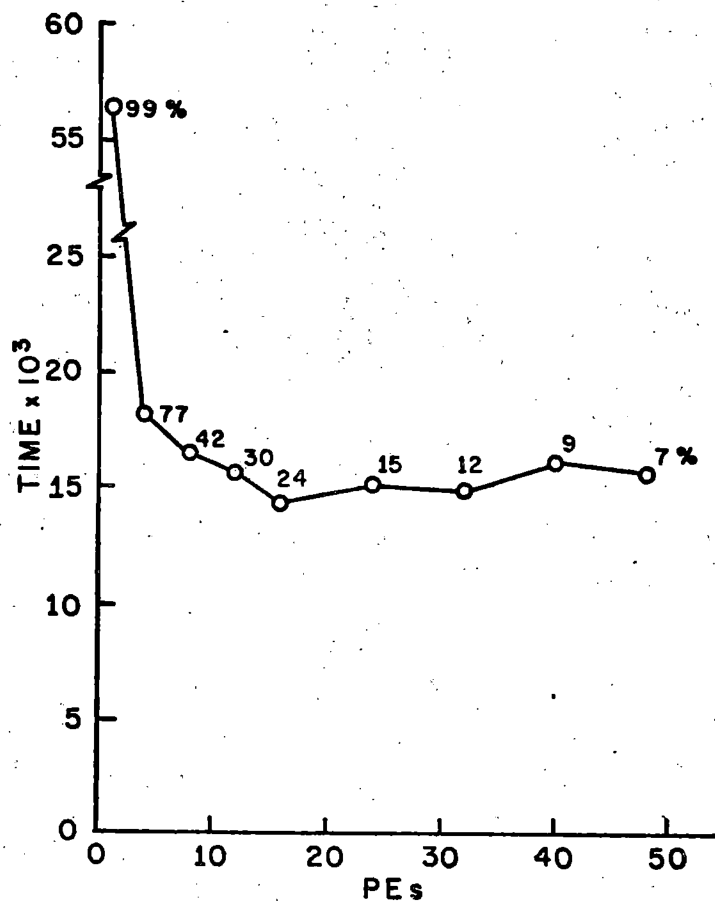


Figure 4.1b

Recursive quicksort (30 items)

The efficiency of ALU utilization appears at each experimental point.

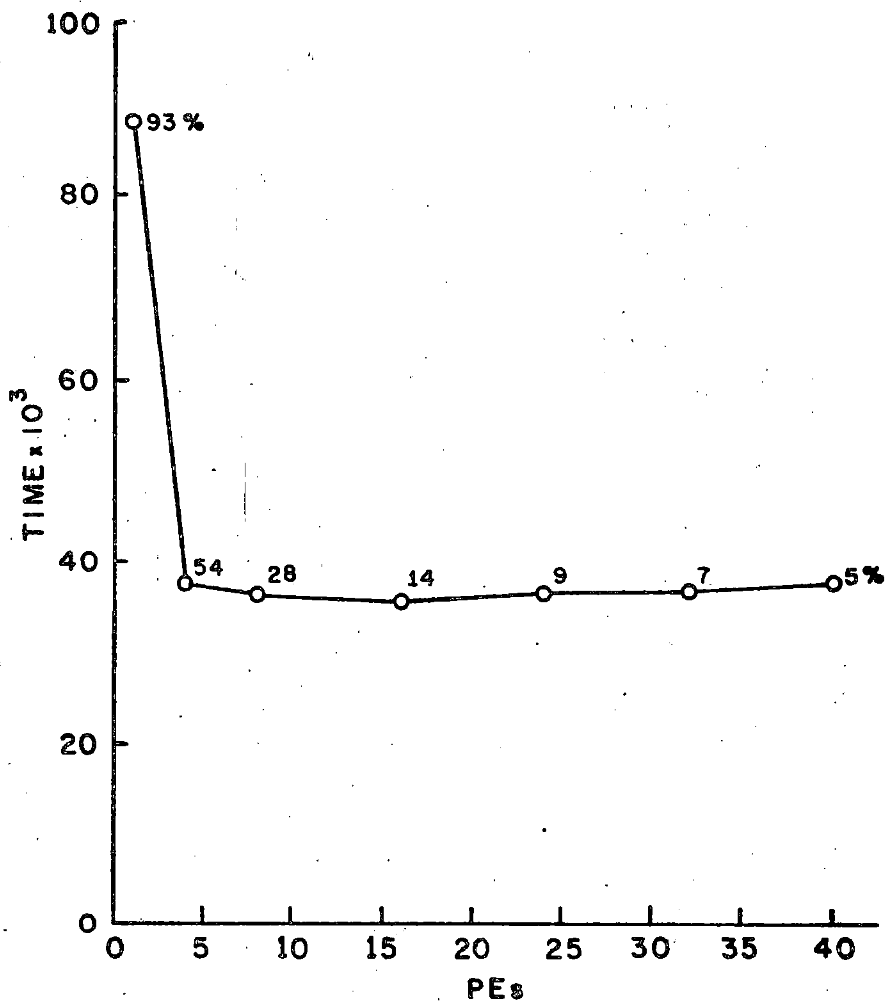


Figure 4.1c
Gaussian elimination (10×10)
The efficiency of ALU utilization appears at each experimental point.

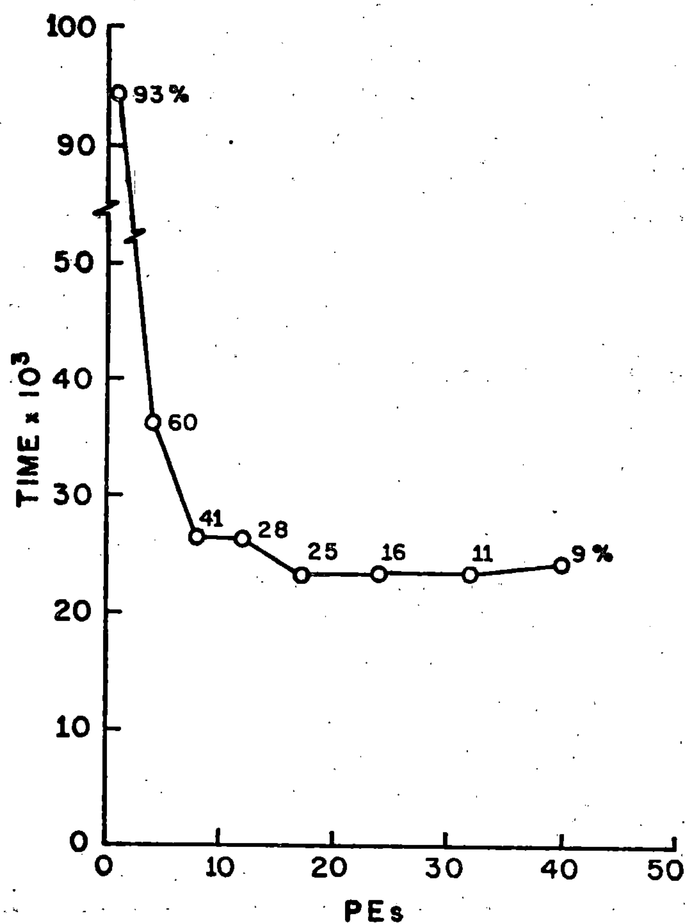


Figure 4.1d

Recursive FFT (32 point)

The efficiency of ALU utilization appears at each experimental point.

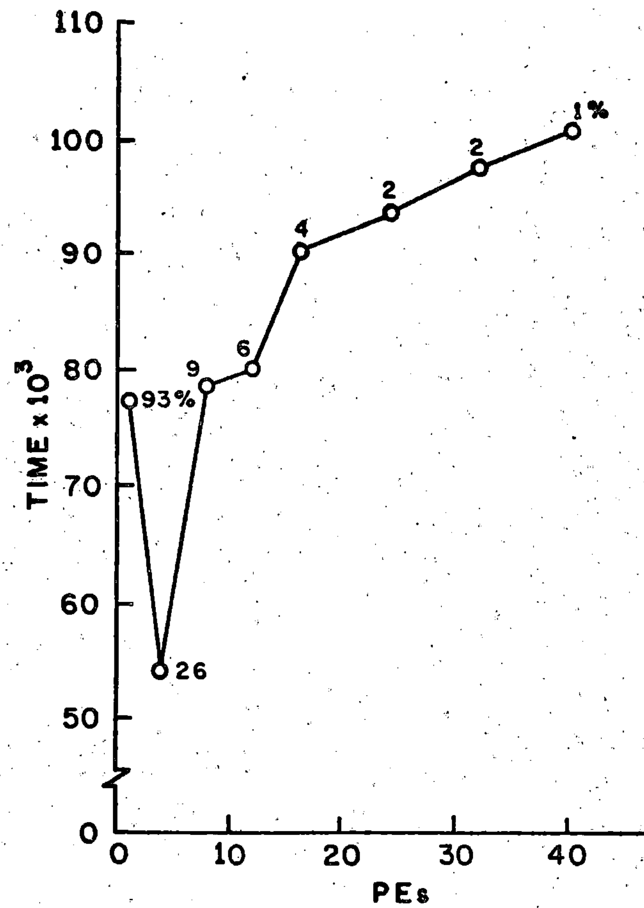


Figure 4.1e

Iterative FFT (32 point)

The efficiency of ALU utilization appears at each experimental point.

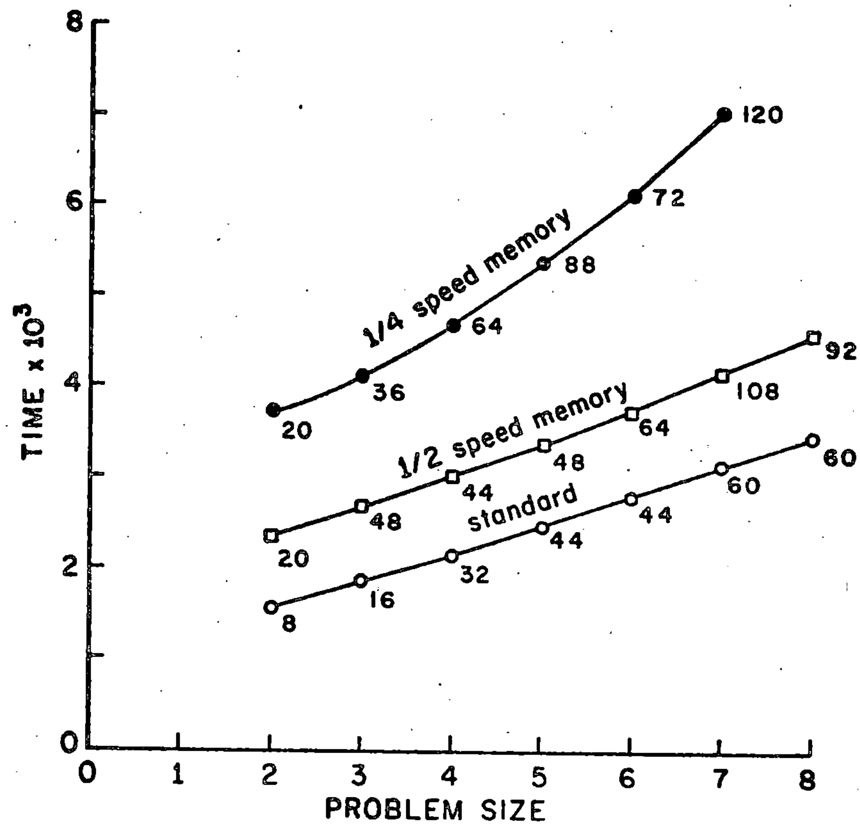


Figure 4.2

Execution time complexity curves for matrix multiply. Each point plots the minimum execution time for a given problem size, where the number of PEs used to achieve that minimum appears adjacent to each point. The family of curves shows that as memory speed is slowed, communication time complexity effects begin to dominate.

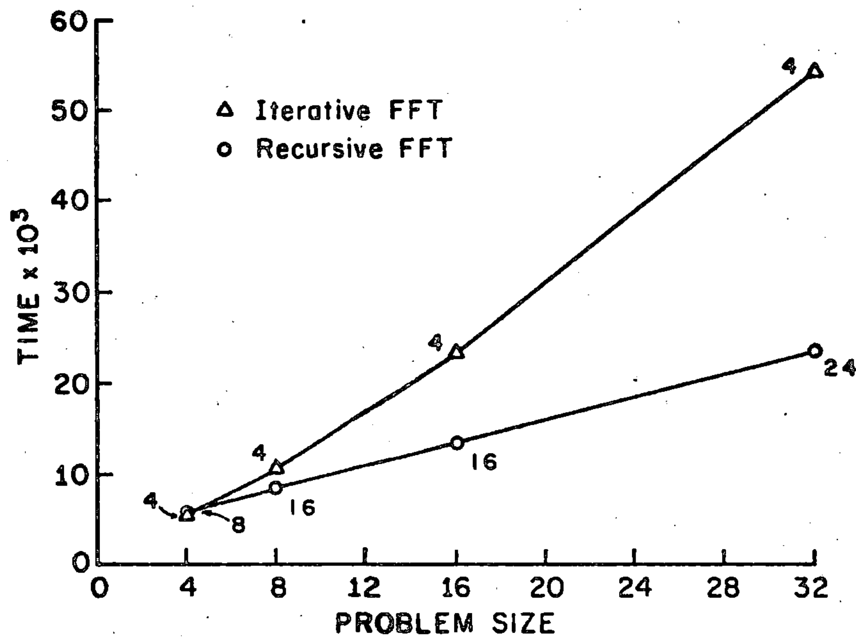


Figure 4.3a

Execution time complexity curves for recursive and iterative FFT: O(n) and O(n log n) with optimal number of PEs, respectively, while both are O(n log n) on a sequential machine. The total number of PEs used appears adjacent to each point.

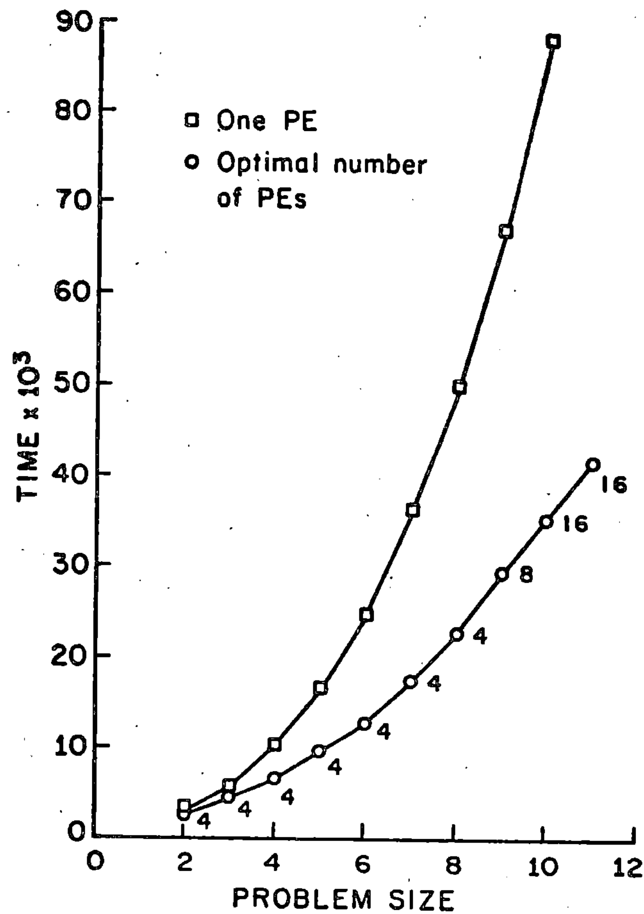


Figure 4.3b

Execution time complexity curves for Gaussian elimination: the curve for one processor is $O(n^3)$ and is similar to what would be achieved on a sequential machine, while the dataflow machine gives $O(n^2)$. The total number of PEs used appears adjacent to each point.

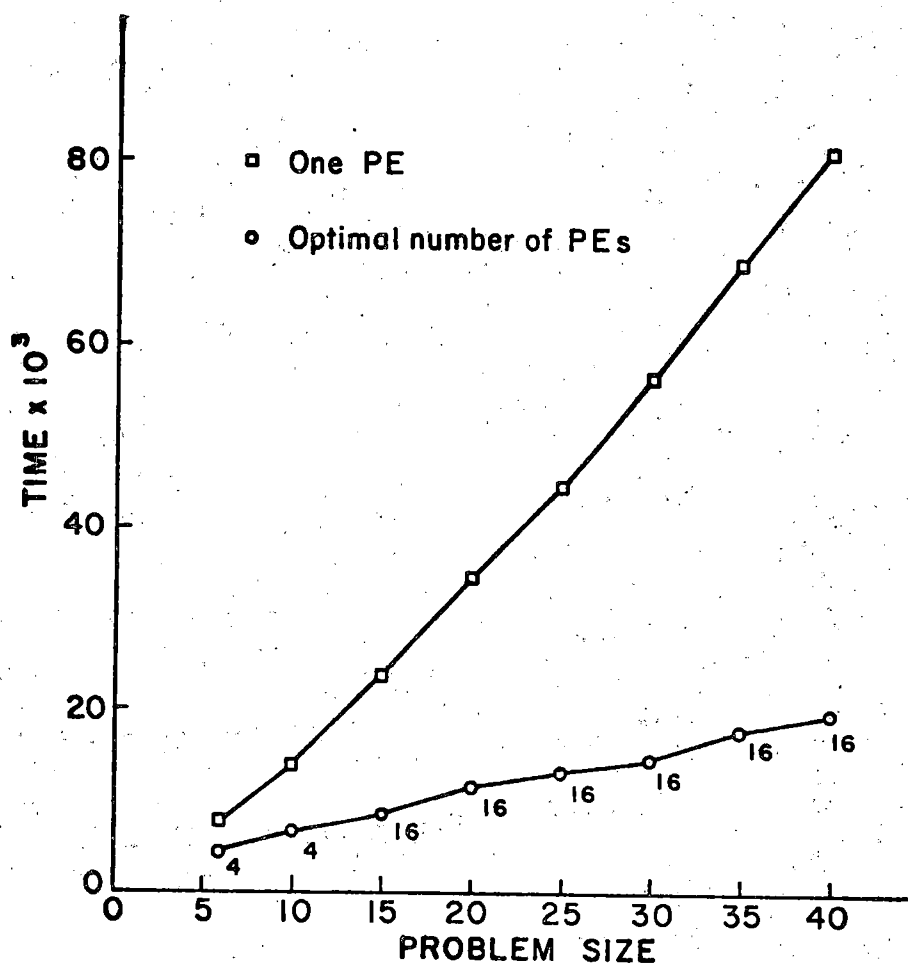


Figure 4.3c

Execution time complexity curve for recursive quicksort: the average behavior is O(n) on the dataflow machine with optimal number of PEs and O(n log n) on a sequential machine. The total number of PEs used appears adjacent to each point.

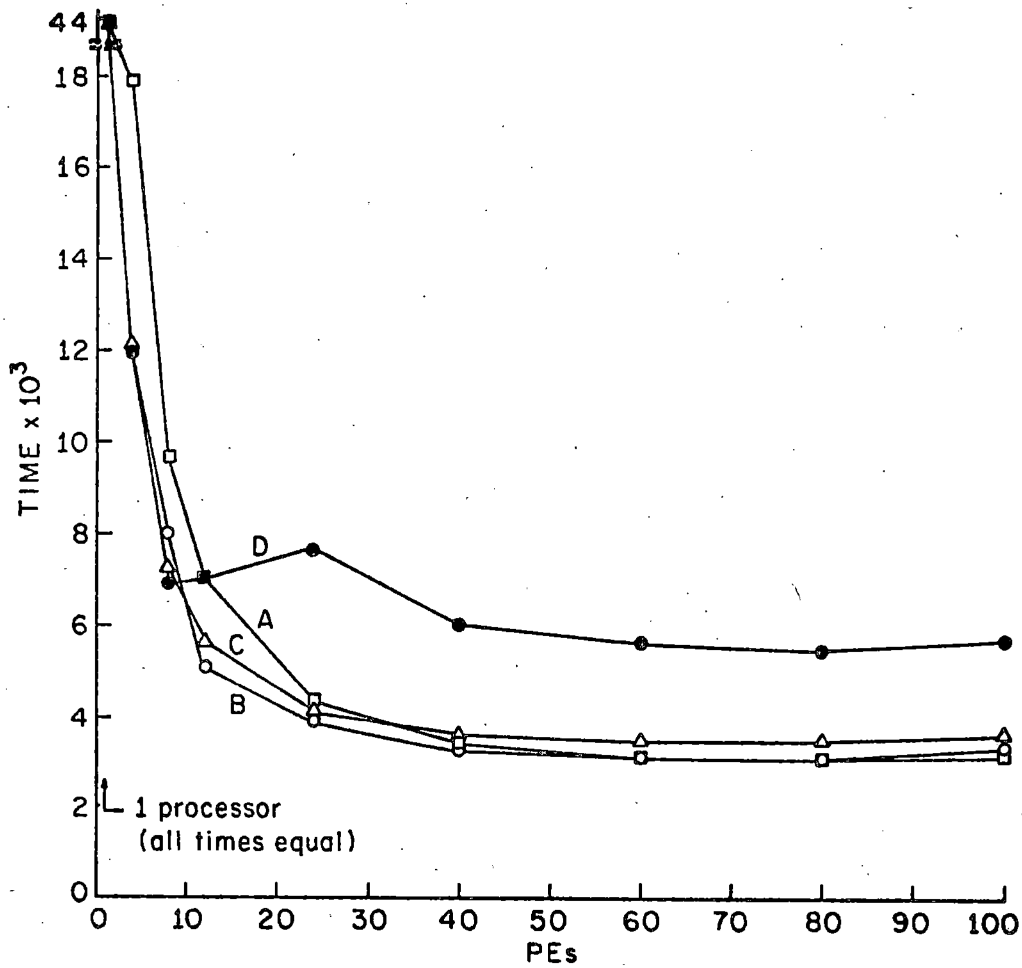


Figure 4.4

Speedup curves for 7×7 matrix multiply under four different assignment functions. Functions A-C encourage locality while D does not.

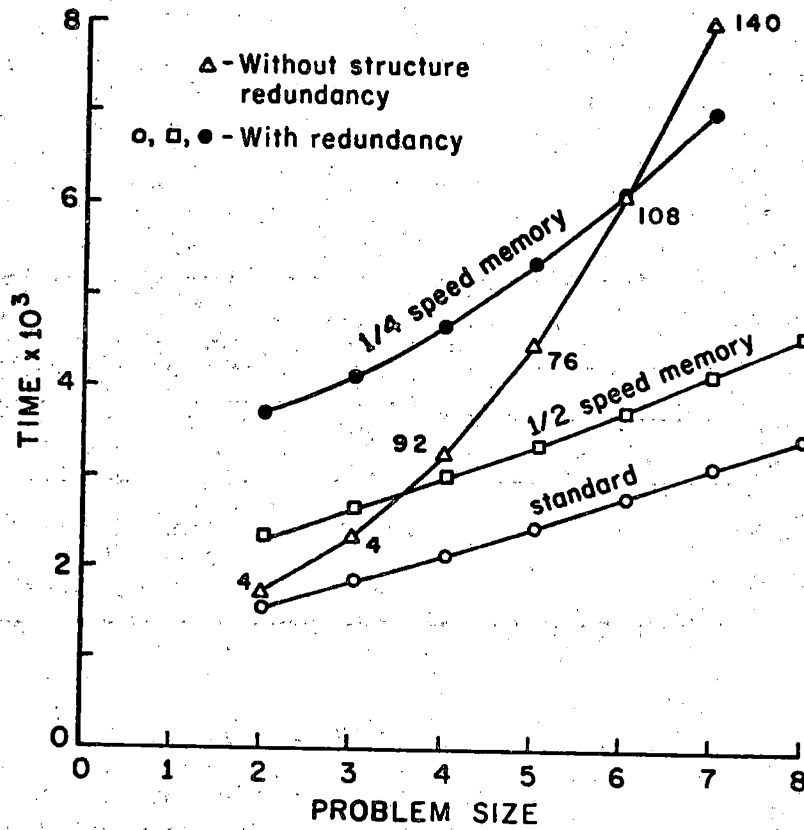


Figure 4.5

The execution complexity curves of Figure 4.2 compared with the curve obtained from a system without redundant structures in memory. Redundant data aids locality and concurrency.

Representation	Time			Space
	select	append	copy	(words)
vector	1	1 or n	n	n
2-3 tree	log n	2 log n	2n (leaves only)	8n

TABLE I

Structure representations and their assigned costs

station	(a) standard configuration	(b) memory buses @0.27 speed of standard configuration
sorter queue	.81	.53
sorter	4.00	4.00
ALU queue	3.34	2.20
ALU	8.44	8.44
code fetch*	3.15	16.67
data fetch*	3.40	25.72
output*	3.80	3.35
token bus	5.58	5.56
mean activity cycle time	32.51	66.47

*Includes box and associated queues

TABLE II

Activity flow analyses for 7×7 matrix multiply using 60 PEs

station	(a) standard configuration	(b) memory buses @0.27 speed
local bus queue*	.58	55.77
local bus*	3.20	12.00
local request processor queue	4.88	2.62
local request processor	6.88	6.61
global time**	2.14	8.00
memory response queue (in PE)	1.82	1.98
mean memory request cycle time	19.49	86.97

*Sum of request (from PE) and response (from memory controller) time

**Mean cycle time of all inter-memory controller messages proportioned by fraction of such messages out of all memory messages

TABLE III

Memory request flow analyses for 7×7 matrix multiply using 60 PEs

unit	(a) standard configuration	(b) memory buses @0.27 speed of standard configuration
sorter*	.39	.28
ALU*	.44	.32
memory controller*	.47	.35
local bus*	.20	.58
global bus	.48	.88
token bus	.29	.21

*mean duty cycle

TABLE IV

Duty cycles for 7×7 matrix multiply using 60 PEs

	standard configuration A	B	C	D
sorter queue	.90	.81	.68	.22
sorter	4.00	4.00	4.00	4.00
ALU queue	5.75	3.34	3.53	.88
ALU	8.44	8.44	8.44	8.44
code fetch*	1.23	3.15	5.36	2.54
data fetch*	2.86	3.40	3.51	2.45
output*	3.85	3.80	5.86	23.94
token bus	5.16	5.58	6.54	19.11
mean activity cycle time	32.20	32.51	37.93	61.57

*Includes box and associated queues

TABLE V

Activity flow analyses for run using 60 PEs from each of curves A, B, C, D (Figure 4.4)

REFERENCES

1. Ackerman, W.B. and J.B. Dennis, "VAL--a value-oriented algorithmic language," Preliminary Reference Manual, Laboratory for Computer Science, MIT, Cambridge, MA, June 1979.

2. A.V. Aho, J.E. Hopcroft, and J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, MA, 1974.

3. Arvind, and K.P. Gostelow, "Some relationships between asynchronous interpreters of a dataflow language," Formal Description of Programming Languages, E.J. Neuhold, Ed., North-Holland, NY, 1977, pp. 849-853.

4. _____, "A computer capable of exchanging processing elements for time," Information Processing 77, B. Gilchrist, Ed., North-Holland, NY, 1977.

5. Arvind, K.P. Gostelow, and W.E. Plouffe, "Indeterminacy, monitors, and dataflow," Proc. Sixth ACM Symp. on Operating Systems Principles, Nov. 1977, pp. 159-169.

6. 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, Sept. 1978.

7. E.A. Ashcroft, and W.W. Wadge, "LUCID - a formal system for writing and proving programs," SIAM J. Comp., 5, 3 (Sept. 1976), pp. 336-354.

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

9. A. Bahrs, "Operation patterns: an extensible model of an extensible language," Proc. International Symposium on Theoretical Programming, Lecture Notes in Computer Science 5, Springer-Verlag, NY, 1974, pp. 217-246.

10. G.H. Barnes, R.M. Brown, M. Kato, D.J. Kuck, D.L. Slotnick, and R.A. Stokes, "The ILLIAC IV computer," IEEE Transactions on Computers, C-17, 8 (Aug. 1968), pp. 746-757.

11. J.L. Bentley, "An introduction to algorithm design," Computer, February 1979, pp. 66-78.

12. L. Bic, "Protection and security in a dataflow system," TR. 126, (Ph.D. dissertation) Dept. of Inf. and Comp. Science, Univ. of Ca., Irvine, CA, Oct. 1978.

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

14. J.B. Dennis, "First version of a data flow procedure language," Lecture Notes in Computer Science, 19, Springer-Verlag, NY, 1974, pp. 362-376.

15. J.B. Dennis, and D. Misunas, "A preliminary architecture for a basic data flow processor," Proc. Third Symposium on Computer Architecture, Dec. 1974, pp. 126-132.

16. S.H. Fuller, D.P. Siewiorek, and R.J. Swan, "Computer Modules: an architecture for large digital modules," AFIPS Conf. Proc., vol. 46 (1977), pp. 637-643.

17. J.W. Glauert, "A single assignment language for data flow computing," M.S. Thesis, Department of Computer Science, University of Manchester, January 1978.

18. V.M. Glushokov, M.B. Ignatyev, V.A. Myasnikov, and V.A. Torgashev, "Recursive machines and computing technology," Information Processing 74, vol. 1, J.L. Rosenfeld, Ed., North-Holland, NY, 1974, pp. 65-70.

19. K.P. Gostelow, and R.E. Thomas, "A view of dataflow," AFIPS Conf. Proc., vol. 48 (June 1979), pp. 629-636.

20. D. Johnson, et al., "Automatic partitioning of programs in multiprocessor systems," COMPCON/80, San Francisco (Feb. 25-28, 1980), (to appear).

21. R.M. Karp, and R.E. Miller, "Properties of a model for parallel computations: determinacy, termination, queuing, SIAM J. Appl. Math., 14, 6 (November 1966), pp. 1390-1411.

22. R.M. Keller, "Parallel program schemata and maximal parallelism II: construction of closures," JACM 20, 4 (Oct. 1973), pp. 696-710.

23. R.M. Keller, G. Lindstrom, and S. Patil, "A loosely-coupled applicative multi-processing system," AFIPS Conf. Proc., vol. 48 (June 1979), pp. 613-622.

24. P.R. Kosinski, "A data flow language for operating systems programming," ACM SIGPLAN Notices 8, 9 (Sept. 1973), pp. 89-94.

25. P.J. Landin, "The next 700 programming languages," CACM, 9, 3 (March 1966), pp. 157-166.

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

27. S.S. Patil, "Closure properties of interconnections of determinate systems," Record of the Project MAC Conf. on Concurrent Systems and Parallel Computations, June 1970, pp. 107-116.

28. W. Plouffe, "Exception handling and recovery in a dataflow system," Ph.D. Dissertation, Dept. of Inf. and Comp. Science, Univ. of Ca., Irvine, CA (in preparation).

29. J.E. Rodriguez, "A graph model for parallel computations," TR-64, Dept. of EE, Project MAC, MIT, Sept. 1969.

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

31. I.E. Sutherland, and C.A. Mead, "Micro-electronics and computer science," Scientific American, 237, 3 (Sept. 1977), pp. 210-228.

32. J.C. Syre, D. Comte, and N. Hifdi, "Pipelining, parallelism and asynchronism in the LAU sytem," Proc. 1977 International Conf. on Parallel Processing, Aug. 1977, pp. 87-92.

33. R.E. Thomas, "A comparison of methods for implementing dataflow structures," Dataflow Note 35, Dept. of Inf. and Comp. Science, Univ. of Ca., Irvine, CA, May 1978.

34. ______, "Performance analysis of two classes of dataflow computing systems," TR. 120 (M.S. Thesis), Dept. of Inf. and Comp. Science, Univ. of Ca., Irvine, CA, 1978.

35. ______, "A weighted reference counting scheme for distributed memory management," Dataflow Note 44, Dept. of Inf. and Comp. Science, Univ. of Ca., Irvine, CA, 1979.

36. P.C. Treleaven, "Exploiting program concurrency in computing systems", Computer, January 1979, pp. 42-50.

37. D.A. Turner, "A new implementation technique for applicative languages," Software-Practice and Experience, Vol. 9 (1979), pp. 31-49.

38. I. Watson, and J. Gurd, "A prototype dataflow computer with token labelling," AFIPS Conf. Proc., vol. 48 (June 1979), pp. 623-628.

39. W.A. Wulf, and S.P. Harbison, "Reflections in a pool of processors - an experience report on C.mmp/Hydra," AFIPS Conf. Proc., vol. 47 (1978), pp. 939-951.