The Tree Machine: A Highly Concurrent Computing Environment

Thesis by

Sally Anne Browning

In Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy

California Institute of Technology

Pasadena, California

1980

(Submitted January 15, 1980)

Acknowledgements

The Computer Science Department at Caltech provides a marvelous backdrop for learning how to do research. The faculty's unconventional approach to Computer Science is hard to resist.

Two faculty members deserve special thanks. Carver Mead and I have maintained a dialog about Oregon and about software systems since my first week at Caltech. He proposed the tree machine to me as a potentially interesting concurrent programming environment suited to VLSI. That proposal led to the research described here. Martin Rem not only influenced the development of the tree machine notation, but was instrumental in constraining the scope of the research to a managable set of questions. His return to Holland in 1978 provided me with an excuse to see a lot of Europe while consulting him about tree machines.

Three of my fellow students have also contributed to this research. Bart Locanthi provided insights into the maze of numerical analysis algorithms and suggested improvements to the processor instruction set. Jim Rowson and Mike Ullner not only were involved in the design of many of the algorithms included here, but also had the fortitude to read and correct this document. Many thanks.

During my first two years at Caltech, I was supported by a predoctoral fellowship from the International Business Machines Corporation. My research has been partially supported by the Defense Advanced Research Projects Agency under contracts #N00123-78-C-0806 and #N00014-79-C-0597.

Abstract

An architecture for a VLSI multiprocessor machine is proposed. The processors are connected together as a binary tree. A collection of algorithms are mapped onto the tree machine. These include heap sort, transitive closure, the travelling salesman, and matrix inversion, among others. A model of computational complexity for the tree machine is suggested, and the algorithms are analyzed in the context of that model. A notation for expressing the algorithms is described, a processor design is proposed, and a compiler for the notation and processor is presented.

Table of Contents

List of Figures and Tables

Introduction

As very large scale integration (VLSI) becomes a reality, we have the opportunity to redefine the notion of what a computer is. The traditional view of a single large processing unit physically separated from a large store by a memory bus is certainly realizable in VLSI. But because processing elements and storage elements can both be implemented in silicon, we may now consider other architectures.

One new approach to designing computers is to embed an algorithm directly into silicon. There are several examples of special purpose designs in the literature, concentrated in the Proceedings of the Caltech Conference on Very Large Scale Integration held in January of 1979.

My research has centered around the notion of a general purpose computing environment that capitalizes on the properties of VLSI. Ivan Sutherland and Carver Mead discussed the properties of VLSI in "Microelectronics and Computer Science", published in Scientific American in September of 1977. Among the things they mention are 1) that most of the space on a chip is occupied by the wires that carry control and data information to the function blocks, 2) that regularity of the interconnect means that it will be less expensive in both design time and area, and 3) that local communication is less expensive than global communication. In the concluding remarks, they say

> "We believe adequate theories that account for the cost of communication will be an important guide for designing the machines that have been made possible by the integrated-circuit revolution."

In the pages that follow, a general purpose machine is presented. It is a collection of small processors, each with some local storage, connected together as a binary tree. It has a regular interconnection pattern and relies on local communication only. The machine description is accompanied by a notation that requires explicit statement of communication and a model of computational complexity in which the time cost is dominated by the communication cost. The notation and computational model are used to map a collection of algorithms from the sequential view of computation to the highly concurrent world of the tree machine. The model of computation, notation, body of algorithms, and infant machine design provide a first cut at the "adequate theory" that Sutherland and Mead describe.

The first chapter of this dissertation introduces the tree machine, and discusses the reasons why a tree-structured interconnection pattern was chosen. The notation used to program the tree is also presented, along with some backgound information about how the notation evolved. The second chapter begins with a discussion of traditional complexity theory in order to lay a foundation for the discussion of complexity on the tree machine. The chapter closes with a look at how some familiar algorithms, heap sort and transitive closure, can be mapped onto the tree machine.

The third and fourth chapters present the main body of tree machine algorithms. The third chapter introduces the notion of NP-completeness and presents algorithms for three

NP-complete problems, including the familiar travelling salesman problem. The fourth chapter addresses some matrix manipulation algorithms. In both chapters, the algorithms are analyzed according to the computational model introduced in Chapter 2.

The fifth chapter introduces the beginnings of a system design for the tree machine. It contains a block level design of the individual processors, an instruction set, a strategy for compiling the tree machine notation into the instruction set, and a protocol for loading the tree.

The sixth and final chapter talks about the contributions this work has made in emphasizing that communication is the most significant measure of cost in a highly concurrent programming environment. There is a collection of hard and as yet unsolved problems that I have found during the course of this research. Some of these are mentioned in the last chapter, pointing to future directions the research might take.

Chapter One

The Tree Machine

Processors and memory have historically been designed and implemented as distinct modules separated by a bus. This arrangement made sense at one time since the two were implemented in completely different media, and the bus made communication between them possible. With the arrival of silicon technology, however, both logic and memory can be implemented in the same medium. Why not implement them both on the same chip? Better yet, why not implement the traditional large processor and large memory as a large collection of small modules, each with a small processor and a small memory. Might not the sum of the parts be greater than the whole?

The next pages present a proposal for a machine made in just that way: a large number of small, almost inconsequential processors are assembled into a system that can outperform a large single processor system. The machine proposed here is called a tree machine: the many processors are connected as a binary tree.

Chapter One begins with a description of the tree machine, concentrating on the rationale behind the machine design. Following the machine description, a notation for programming it is given. The second chapter begins with an introduction to traditional complexity theory, recalibrates the theory to the highly concurrent world of the tree machine, and looks at some familiar algorithms in the context of the tree machine. The third chapter looks at

some problems that are intractable on single processor machines. The fourth chapter presents some matrix manipulation algorithms. The fifth chapter describes the underlying processor architecture and the system support programs. The final chapter reiterates the high points of the previous seven, and talks about future directions.

1.1 The Tree Machine

The tree machine is composed of many tiny processors, each with a small amount of program store, a few registers for data, and general arithmetic and logic capabilities. The processors are connected together to form a binary tree. That is, each processor has a single parent processor and all but the leaves have two descendents. These connections are the only communication paths in the system.

The next paragraphs will attempt to answer a series of questions about the tree machine: Why a tree? Why a binary tree? Is it a special purpose machine? Is it practical to build? If so, how can it be programmed?

1.1.1 Why a tree?

Suppose you have a pile of ten thousand processors, and have been given the task of organizing them as a system. How would you wire them together? The most general way is to connect each processor to every other processor, as shown in figure 1.1.1(a). This might work for tens of processors, but is nearly impossible for thousands of

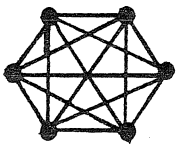
processors. In this case, each processor must have 9999 connections: you would end up with a hopeless jumble of about 50 million wires!

Suppose you connect the ten thousand processors together in a long chain, a la figure 1.1.1(b). Then there are only ten thousand wires, one per processor. But suppose the first processor wants to communicate with the last one: the message must travel the whole length of the chain, through 9998 intermediate processors. Hooking the ends of the chain together into a loop, as in figure 1.1.1(c), softens the blow a little. The longest distance a message might go is half way: through 4999 processors.

Another option is to make some kind of rectangular mesh out of the processors. Figure 1.1.1(d) shows one such pattern. Here the number of wires is manageable, and the worst case access time is faster than either the chain or the ring. The ten thousand processors can form a 100 by 100 grid; the most distant processors are separated by 197 intermediate processors.

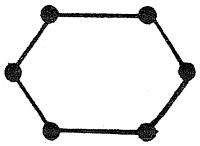
The final wiring pattern shown in figure 1.1.1 is a tree. Here again, the number of wires is proportional to the number of processors, but the access time is considerably better than any of the other schemes. Suppose the tree is a 10-ary tree: each processor has 10 descendents. The distance between the root and the leaves is $\log_{10}10{,}000=4$. The longest distance any message must travel is twice the height of the tree.

The table in figure 1.1.1 summarizes the wire requirements and access time for the five wiring patterns discussed.

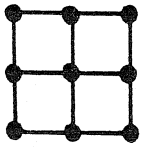


(a) complete graph

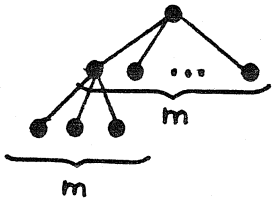
(b) chain



(c) ring



(d) mesh



(e) tree, fanout=m

wiring pattern	# wires	access time
(a) complete graph	$n(n-1)/2$	1
(b) chain	$n-1$	$n-1$
(c) ring	n	$n/2$
(d) mesh	$2(n-\sqrt{n})$	$2(\sqrt{n}-1)$
(e) tree, fanout=m	$n-1$	$2\log_m n$

Figure 1.1.1 Some Potential Wiring Patterns.

The fast access time to any processor in the tree is part of the answer to the question, "Why a tree?".

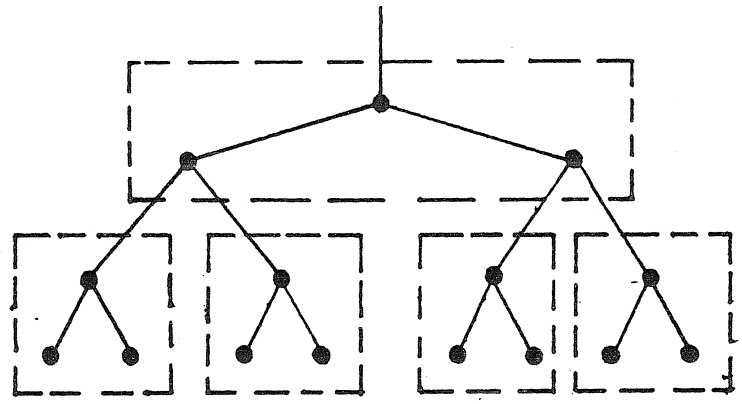
Another part of the answer has to do with the number of external connections: how many processors reside on the "edges" of the pattern. If, for example, several processors reside on a chip, how many pins must be made available to wire the chips together? Figure 1.1.2 shows the connection requirements for the five wiring patterns of figure 1.1.1. Notice that the chain, the ring and the tree are the winners: in each case, the number of pins required is independent of the number of processors on a chip. As technology improves and more and more devices can be put on a chip, we don't want to have to move to larger and larger packages.

n = number of processors in system
m = number of processors on a chip

wiring pattern	external connections
(a) complete graph	(n(n-1) - m(m-1))/2
(b) chain	2
(c) ring	2
(d) mesh	4 m
(e) binary tree (leaf chip)	1
binary tree (non-leaf)	1 + (m+1)

Figure 1.1.2 External Connections for the Wiring Patterns.

The number of external connections per chip in a tree machine depends on whether the chip contains leaf nodes or not. Chips that provide the leaves of the tree need only a single external connection. Non-leaf chips need not only the external connection for the root, but one for each internal port of the processors in the lowest level on the chip as well. This is shown below for three processors per chip and five chips in the system.



What this means is that the leaf chips are appropriate to VLSI technology: the number of external connections does not depend on the number of processors on the chip. The chips used to build the upper levels of the tree are better suited to medium or large scale integration since the number of connections grows linearly with the number of processors.

So far we have discussed physical properties of the tree. There are some conceptualization issues that also support the tree interconnection structure. The tree is an acyclic structure: there are no circular paths. This facilitates proof of freedom from deadlock. While deadlock proofs must be constructed for each tree machine program, the hierarchical nature of the tree leads to some general proof techniques.

There are a couple of ways to show that a program is free of deadlock. One, proposed by Martin Rem [Rem79b], is to draw a communication graph for the program. If that (directed) graph, a state machine, has no traps, that is, nodes that can be entered but not left, the program is free of deadlock. Another way of showing freedom from deadlock is to show that the communication can be modelled as a special kind of Petri net called a marked graph. Young-il Choo has investigated this technique for his Masters Thesis

[Choo80]. In both techniques, the hierarchy of tree machine programs makes it possible to characterize the behavior of lower levels of the tree in terms of the behavior of their external ports. As a result, proofs at each level of the hierarchy need only look one level down into the tree (instead of all the way to the bottom), and proofs can be completed with induction.

These, then, are the reasons behind choosing to wire a large number of processors together as a tree. The tree is the only one of the simple, planar wiring schemes that has both fast access time to any processor in the structure and a constant external connection function. In addition, the tree is an acyclic structure: it is easy to show that a given program is free of deadlock. A more complete treatement of wireability considerations, including analysis of nonplanar structures like the hypercube, has been done by Bart Locanthi [Locanthi79].

1.1.2 Why a binary tree?

In figure 1.1.1, a tree with arbitrary fanout was shown. Because the tree machine is an architectural proposal, a specific fanout must be chosen. Suppose that a fanout $m>1$ is chosen. Now, suppose that we have a tree machine program that requires a fanout of f and uses p processors. There are three cases to consider: 1) $f=m$, 2) $f<m$, and 3) $f>m$. What happens to the height and consequently the number of processors in the solution tree?

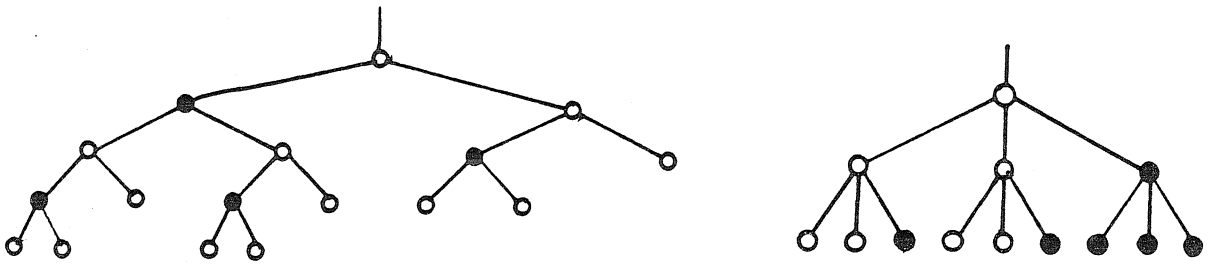
If f and m are equal, it's easy. The program and the tree match. Exactly p processors are required and the resulting

tree will be $\log_f p$ processors high.

If f is less than m, we will allocate only f of the m descendents of each processor. Figure 1.1.3(a) shows the allocation scheme for f=2 and m=3. This results in a tremendous waste as the tree gets deeper: the number of idle processors grows exponentially with the depth of the tree.

If f is greater than m, we must use several processors to simulate the increased fanout. Figure 1.1.3(b) illustrates the method. In particular, f-m processors are used to supply the proper fanout at each logical level of the tree. The padding processors form a piece of the tree that is $\log_m f$ processor high. Thus we consume extra processors at a rate linearly related to p, the number of processors required by the problem.

Figure 1.1.4 gives the precise formulae for the height and number of processors in the solution trees that arise for the three cases. Clearly, we want to avoid the exponential waste that arises when the fanout of the physical tree machine is bigger than that required by the problem. But each program will be different; we can not always guarantee a perfect fit. Thus the fanout built into the hardware should be the smallest useful one. The linear waste of simulating larger fanouts is considerably less costly than the exponential waste caused by underallocating the processors in the tree. A binary tree has the smallest fanout of all trees, ignoring the degenerate case, the chain.



(a) f<m (f=2,m=3) (b) f>m

f=hardware fanout m=program fanout

Figure 1.1.3 Allocation Technique for Mismatched Fanouts.

m = tree machine fanout
f = program fanout
p = required processors

case	tree height	extra processors
f < m	$\log_f p$	$\sum_{i=0}^{\log_f p - 1} m^i - f^i$
f = m	$\log_f p$	0
f > m	$(\log_f p)(\log_m f + 1)$	$(f-m)(p-1)/f$

Figure 1.1.4 Tree Characteristics with Mismatched Fanouts.

The table above is just a little simplistic. While there is a linear growth in the number of processors required to establish the program fanout when the physical fanout is too small, the resulting tree is unbalanced whenever the program fanout is not a multiple of the physical fanout. The physical structure <u>is</u> balanced, and must be as deep as the longest path, given as the tree height in figure 1.1.4. The correct number of unused processors must also include the ones that are unneeded but present because of the unbalance. This number is

$$\left(\sum_{i=0}^{h} m^i \right) - p \quad \text{where } h=(\log_f p)(\log_m f+1)$$

This number grows exponentially with the depth of the tree. If there were some way to keep the tree balanced, the number of extra processors would grow linearly as predicted in the table above. James T. Kajiya of Caltech has proposed a tree-like architecture that can balance itself. It has a few extra communication paths that allow the location of parent and children processors to be less rigid than in a strict binary tree. The proposed structure retains the planar layout feature, and as such, may be practical to build. If the work continues to be as promising as the early reults, his structure may be the way to build a binary tree.

1.1.3 Is the Tree Machine a Special Purpose Machine?

In the past decade several machines that intermingle logic and memory have been designed, each with a specific purpose in mind. A survey paper by Thurber [Thurber75] describes many of them. More recent examples are the hexagonal array machine of Kung and Leiserson [Kung79] for matrix multiplication, radar tracking machines [Denny79], and sorting machines [Armstrong77,Bently79]. Is the tree machine yet another special purpose machine?

No. We will show that the tree machine is a general purpose computing machine. In the next few chapters examples of a wide range of algorithms for the tree machine

are presented. In each case, a special purpose machine could be, and often has been, designed to do the task more quickly. But none of those special purpose machines can do the variety of problems that the tree machine can: one operation is made fast at the expense of another. The tree machine presents a system with balanced capabilities; it is general purpose.

1.1.4 Is it Practical to Build a Tree Machine?

The question of buildability has already been answered, in part. The wiring pattern is planar and the number of wires grows linearly with the number of processors. Figure 1.1.5 shows a possible layout of processors on a chip. Notice that the length of the wires doubles every two levels from the leaves to the root. Notice also that the amount of space available for a processor grows with each level. There is enough room to build the beefier drivers needed to insure constant communication time between levels of the tree. Bart Locanthi has addressed the question of power requirements for the tree machine in [Locanthi79].

A processor architecture is proposed in the fifth chapter. It too is buildable. In fact, if we assume that a logic cell is roughly equivalent in area to four dynamic memory cells, and that the processor is made up of equal number of memory and logic cells, about ten processors can be put on a chip using the same design rules that produce 64K RAMs. Given the very large scale integration of the 1980's we can expect several hundred processors on a chip!

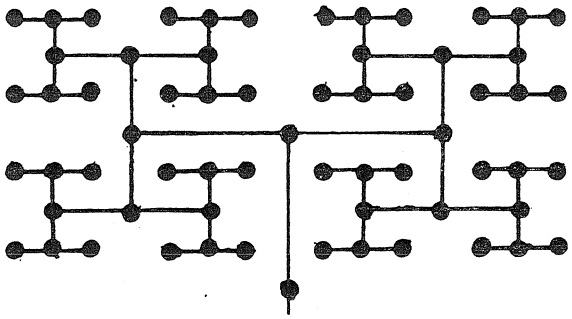


Figure 1.1.5 A Planar Layout of a Binary Tree Machine.

1.1.5 How is the Tree Machine Programmed?

Other tree-connected machines have been proposed that identify potential concurrency in an algorithm written in a higher level language. Examples are the applicative language machine of Mago [Mago79] and the various LISP machines [Keller78,Locanthi80].

The philosophy proposed here is different. We assume that the programmer is the ultimate expert on the algorithm. We expect the algorithm to be designed with concurrency in mind from the beginning. To that end, a high level programming notation is introduced in the next section, and used throughout the rest of this paper. Chapter Six describes a compiler that provides the mapping from logical to physical fanout, but the programmer must break the problem into modules that will run on the processors of the tree machine. A tree machine algorithm is successful only if the programmer fully understands the problem at hand.

The remainder of this chapter shows the evolution of the tree machine notation. The original algorithms were programmed in Simula, an object-oriented language described briefly in section 1.2. As this proved unsatisfactory due to inadequate communication primitives, the algorithms were

reprogrammed in CSP, a notation specifically designed to provide interprocess communication facilities. CSP is the subject of section 1.3. This too was unsatisfactory because the underlying machine architecture is not reflected in the notation. A notation for tree machines that reflects the architecture has developed, and is described in section 1.5. It is used for the algorithms presented in later chapters.

1.2 A First Step: Simula

Simula [Birtwhistle73] is essentially a super-set of Algol-60 with an important addition: the class data type. A class is a collection of data and procedure declarations bound together as a single unit. The data is local to the class object, and the procedures manipulate the data. In fact, if locality of data is emphatically enforced, the only access to the data stored in a class instance is through procedure calls. Then as long as procedure functionality is preserved, the underlying implementation can be modified without affecting existing programs that reference the class.

Each class instance can be thought of as an independent machine, responding to well-defined requests from the outside world in the form of procedure calls. This metaphor is useful for describing concurrent programs. The data attributes correspond to local storage cells, the procedures to actions initiated by messages from neighboring processors.

Another feature of Simula is the ability to factor out things common to a set of related classes. These common attributes are combined into a superclass, that is, a class which is an antecedent of all the others. The subclasses, descendents of the class of common attributes, provide a definition of the variant parts.

This factoring technique leads to hierarchical definitions. The tree machine, for example, can be defined hierarchically. CLASS processor might define the hardware capability at each node: number of registers, words of program store, communication protocol. This class is a superclass of the collection of classes representing the code to solve each problem. That is, CLASS sort might supply program text for heap sort while CLASS clique describes the algorithm for finding the largest clique in a graph. Each class is a subclass of CLASS processor, sharing the same hardware definition.

Simula is, however, intended for expressing programs that will run on single processor. There is no facility for sending messages between processors. Procedure calls are an inadequate substitute because the calling processor is blocked until a return from the called processor is executed. There is also no clean way to indicate concurrent execution, although it can be simulated at the expense of many lines of code irrelevant to the problem at hand.

Another problem with Simula is its verbosity. A tree machine algorithm typically involves several different processor definitions, each doing only a small piece of the computation. The collection of processors must be viewed

together in order to understand the algorithm. A language like Simula, with many ways to say the same thing and requiring many symbols to say anything, forces lengthy descriptions at the expense of clarity.

The original tree machine algorithms were programmed in Simula and appear in [Browning79c].

1.3 A Second Step: CSP

Hoare's notation for Communicating Sequential Processes (CSP) [Hoare78] certainly doesn't suffer from verbosity. He bases CSP on the concise, restrictive language proposed by Dijkstra [Dijkstra76], adding message-passing facilities and the ability to describe concurrent computation.

I will begin with a description of Dijkstra's language, then describe Hoare's additions, and finally give a critique of my experiences with CSP. The syntax that I use does not exactly match that used by Dijkstra or Hoare, although the semantics are the same.

1.3.1 The Basic Statements

Dijkstra defines a language with five kinds of statements: skip, abort, assignment, a loop, and a conditional statement. There are no procedure calls, no goto's, no choice among ten different ways to write a loop.

There is a unifying notion of a guarded command. The guard is a condition. If the guard is true the command that

follows may be executed. The statement "if c is less than n, interchange the values of c and n" is written as

```
c < n --> c,n:=n,c
```

More than one statement can be included in the command following the guard. Each statement is separated from the next by a semicolon. There is no semicolon following the last statement in the command.

Guarded commands appear in the conditional and loop statements. The conditional statement is analogous to the *if-then-else* and *case* statements of Algol-style languages. It has the form

```
[ guard --> command
! guard --> command
...
! guard --> command ]
```

There can be as many guarded commands included in the statement as desired. There is no implied order in the attempt to satisfy a guard, however. That is, unlike Algol, Simula, Pascal, etc, if the second guard is tested and found true, it does not mean that the first guard is not true. There is no catchall *else* condition that is accepted when all other guards are false. In other words, each guard must be fully specified.

The guarded command given above can be written as a conditional statement:

```
[ c < n --> c,n := n,c
! c >= n --> skip ]
```

The second guard, c>=n, must be included even though the command that follows, skip, is a no-op. When no guard within a conditional statement is satisfiable, the conditional statement fails and the program will abort.

The other statement that contains guarded commands is the loop construct. It is written

```
( guard --> command
| guard --> command
...
| guard --> command )
```

The statement is executed repeatedly as long as any guard can be satisfied. When none of the guards are true, the loop terminates. The loop construct never aborts. It can, however, never terminate. Semantically, there is no difference between nontermination and abortion [Dijkstra76,p.35].

The sample conditional statement can also be written as a loop:

```
( c < n --> c,n := n,c )
```

The loop will be executed at most once. If c is at least as large as n, the guard is not satisfiable, and the loop is terminated without the values of c and n being swapped. If c is less than n, the values are interchanged and the guard is reevaluated. This time c is certainly larger than n and the loop is terminated.

The equivalent of Algol's for loop is written with explicit initialization, testing, and incrementing of the control variable. There is no special language construct.

```
i:=1; ( i <= n --> PRINT; i:=i+1 )
```

The skip statement, as mentioned above, is a no-op. The abort statement causes an error termination of the program. The assignment statement is much like assignments in Algol-style language, except that a list of variables can appear on the left side, with a corresponding list of values on the right. These are all valid assignment statements:

```
i := 1;
n,c := c,n
a := 2 * b/5;
```

It must be remembered, however, that all of the expressions on the right hand side of the assignment are evaluated before any assignment is made to variables on the left hand side.

1.3.2 Statements to Express Parallelism

The language constructs just described are used to define sequential, though possibly non-deterministic, programs. To these Hoare has added a message sending capability, a class-like structure for defining process units, and statements to specify parallel execution.

1.3.2.1 Messages

A message has a name and, optionally, some arguments. Examples are increment or load(a). In the first case, the message name carries sufficient information. In the

second, additional information is provided by the argument.

Messages are always sent to and received from named processes. The process name is separated from the message name by a symbol indicating the direction of the communication. An exclamation point (!) means send the message to the named process. A question mark (?) means receive this kind of message from the named process. A message is sent only when both the sender and receiver are willing to process it. Input statements, those with a ?, can appear as guards, but output statements cannot. An input statement guard is satisfiable if the specified message can be received from the specified process.

1.3.2.2 Process Definitions

A process definition specifies local data attributes and some program text that accepts, sends, and responds to messages from other processes. It also supplies a generic name for the kind of process. Processes are replicated in arrays, with a subscript appended to the generic name. For example, to model a bucket brigade, a process fireman might be defined and replicated fifty times:

```
fireman(i:1..50)::
( fireman(i-1)?bucket --> fireman(i+1)!bucket )
```

Here a subscript, i, indicates one of the fifty firemen. A fireman is prepared to receive a bucket from the left and pass it to the right.

Suppose the empty buckets travel back to the well through the same brigade, but from right to left. The program text

for fireman can accommodate the change:

```
fireman(i:1..50)::
[ fireman(i-1)?bucket --> fireman(i+1)!bucket
| fireman(i+1)?empty --> fireman(i-1)!empty ]
```

The endpoints of the brigade are special since there is no fireman to the left of fireman(1) or to the right of fireman(50). There must be special definitions for them. Assume that there are 75 buckets that can be in use at one time. Fireman(1) can fill a bucket only if there is one available.

```
fireman(1)::
integer b;
b := 75;
[ b>0 --> b:=b-1; fireman(2)!bucket
| fireman(2)?empty --> b:=b+1 ]

fireman(i:2..49)::
[ fireman(i-1)?bucket --> fireman(i+1)!bucket
| fireman(i+1)?empty --> fireman(i-1)!empty ]

fireman(50)::
[ fireman(49)?bucket --> fireman(49)!empty ]
```

There is a problem with the program given above. Deadlock can arise if fireman(1) is trying to send a bucket to fireman(2) at the same time that fireman(2) is trying to tell fireman(1) that the brigade is empty. The way to fix this problem is change the first guarded command in the defintion of fireman(1) to read

```
b>0 and fireman(2)!bucket --> b:=b-1
```

Regrettably, Hoare does not allow the use of output statements in guards.

1.3.2.3 Indicating Parallelism

The third feature that Hoare adds is the ability to indicate parallelism. Each process is a sequential one, but a collection of them can execute in parallel with the messages providing any required synchronization. The symbol || is used to indicate that the named processes execute concurrently. The bucket brigade is set in motion by the statement

```
fireman(1) || fireman(2..49) || fireman(50)
```

1.3.3 A Critique of CSP

CSP does not suffer from the limitations attributed to Simula. A message facility is an integral part of the notation, there are facilities for describing parallel execution among processes, and the language is concise. There is also something like a class construct for defining local data attributes and confining external access to specific entry points.

However, CSP has some deficiencies. First, the hierarchical definition capability of Simula is missing. There is no way to define a superclass that contains the attributes that all the processes share. Instead, the common part must be repeated in each variation. Second, the only way to replicate processes is as arrays. Using subscripting conventions to represent the tree as an array obscures its structure.

Also, CSP has no way of restricting the communication of a process to a select set of other processes. That is, there is no way to enforce a particular interconnect structure. The notation allows a process to communicate with any other process it can name. In a tree, as with any large physical structure, there is a very rigid set of processors that can communicate directly with a given process. The notation should emphasize the physical structure.

By choosing to represent communication as messages sent between named processes, Hoare has made it impossible to assemble a collection of process definitions that can be used as building blocks in many different programs. The processor names are locked into the program text since a process must know the name of its partner in communication. Thus each process definition must be tailor-made for a specific program.

1.4 A Tree Machine Notation

This section proposes a notation that is a blend of Simula and CSP ideas, with some useful additions. While the notation, used throughout the remainder of this paper, is tailored to the tree machine, it can easily be made more general. Extensions are discussed at the end of the chapter.

The syntax of the notation is derived from CSP. The skip, abort, assignment, loop, and conditional statements are identical to those in CSP, and have already been described. The message statements are also similar. Processes correspond directly to the processors of the tree machine.

The processors are defined as self-contained units that communicate through ports to other processors. The processor definitions assume that the ports are capable of processing messages. They assume nothing about the identity of the processor on the other end of the port. This is a significant departure from Hoare's style of communication, where process definitions name other processes directly.

There are several advantages to naming communication partners indirectly by using ports. A processor can be defined without regard to the eventual connection plan of the network. A processor expects to follow a specific communication protocol when accessing a port. Any processor that follows the same protocol can be connected to the other end of the port.

This definitional locality makes possible a parts kit of standard processor definitions. Each part is a processor or tree of processors that can be characterized by the behavior of its ports. As long as the expected messages are sent and received, the part will work anywhere in the system.

In addition to the ability to define processors, the notation provides a mechanism for specifying the correspondence between ports and the tree processor interconnections of the complete machine. The mapping of the processor's ports onto the physical communication paths must be supplied in order to complete the program description. This mapping is one-to-one and onto: every port must be connected to another port, and no two ports can be connected to the same port.

A complete tree machine program has two parts: a set of processor definitions and a connection plan. The syntax of these will be discussed later.

1.4.1 Messages

Messages are written just as they are in CSP except that a port is named instead of a process. Another difference is that output statements as well as input statements can appear as guards. An output statement guard is satisfiable if the destination processor is willing to receive the message. Output statement guards are used repeatedly in the algorithms of Chapters 2 and 3. As in CSP, input statement guards can be satisfied if the specified port can receive a message.

The notation also supports broadcast output. A given message can be sent to several ports simultaneously. An output statement can have a list of destination ports each of which will receive the message. If an entire array of ports is to receive the message, the subscript '*' is used to specify all of the array elements. For example, if L, R, and S(1:5) are ports, these are valid output statements:

```
L!hi
L,R!hi
S(3)!hi
S(*)!hi
R,S(*)!hi
S(i),S(4)!hi
```

Broadcast mode output is used frequently in the algorithms of Chapters 2 and 3, usually to send a message to all of the sender's descendents.

Input statements can specify a list of ports as well. In this case, the message can be received from any one of the named ports. Thus if L, R, and S(1:5) are ports, L,R?hi means accept the message 'hi' from either L or R. The message S(*)?hi means accept the message from any of the S ports. This notation can only be used where it really doesn't matter which port received the message. There is no way to retrieve the identity of the sender unless it is part of the message itself.

1.4.2 Processor Definitions

A tree machine program is a collection of modules. Each module is a processor. A processor definition begins with the keyword processor, a name for the processor, and, optionally, an argument. Next come the port and data declarations. The body of statements that form the program text of this module complete the definition. We will look at each of the components of the processor definition in more detail.

The input and output statements name ports to be used in the communication. Each processor can have external ports and internal ports. Every processor is the root of a tree of processors. Communication with this tree is possible only via the external port of the root processor. The internal ports are used for communication between the processor and its subtrees.

A processor must have exactly one external port, but may have any number of internal ports. Leaves have an external

port, but no internal ports. Programs are written for an arbitrary tree of processors, and mechanically translated into programs for a binary tree.

The definition of a processor that sorts marbles by color is given below. The processor, called colorSorter, has one port to the outside world and three ports to subtrees. The processor expects to receive marbles from its external port and will send them to an internal port chosen according to the color of the marble. If the marble is blue, it is routed to the B port, similiarly for green and red marbles. If it is not blue, green, or red, the marble is sent to the reject bin. The color is represented by an integer, with blue=1, green=2, and red=3.

```
processor colorSorter;
external port In;
internal port B,G,R,reject;
integer color;

{ In?marble(color) -->
        [ color=1 --> B!marble
        | color=2 --> G!marble
        | color=3 --> R!marble
        | color<1 or color>3 --> reject!marble
        ]
} .
```

A processor definition can be parameterized to allow one processor definition to fit a variety of situations. The parameter is a constant within the processor definition, and it cannot be modified during the computation. The parameter can be used to specify the number of internal ports a processor has. It can be used as a constant in an expression or in any situation that does not attempt to change its value. It cannot be the target of an assignment, since it is a constant, not a variable.

The processor definition for colorSorter is rewritten to handle n colors instead of three:

```
processor colorSorter(n);
external port In;
internal port c(i:n),reject;
integer color;

{ In?marble(color) -->
        [ i<=color<=n --> c(color)!marble
        | color>n or color<i --> reject!marble
        ]
} .
```

In order to complete the example, the definitions of the processors connected to the internal ports of colorSorter must be supplied and the interconnection scheme must be specified. The missing processor definition is given below. The bin processor keeps a count of how many marbles it has received. Bin is a leaf node in the tree so it has no internal ports.

```
processor bin;
external port p;
integer cnt;

cnt:=0;
{ p?marble --> cnt:=cnt+i } .
```

1.4.3 The Connection Plan

The processor definitions define the building blocks that the program will use. The connection plan shows how they fit together.

The keyword tree indicates a connection plan. It is followed by a name for the tree and an optional list of parameters. The remaining statements in the plan describe

connections between processors. A connection specifies which kinds of processors will be connected to the internal ports of a given processor. A connection is always between an internal port of one processor and the external port of the other. The external port connections, then, are implicit in a connection statement. I will begin with an explanation of the shorthand form of a connection plan, and gradually move toward the general form.

The connection plan for colorSorter and the bins can be expressed very simply as

```
tree marbleSorter(n):  colorSorter(c(1:n),reject) ~ bin .
```

This statement says that the tree is called marbleSorter. It is parameterized by a value, n, that must be provided before the tree can be built. A colorSorter is the root of the tree. It has n+1 internal ports, named c(1), c(2), ..., c(n) and reject. These are all connected to bin processors. The symbol '~' is read "is connected to". Implicit in the statement is that there are n+1 bins, each with its external port connected to colorSorter. The tree looks like this:

Since the fanout of colorSorter can be determined from the processor definition and all the internal ports are connected to bin processors, the connection plan can be further abbreviated as

```
tree marbleSorter(n):  colorSorter ~ bin .
```

The connection plans given so far are shorthand for the more general form. The example gives a tree with two levels, each made of different processors but with all the same kind of processors on a given level. Suppose that this is not the case. If the reject bin in the marble sorting example is a special processor called Rbin that keeps track of the total number of marbles that are successfully sorted as well as the number of bad ones, the tree gets a little more complicated. Here are a modified definition of colorSorter and a definition of Rbin.

```
processor colorSorter(n);
external port In;
internal port c(i:n), reject;
integer color;

( In?marble(color) -->
        [ i<=color<=n --> c(color)!marble ;
                          reject!good
        | color<i or color>n --> reject!bad
        ]
) .

processor Rbin;
external port p;
integer g,b;

g,b:=0,0;
( p?good --> g:=g+1 | p?bad  --> b:=b+1 ) .
```

The connection plan becomes

```
tree marbleSorter(n):
        colorSorter(c(i:n))~bin ,
        colorSorter(reject)~Rbin .
```

Again, there are two kinds of processors on the second layer of the tree: n bins and one Rbin. Here, however, connection statement has two parts, separated by a comma.

The comma signifies that the order of the two parts of the statement does not matter. Each part fully specifies which ports are connected to what type of processor.

Suppose the tree had more than two levels. This might happen if, in addition to differing in color, the marbles could be either square, round, or some other shape. If the marbles are sorted first by shape and then by color, a three level tree could be used:

```
tree fancyMarbleSorter(n):
    shapeSorter(square,round)~colorSorter ,
        shapeSorter(reject)~Rbin ;
    colorSorter(c(1:n))~bin,
        colorSorter(reject)~Rbin .
```

Notice that the second connection statement describes a previously defined tree. Since a tree has a single external port, it can be connected to an internal port of a processor in the same manner a single processor is connected. Another way of of describing the tree above is given here:

```
tree fancyMarbleSorter(n):
    shapeSorter(square,round)~marbleSorter ,
        shapeSorter(reject)~Rbin .
```

A three level tree requires two connection statements in the connection plan. In order to avoid writing k statements for a tree that is k+1 levels high, a shorthand is provided for replicating a connection. For example, imagine a tree made up of three kinds of processors: a root, some leaf processors, and in between, mid processors that each have three descendents. If there are M layers of mids, the tree can be described as follows:

```
tree t(M):
    root ~ mid ;
    M-1  ( mid ~ mid ) ;
    mid ~ leaf .
```

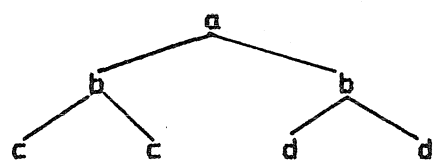
The shorthand means "repeat the connection (mid ~ mid) M-1 times". The resulting tree will have M levels of mid processors, and M-1 levels of connections between them.

In order to achieve a pattern of alternating processors by layers, one might write

n (a ~ b ; b ~ a)

This will give 2n+1 levels consisting alternatively of a's and b's.

Suppose the tree below is to be built:



It cannot be described with the connection plan presented thus far. The connection plan tree t ; a~b ; b ~ c ; b ~ d is ambiguous: which b is connected to the c's and which to the d's? Here, each b must be identified via the path to it from the root of the tree. This tree is described recursively by defining the two subtrees that have root nodes of b processors, and then assembling the complete tree from these.

```
        tree firstB:   b ~ c .
        tree secondB:  b ~ d .
tree t:  a(p1) ~ firstB , a(p2) ~ secondB .
```

The general form of the connection plan provides for layer repetition, a heterogeneous collection of processors on a given layer, and uniquely naming a port in terms of its path from the root. The trees described in Chapters 2 and 3 tend to be simple, requiring only the shorthand notation.

1.4.4 Syntax Definition

1.4.4.1 Processor Defintion Syntax

```
<processor> ::= processor <name> ';' <declPart> ';' <body>
<name> ::= <ident> | <ident>'(' <ident> ')'


<declPart> ::= <portDecl> | <portDecl> ';' <typeDecl>
<portDecl> ::= <eport> | <eport> ';' <iport>
<eport> ::= external port <ident>
<iport> ::= internal port <pnList>
<pnList> ::= <pn> | <pn> ',' <pnList>
<pn> ::= <ident> | <ident> '(' <expr> ':' <expr> ')'


<body> ::= <statement> | <statement> ';' <body>
<statement> ::= skip | abort | <iteration> | <conditional>
                | <assignment> | <message>

<iteration> ::= '{' <guardList> '}'
<conditional> ::= '[' <guardList> ']'
<guardList> ::= <guardedBody>
                 | <guardedBody> '|' <guardList>
<guardedBody> ::= <guard> '-->' <body>
```

```
⟨guard⟩ ::= ⟨booleanExpr⟩ | ⟨message⟩

⟨assignment⟩ ::= ⟨identifierList⟩ ':=' ⟨exprList⟩
⟨identifierList⟩ ::= ⟨ident⟩
                    | ⟨ident⟩ ',' ⟨identifierList⟩
⟨exprList⟩ ::= ⟨expr⟩ | ⟨expr⟩ ',' ⟨exprList⟩

⟨message⟩ ::= ⟨portList⟩ '?' ⟨contents⟩
            | ⟨portList⟩ '!' ⟨contents⟩
⟨portList⟩ ::= ⟨port⟩ | ⟨port⟩ ',' ⟨portList⟩
⟨port⟩ ::= ⟨ident⟩ | ⟨ident⟩ '(' ⟨expr⟩ ')'
         | ⟨ident⟩ '(*)'
⟨contents⟩ ::= ⟨ident⟩ | ⟨ident⟩ '(' ⟨exprList⟩ ')'
```

1.4.4.2 Connection Plan Syntax

```
⟨tree⟩ ::= tree ⟨name⟩ ':' ⟨connectBody⟩ '.'
⟨name⟩ ::= ⟨ident⟩ | ⟨ident⟩ '(' ⟨ident⟩ ')'

⟨connectBody⟩ ::= ⟨connectList⟩
                | ⟨connectList⟩ ';' ⟨connectBody⟩
⟨connectList⟩ ::= ⟨connection⟩
                | ⟨connection⟩ ',' ⟨connectList⟩
⟨connection⟩ ::= ⟨ident⟩ '~' ⟨ident⟩
               | ⟨ident⟩ '(' ⟨pnList⟩ ')' '~' ⟨ident⟩
               | ⟨expr⟩ ( ⟨connectBody⟩ )
⟨pnList⟩ ::= ⟨pn⟩ | ⟨pn⟩ ',' ⟨pnList⟩
⟨pn⟩ ::= ⟨ident⟩
       | ⟨ident⟩ '(' ⟨expr⟩ ':' ⟨expr⟩ ')'
```

The usual definitions are assumed for ⟨ident⟩, ⟨expr⟩, and ⟨boolean expr⟩. Comments can appear wherever a blank can,

and are enclosed in double quotes: "This is a comment...

1.4.5 Future Directions

This notation describes static trees. The tree is assembled before the computation is initiated During the computation, its structure remains unchanged. Static behavior is sufficient for the algorithms presented here. It might be interesting to examine trees with dynamic behavior.

The tree structure is reflected in the notation only because each processor is limited to a single external port. By removing this restriction and requiring that external as well has internal ports be explicitly declared in the connection statements, other structures can be described. The programs retain the important characteristics: the processors are locally sovereign and the network is characterized by the traffic through the external ports.

Another extension of the language is to allow substructures other than trees to be defined and included in connection plans. For example, user defined composite processors might be nice, especially as a library of processor definitions develops. I have not included them because of vaguely disquieting issues that arise. First, substructures are no longer characterized solely by the behavior of the external port. This may affect the provability of the structure. Second, it becomes harder to specify the behavior of the ports. Timing assumptions about the processors inside a substructure may have to be

made. Since a definitional capability for other substructures is a useful one to have, however, it should be investigated. My misgivings may be unfounded.

Messages in this notation are matched on the basis of like names. This is fine for custom processors and small problems, but might be cumbersome as libraries of processor definitions are developed, and complicated problems are tackled. It might be necessary to augment the notation with a way of explicitly specifying the matching of messages between processors.

Finally, there is no capability in the notation for indicating the overall strategy of each algorithm. The individual processors are defined and connected, but it is seldom apparent from these two things what the complete tree is supposed to do. In this thesis, I will present a textual, informal description of the strategy. As it is better understood, a formal specification should be developed.

Chapter Two

The Analysis of Algorithms

There are many different ways to solve a given problem. Complexity theory provides a framework for evaluating each solution, or algorithm, so that the best one can be chosen. The usual measures of complexity are the time and space requirements of the algorithm. The relationship of time and space to the size of the problem shows the asymptotic behavior of the algorithm as the problem gets arbitrarily large.

Complexity measures are given as functions of the size of the problem, usually called n. For instance, the space and time requirements of a sorting algorithm depend on the number of items to be sorted. If n elements can be sorted in cn^2 time for some positive constant c, the algorithm is said to be $O(n^2)$ in time, read "order n^2".

The space requirements of an algorithm are computed by counting the number of storage locations it requires. The time complexity is usually a measure of the number of operations that must be performed on each input, and should properly be called an operation count.

Most measures of complexity are based on a particular model of computation that is chosen to suit the problem domain. For example, analysis of matrix manipulation is done in the framework of a machine with an array of memory, while list processing algorithms are more at home in a computational model with a pointer space. I will describe several different models of computation, ending with a description

of one that can be used to analyze the complexity of tree machine algorithms. The first model, the Random Access Machine (RAM) is discussed in more detail than the other sequential machine models because the individual processors of the tree machine are modelled as RAMs.

Each computation model is accompanied by criteria for calculating time and space costs for algorithms programmed in the context of the model. Most of the traditional models make the assumption that memory cells are equally remote from the processor. Algorithms are not penalized for ignoring locality in accessing memory.

Memory access is a degenerate form of the interprocess communication found in the tree machine. In the tree machine context it becomes apparent that a distance penalty must be assessed for each communication as part of the time cost of an algorithm. Another model that penalizes random access algorithms is the Turing machine model. Since the tape can advance only one cell per step, much as a tree machine message can traverse only one level in the tree at each cycle, examination of adjacent cells is cheap while examination of distant cells is costly. However, the Turing machine program is stored in a lookup table that can be viewed as infinite. There is no distance penalty applied to program elements stored in this table. Thus the model is inconsistent in its treatment of communication costs.

2.1 Random Access Machine Model

The most common model for complexity analysis is the random access machine (RAM) [Aho74]. A RAM has a single processor

with one accumulator, an unlimited memory, and some input and output ports. One integer can be read from or written to the ports or the memory at a time. Integers can be of arbitrary size.

The instruction stream does not occupy memory. Thus, it cannot modify itself. The exact instruction set doesn't matter. Arithmetic, logical, input/output, memory addressing, and branching instructions can be assumed, along with anything else that is common to the instruction repertoire of real machines. RAM programs will be stated here in pidgin Algol. Because asymptotic costs are given, the cost difference between Algol and machine language is absorbed in the constant.

The time and space costs of a RAM program can be viewed from two perspectives. The uniform cost criterion assumes that all instructions take the same amount of time and that each datum occupies the same amount of space. The logarithmic cost criterion attempts to recognize that memory is limited. Costs are assigned depending on the number of bits it takes to represent the instruction and the data. Both cost models are illustrated in the following paragraphs.

Consider the following problem. Given a vector a, generate another vector x such that

$$x_i = \sum_{j=i}^{i} a_j \quad \text{where} \quad i=1,2,\ldots,n$$

The two problems that follow are RAM solutions to the vector summing problem described above. The first algorithm mimics the problem definition. The second is optimal for the RAM computational model.

```
integer array a[1:n], x[1:n];
integer i,j;

for i:=1 to n do
begin
    x[i] := 0;
    for j:=i to i do
    x[i] := x[i] + a[j];
end;
```

Figure 2.1.1 Vector Summing Solution #1 (RAM model).

The first program, figure 2.1.1, computes the values for the vector x by repetitively summing the values of a. The uniform cost space complexity is found by counting the memory requirements. There are two vectors of n elements each, plus two temporary variables, so the space cost is 2n+2 or O(n). The time complexity is somewhat more complicated. The outer loop is executed n times and the inner loop is executed i times for the i^{th} iteration of the outer loop. Thus the time required, given the uniform cost criterion, is

$$n + (1+2+\ldots+n) = n + n(n-1)/2 = (n^2+n)/2 = O(n^2)$$

The second algorithm, figure 2.1.2, requires only one loop to do the computation because it takes advantage of previous work. The space complexity remains O(n), but the time complexity is improved. The single loop executes n

times and contains two instructions. Thus, it will take 2n or O(n) time.

```
integer array a[1:n], x[1:n];
integer i,sum;

sum := 0;
for i:=1 to n do
begin
    sum := sum + a[i];
    x[i] := sum;
end;
```

Figure 2.1.2 Vector Summing Solution #2 (RAM model).

So far, costs have been based on the uniform cost criterion. Another way of assigning costs is to relate the cost to the number and size of the operands. This is called logarithmic cost [Aho74], and is based on the assumption that the cost of an instruction is the sum of the lengths of its operands. The length of an operand is the number of bits required for its representation:

$$l(i) = \begin{cases} \log_2 i + 1, & i>0 \\ 0, & i=0 \end{cases}$$

To compute the logarithmic time cost for the more efficient program, figure 2.1.2, it is useful to look at a machine code translation, given in figure 2.1.3. The time costs for each instruction can be tediously computed using the formula given above and assessing a penalty for the length of the operand and for the length of data fetched in each memory access. The resulting time cost is $O(n\log_2 n + n\log_2 s)$, where $\log_2 s$ is the maximum number of bits it takes to represent the sums or vector elements. Logarithmic space complexity is defined as the sum, over all memory and the accumulator, of the length of the largest number stored

in each cell during the computation. Here again, the space complexity is $O(n\log_2 s)$.

```
            load    =0          ; sum:=0
            store   sum
            load    =1          ; i:=1
            store   i
for:        load    =n          ; for i:=1 to n do
            sub     i
            jneg    done
            load    a           ; fetch a[i]
            sub     =1
            add     i
            store   ind
            load    *ind        ; sum:=sum+a[i]
            add     sum
            store   sum
            load    x           ; x[i]:=sum
            sub     =1
            add     i
            store   ind
            load    sum
            store   *ind
            load    i           ; i:=i+1
            add     =1
            store   i
            jump    for         ; loop
done:       halt
```

Figure 2.1.3 Machine Language Translation

Since logarithmic costs are much more difficult to compute than uniform costs. Thus, uniform costs are usually used with the RAM model of computation. The logarithmic criterion is an attempt to be realistic about memory accesses. There are easier ways of doing that: the tree machine model provides a less cumbersome technique.

2.2 Reference Machine Model

The RAM model assumes the existence of an unstructured store. Many algorithms access memory in a prescribed manner, however, and have no need of the less structured access capability. List processing techniques are an important example of this kind of algorithm. Reference machines [Tarjan77] have been proposed as a computational model that is more appropriate for such problems.

Like a RAM, a reference machine is a processor, a memory, and a collection of registers. The memory is organized as a pool of records, each containing entries that are either data or references to other records. All records are identical in structure.

There are registers that can hold data, and registers that can hold references. All CPU operations apply to these registers. The instruction repertoire includes instructions to transfer data between either kind of register and the memory, to perform the usual arithmetic and logical operations, to create new records, and to conditionally branch.

The main difference between this model and the RAM is that the reference machine severely limits access to the memory. Explicit reference is required. This makes the model appear less powerful than the RAM: some algorithms that rely on address arithmetic cannot be used. However, it can simulate LISP and other list processing languages. Thus, the reference machine is merely different from the RAM, not less powerful.

The reference machine model is interesting because it recognizes that memory access is not free. Regrettably, the model stops short of the point. Though memory accesses are restricted, there is no distance measure applied to make clumped accesses cheaper than widespread ones.

2.3 Turing Machine Model

A Turing machine [Turing36] is pictured in figure 2.3.1. It has an infinitely long tape that is divided into cells, each of which can hold one symbol. One cell at a time is scanned by a tape head that can read and write the tape.

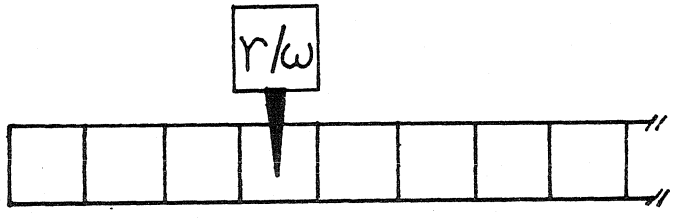


Figure 2.3.1 A Turing Machine

The Turing machine is controlled by a primitive program that describes transitions from one state to another based on the cell being scanned by the tape head. The program is represented as a set of quadruples (initial state, input symbol, action, next state). The action taken can be one of three things: move the tape one cell to the left, move the tape one cell to the right, or write a character into the cell under the head. No two quadruples in the set can have the same initial state and input symbol.

The Turing machine is initialized to a specified state. From there, actions are taken and new states entered by finding a quadruple such that the current state and symbol

under the tape head match the initial state and input symbol in the quadruple.

As an example, we will look at the Turing machine program for adding two integers. First, however, we need to set the stage by defining the Turing machine. A Turing machine is defined by a quintuple (Q,I,b,q_0,q_f), where Q is the set of states, I the set of input symbols, b the blank character, q_0 the initial state, and q_f the final state. The Turing machine for this example, then, is $(\{s_0,s_1,s_2,s_f\},\{1,0\},0,s_0,s_f)$.

Numbers will be represented in <u>unary</u> form, as a string of ones. Thus the number five is represented as the string '11111'. The first number in the sum starts in the leftmost cell of the tape. The other number is separated from the first one by the blank character '0'. The sum will be written on top of the two numbers. Figure 2.3.2 shows the tape for forming 3+4.

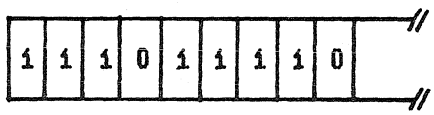


Figure 2.3.2 Sample Turing Machine Tape.

The program proceeds this way: scan the tape, moving right, until the blank character is found. Write a '1' in that cell and continue scanning. When the next blank character is encountered, move left one cell, write a '0' there and halt. Thus the tape of figure 2.3.2 is transformed into '1111111', the unary representation of

seven. Figure 2.3.3 gives the Turing machine program described above.

$(s_0, 1, R, s_0)$
$(s_0, 0, 1, s_1)$
$(s_1, 1, R, s_1)$
$(s_1, 0, L, s_2)$
$(s_2, 1, 0, s_f)$

Figure 2.3.3 Turing Machine Program: Addition.

The Turing machine model of computation is even more primitive than the RAM and reference machine models discussed in the previous sections. Turing invented it before digital computers existed as a framework for comparing abstract computational processes. The model is too primitive to be useful for most complexity analysis. However, it can compute anything a RAM machine can in polynomially related time. That is, if the Turing machine takes $f_1(n)$ time and the RAM takes $f_2(n)$ time, then there exist polynomials $p_1(x)$ and $p_2(x)$ such that $f_1(n) \leq p_1(f_2(n))$ and $f_2(n) \leq p_2(f_1(n))$ for all values of n.

This relationship between the RAM and Turing machine models comes in handy when analyzing programs where the polynomial blowup is lost in the noise. For example, if the time complexity of an algorithm is an exponential function, all polynomially related functions are also exponentials. This fact is used in analyzing the set of NP-complete problems described in the next chapter.

I have described a Turing machine that has a single tape and behaves in a deterministic manner. That is, no two quadruples can have identical initial states and input

symbols. Other flavors of the machine have been proposed: multi-tape machines, nondeterministic machines, and multi-tape nondeterministic machines. For each flavor there is a set of problems that benefit from being analyzed on that particular computational model. Aho, Hopcroft, and Ullman [Aho74] and Tarjan [Tarjan78] discuss these variations in more detail.

2.4 Tree Machine Model

All of the computational models presented above are sequential machines, but the tree machine environment is highly concurrent. A different model of computation is needed to describe the complexity of tree machine algorithms.

Since each processor in the tree machine is a deterministic, sequential processor, the traditional view of complexity can be applied to each individual node of the tree. In the complexity analyses that follow, the RAM model with uniform cost criterion is used to characterize the behavior of the individual processors.

The messages exchanged between processors introduce synchronization and dependencies among the nodes of the tree, and must be figured into the time cost of the tree machine algorithm. The uniform time cost is appropriate here because messages can travel between adjacent levels of the tree in one time step. The farther a message has to go, the greater the cost. Thus, a message travels from the root of a tree of n processors to the leaves in $O(\log_2 n)$ time.

Every message does not contribute to the time cost of the algorithm, however. The time cost is really a measure of the sequentiality of the algorithm. Suppose that a message is broadcast from a processor to all its descendents in synchrony. The broadcast communication has the same cost as a communication from the parent to a given descendent because the elapsed time is the same. Similarily, if two communications in different parts of the tree occur in parrallel, the time cost is the cost of one communication. The second one, happening in parallel, is free. Thus, overlapping communications do not individually contribute to the time cost of the algorithm.

Space complexity of tree machines is easy to calculate. Each processor is of constant size, and once the data space in a processor is used, another processor must be allocated. The space cost, then, is the number of processors used to solve a problem.

Let us return to the vector summing problem of the RAM discussion. Given a vector a, calculate x such that

$$x_i = \sum_{j=1}^{i} a_j \quad \text{where} \quad i=1,2,..,n$$

The tree machine algorithm for vector summing is presented in detail in a paper I presented at the Caltech VLSI Conference [Browning79a]. The initial vector a is loaded into the tree in infix order, as shown in figure 2.4.1. The summing begins in the left corner and proceeds up and to the right as the a_i's are replaced with the

corresponding values of x. In the following paragraphs we will examine the algorithm in detail in order to understand how time costs are assigned in tree machine algorithms.

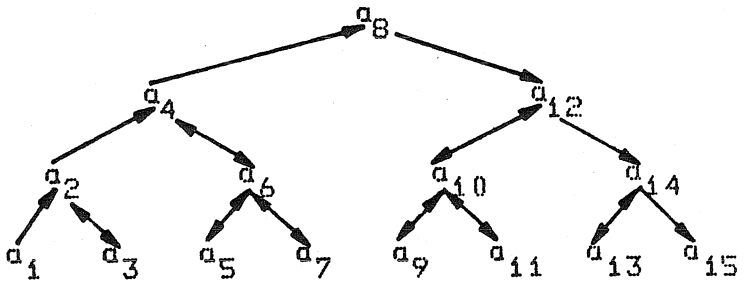


Figure 2.4.1 Vector Summing Tree, n=15.

The arrows in figure 2.4.1 indicate the flow of messages in the tree. The starting place is the leftmost leaf, called: a_1. The computation ends in the rightmost leaf, labeled a_{15}. As the wave front of summing approaches the root, more and more processors are operating in parallel. As the computation crawls back down the right half of the tree fewer and fewer sums are formed in parallel. The net result is that all sums are formed by the time the last one is, and the last one is formed as quickly as a message can travel from the leftmost leaf to the rightmost leaf. That distance is $2\log_2 n$. Thus, the time spent computing the vector sum is $O(\log_2 n)$.

2.4.1 The Vector Summing Tree

Figure 2.4.1 shows a tree machine configuration for n=15. The connection plan we write down must be a general one, parameterized to fit any value of n. Because all the ports of every processor used in the tree must be hooked up to something, we must define the root, leaves, and middle

processors separately. Thus, we define vSumRoot, vSum, and vSumLeaf as the basic building blocks of the vector summing tree.

In figure 2.4.2, we see that there is one set of connections between the vSumRoot and vSum processors, and one set of connections between vSum and vSumLeaf processors. These will remain constant with n. The parameterization, then, belongs in the declaration of vSum to vSum connections. The connection plan is given in 2.4.2. Note the second line: the number of vSum to vSum connections is $\log_2 n - 2$. When n=15, this is indeed 1. (The logarithm is truncated to an integral value.)

```
tree vSumTree(n):
    vSumRoot ~ vSum
    log2n-2 ( vSum ~ vSum ) ;
    vSum ~ vSumLeaf .
```

Figure 2.4.2 The Vector Summing Tree.

2.4.2 The vSumRoot Processor

Each of the three kinds of processors have four activities to undertake. Variables must be initialized, the vector values must be loaded into the proper nodes, the sums must be formed, and the answer must be unloaded from the tree. The vSumRoot is responsible for beginning the initialization phase. It also controls the loading and unloading of the vector values. The vector summing, however, begins in the leaves of the tree.

During the initialization phase of the computation, the vSumRoot sends two different messages to its descendent

subtrees: expect and subscript. The expect message tells the processor how many values must go to the left, how many to the right, and which one will stay here. That is, the processor that represents the fourth element in a vector of length 15 will see seven values during the loading phase: the first three are destined for processors in the left subtree, the fourth value becomes the value stored in this processor, and the remaining three are sent to the right subtree. The subscript message informs each processor which element of the vector it represents. The subscript is used in the summation step to insure that the right things get summed together.

The loading phase, partially described above, is also initiated by the vSumRoot. It reads load messages, n of them, from the system bus. These are distributed to the left and right subtrees, with the middle values remaining in the root, assigned to the variable x.

The vSumRoot plays a relatively passive role in the vector summing process, mimicking the code of the vSum processor. It will receive a partial sum from the left subtree. Since all the subscripts to the left are less than the subscript of the root element, that sum should be added in to the value of x. Correspondingly, all the elements in the right subtree have subscripts greater than the one in the root. Thus, their values are not needed for the sum stored in the root.

This left and right handedness applies to every subtree in the graph, not just to the root, and is the key to the algorithm. During the summation phase, sumUp and sumDown messages fly furiously around the tree. The message names

sumUp and sumDown indicate the direction in which the messages travel. In each, the messages carry a partial sum, and an integer that reflects the element with the largest subscript that has contributed to the sum. sumUp messages from the left are always added into the x value. The sumUp messages from the right never affect x. The root never receives a sumDown message. It does, however, send them, both to the left and to the right.

Once the summation is accomplished, the unloading of the new vector is initiated by the vSumRoot. A technique similar to the loading process is employed: the first half of the values come from the left, then this element is unloaded, and the other half come from the right. This mirrors the infix arrangement of the elements in the tree. The infix visitation rule is to visit the left subtree, visit this node, and then visit the right subtree.

The definition of the vSumRoot processor is given below, as figure 2.4.3. When we view this processor as an entity separate from the tree we find that the initialization and summation phases are independent of the size of the problem. Thus, they are constant time operations. The loading and unloading do depend of the problem size, however, and are linear in n, that is, O(n). Thus this algorithm can be no better than O(n) in time.

```
processor vSumRoot(n);
external port bus;
internal port l,r;
integer i,x,t,s;

"initialization"
s := n/2 ; l,r!expect(s,i) ;
l!subscript(s/2) ; r!subscript(3*s/2) ;

"load the values of vector a"
i:=i ; [ i<= n/2 --> bus?load(t) ;
                     l!load(t) ; i:=i+i ] ;
bus?load(x) ;
i:=i ; [ i<= n/2 --> bus?load(t) ;
                     r!load(t) ; i:=i+i ] ;

"sum the values to form vector x"
l?sumUp(t,i) ; x:=x+t ; l!sumDown(t,i) ;
r?sumUp(t,i) ; r!sumDown(x,s) ;

"unload the values of vector x"
i:=i ; [ i<= n/2 --> l?unload(t) ;
                     bus!load(t); i:=i+i ] ;
bus!unload(x) ;
i:=i ; [ i<= n/2 --> r?unload(t) ;
                     bus!load(t); i:=i+i ] .
```

Figure 2.4.3 The vSumRoot Processor.

2.4.3 The vSum Processor

The vSum processor fills essentially the same role in the computation as the vSumRoot does, with two differences. First, the vSum is not connected to the system bus. All the load messages it touches come from the root. Secondly, since it is descendent from the root, the vSum processor receives as well as sends expect, subscript, and sumDown messages. The sumDown messages provide the vehicle for values in the left subtree to be included in sums formed in the right subtree.

The complexity characteristics of the vSum processor are reminiscent of the vSumRoot: loading and unloading costs vary with n, while summing and initialization are constant time operations. The actual cost of the loading and unloading operation is complicated by the fact that the position of the vSum processor in the tree dictates the number of elements it will see. If L represents the level in the tree at which a vSum processor resides, then the time complexity can be written as $O(n/2^L)$. (The root is level 0.) Since this is less than O(n), the algorithm remains no worse than linear.

```
processor vSum;
external port p;
internal port l,r;
integer x,s,i,n,t,v,j;

"initialization"
p?expect(n,v) ; l,r!expect(n/2,v+1) ;
p?subscript(s) ;
l!subscript(s/2) ; r!subscript(3*s/2) ;

"load the values of vector a"
i:=1 ; [ i<= n/2 --> p?load(t) ;
                     l!load(t) ; i:=i+1 ] ;
p?load(x);
i:=1 ; [ i<= n/2 --> p?load(t) ;
                     r!load(t) ; i:=i+1 ] ;

"form the sums"
l?sumUp(t,i) ; x:=x+t ; l!sumDown(t,i) ;
r?sumUp(t,i) ; r!sumDown(x,s) ; p!sumUp(x+t,i) ;
j:=1 ;
[ j<=v -->
    p?sumDown(t,i) ;
    [ i<s --> x:=x+t | i>=s --> skip ] ;
    l!sumDown(t,i) ; r!sumDown(t,s) ;
    j:=j+1
] ;

"unload the vector x"
i:=1 ; [ i<= n/2 --> l?unload(t) ;
                     p!unload(t) ; i:=i+1 ] ;
p!unload(x) ;
i:=1 ; [ i<= n/2 --> r?unload(t) ;
                     p!unload(t) ; i:=i+1 ] .
```

Figure 2.4.4 The vSum Processor.

2.4.4 The vSumLeaf Processor

The vSumLeaf processors have no subtrees beneath them to worry about. Thus, the messages they send and receive are much simpler than elsewhere in the tree. The vSumLeaf receives and ignores the expect message. It will see only one value during the loading step, and send only one value in the unloading step. The subscript message initializes

the variable s. During the vector summing phase of the computation the vSumLeaf will send a sumUp message and receive a sumDown message. The sumUp initiates the stream of like messages flowing through the tree, and the sumDown terminates the vector summing phase. If the subscript contained in the sumDown message is less than the subscript stored in the processor, the partial sum is added to the value stored in variable x. Otherwise the partial sum is ignored. The final action of the vSumLeaf is to unload its newly formed value of x.

The program text for the vSumLeaf processor is given in figure 2.4.5. Notice that n, the size of the vector, is in no way involved in any of the steps of the computation. The execution of vSumLeaf program, then, is constant in time.

```
processor vSumLeaf;
external port p;
integer x,s,v,t,i,j;

"initialization"
p?expect(s,v) ; p?subscript(s) ;

"load the values of vector a"
p?load(x);

"vector summing"
j:=1 ;
{ j<=v -->
    p!sumUp(x,s) ; p?sumDown(t,i) ;
    [ i<s --> x:=x+t | i>=s --> skip ] ;
    j:=j+1
} ;

"unload the values of vector x"
p!unload(x) .
```

Figure 2.4.5 The vSumLeaf Processor.

2.4.5 The Complexity of the Tree Machine Algorithms

The space complexity of a tree machine program is very easy to calculate: one merely counts the number of processors used in the solution. In this case, the vector summing of a vector containing n elements requires a processor for each element. Thus, the space complexity is precisely n.

We have seen that the time complexity of the individual processors in the tree is no worse than O(n). That is only (an insignificant) part of the picture, however. In order to understand the time requirements of this and other tree machine algorithms, we must look at what happens when the processors play together.

In the vector summing problem, we can calculate the time complexity by computing the longest distance that any

message must travel in each phase of the computation. Since pipelining can be employed to overlap the transport time of multiple pieces of data, the number of messages that travel the maximal distance does not usually affect the order of the time cost.

The initialization phase calls for a message to travel from the root to the leaves. In a tree of n processors, this distance is $\log_2 n$. During the loading and unloading steps, n numbers travel the full height of the tree, again $O(\log_2 n)$. Since pipelining allows the movement of data to overlap, the time cost of all n messages is $O(n+\log_2 n)=O(n)$ rather than $O(n\log_2 n)$.

The vector summing phase begins in the left-most leaf and ends in the right-most leaf. The sumUp and sumDown messages travel the height of the tree once. That is, the summation step requires $O(\log_2 n)$ time.

The overall time complexity is the sum of all the pieces, in this case, $O(n)$. As with many tree machine algorithms, the loading and unloading time dominates the time cost. The computation time, reduced as it is by the concurrency, is lost in the noise. If we assume that the tree is already loaded with the vector and the answer is left in the tree, the computation time of $O(\log_2 n)$ is an improvement over the linear computation time of the best sequential machine algorithm (figure 2.1.2).

In the next two chapters a varied collection of algorithms are mapped onto the tree machine. In each case the complexity of the individual processors does not contribute

to the complexity of the tree machine solution as a whole. The most significant activity is the flow of messages throughout the tree. Thus, the programming style that is emphasized is one that limits the length and number of communications between processors. The processors are made as independent of each other as possible, and the communication is restricted as much as possible to nearest neighbor conversation. The flow of control of the algorithm is distributed throughout the tree, with no central puppeteer controlling the actions of the other processors. While this style of programming is a bit unsettling when one is used to RAM-style machines, the results are very satisfying.

2.5 Sorting

The problem of sorting a set of elements into ascending or descending order has received much attention. There are a variety of algorithms available that can arrange n numbers in the prescibed order in $O(n\log_2 n)$ time [Knuth73]. Since the elements cannot be loaded faster than one at a time, any tree machine algorithm will require at least O(n) time to sort n elements.

The lower bound time complexity, O(n), is achieved by the algorithm that follows. When the numbers have been loaded into the tree machine, the largest number in the set is available in the root of the tree. As soon as it is removed, it is replaced with the largest among those remaining. Thus, the numbers are sorted in the time it takes to load and unload the tree.

I have chosen to implement heap sort on the tree and will use one processor for every number to be sorted. As the numbers are loaded into the tree, they are partially sorted. The largest number in any subtree is stored in the root of that subtree. As the numbers are unloaded, the sorting is completed: each root of a non-empty subtree is refilled with the largest value among its descendents. During loading and unloading the tree remains balanced.

The program defined here never terminates. It will always be either waiting for a new number to sort or trying to output one of the sorted numbers. It is a simple matter to modify it to terminate when a special end-of-file indicator appears in the input stream. Several interesting extensions to the basic sorting tree are discussed later: a stack, a queue, and a priority queue. These data structures are implemented on a tree machine with small modifications to the non-terminating sorting algorithm.

The tree is defined with the connection plan below.

```
tree sortTree(n):
    log2n-2 ( sort ~ sort ) ;
    sort ~ sortLeaf .
```

Figure 2.5.1 The Sort Tree.

As indicated, there are two kinds of processors in the tree. It is a binary tree with height $\log_2 n$. The leaf processors are sortLeafs; the others are sort processors.

The processor definitions are given in Figure 3.1. There are two values stored in each sort processor: num is one of the numbers to be sorted, cnt is a count of how many

processors in the subtree rooted at this processor are storing one of the input numbers. If cnt=-1, this processor is not currently storing a number, and can accept one. If cnt=0, the only valid number in the subtree is here, at the root. If cnt>0, there is at least one number in the subtree. The leaf processors record a number in num; they don't need a cnt variable as there is no subtree beneath them.

Each processor will either accept a load message from or send an unload message to the processor connected to its external port. If a load message, with argument c, is received, the processor will either keep c or pass it on, depending on the current value of cnt. If cnt=-1, the value of c is assigned to num. If cnt>-1, the larger of c and num is kept as num, and the smaller is passed either to the left port, if cnt is even, or to the right port, if cnt is odd. The tree is kept balanced by alternating between the two internal ports.

If an unload message is sent, a new value of num must be selected from among the values in the subtree. Cnt is again used in the replacement selection. If cnt=0 there are no more valid numbers in the subtree. If cnt=1 only the left subtree has a number in it. That number becomes the new value of num. If cnt>1, both subtrees contain a valid number so the larger one must be chosen. This is accomplished by asking for both candidates, keeping the largest one, and reloading the smaller one. The program text for the load message is mimicked in the reloading operation to maintain the balance of the tree.

```
processor Sort;
external port p;
internal port l,r;
integer cnt,num,c;

cnt := -1;
{ p?load(c) -->
    [ cnt = -1 --> num := c; cnt := 0
    | cnt >= 0 -->
        "keep the largest one of c and num"
        { c > num --> num,c := c,num } ;
        "lean to the left, lean to the right ..."
        [ even(cnt) --> l!load(c)
        | odd(cnt) --> r!load(c) ] ;
        cnt := cnt + 1
    ]
| p!unload(num) -->
    [ cnt = 0 --> cnt:=-1
    | cnt = 1 --> l?unload(num) ; cnt := 0
    | cnt > 1 -->
        "reload with largest value in subtrees"
        "this is a repeat of the load code"
        l?unload(num) ; r?unload(c) ;
        { c > num --> num,c := c,num } ;
        [ even(cnt) --> l!load(c)
        | odd(cnt) --> r!load(c) ] ;
        cnt := cnt - 1
    ]
} .


processor SortLeaf;
external port p;
integer num;

{ p?load(num) --> p!unload(num) } .
```

Figure 2.5.2 The sort and sortLeaf Processors.

This discussion wouldn't be complete without mentioning that several machines have been proposed to sort things in linear time. Armstrong [Armstrong77] has looked at sorting memories, Bently and Kung [Bently79] have proposed a double tree as a VLSI sorting engine, and Thompson and Kung [Thompson79] have analyzed mesh connected architectures for sorting. None of these machines can get around the fact that until the data is loaded and unloaded in parallel, the

least amount of time and space needed to sort n things is O(n).

With minor modifications to the program text, the sorting tree machine can be used to imitate either a stack or a queue. A stack is a data structure with the property that the last item inserted into the stack will be the first one removed from the stack. By changing the program text in processor Sort to always keep the new number (c) and pass on the old number (num), we can simulate a stack. A queue has the property that the first number inserted into the queue will be the first number removed from the queue. If the sort processor is modified to always pass on the new number, it will act like a queue. In both cases, the unload behavior is changed so that the processor is refilled alternately from the left and right subtrees.

By adding a delete operation to the sort tree described here, we can extend it to act as a priority queue [Aho74, p.147]. Delete removes a given value from the set of numbers stored in the tree. The load operation corresponds to the insert operation of the priority queue, and unload is the same as identifying the minimum or maximum value in the queue. Load and unload take a constant amount of time per element loaded or unloaded. Delete, on the other hand, can take as much as $O(\log_2 n)$ time to complete. While delete operations can overlap each other, all deletes in progress must finish before an unload is initiated. Thus, the time it takes to process a series of actions on the priority queue is related not only to the number of operations, but the order in which they come, as well. The worst possible sequence is to never allow the deletes to overlap: each delete is followed by a load or unload that

must wait for completion. In this case, the time cost is $O(n\log_2 n)$, just as it is on sequential machines. The best possible sequence has no deletes at all, and it takes O(n) time to complete. The expected behavior lies somewhere in between.

Leiserson [Leiserson79] presents an architecture for multiple priority queues that utilizes an array of primary store for each queue, backed up by a single tree-shaped secondary store. He shows that load and unload operations can be performed on m priority queues in parallel in constant time. While he ignores the costly delete operation in his discussion, the integration of array and tree structures is interesting.

2.6 Transitive Closure

Let G be a directed graph with n nodes and e arcs. The transitive closure G*, itself a directed graph, is generated from G as follows. If u and v are nodes of G, then there is an arc from u to v in G* if there is a path from u to v in G.

There are at least two ways to generate the transitive closure of a graph on a single processor machine. Let B be an nxn incidence matrix for G. That is, $b_{ij}=1$ if arc (i,j) is in G, and $b_{ij}=0$ otherwise. Then B^n is the incidence matrix for G*. Since matrix multiplication is an $O(n^3)$ operation and B^n is computed by doing n matrix multiplications, this method of forming the transitive closure takes $O(n^4)$ operations.

Warshall's algorithm [Warshall62] is a more efficient method. An incidence matrix is again used to represent the graph. The algorithm, figure 2.6.1, employs three nested loops that traverse the matrix adding arcs. After k steps of the outer loop, matrix element $b_{i\,j}$=true if and only if there is a path from i to j through intermediate nodes taken from the set {1,2,...,k}.

```
boolean array b[1:n,1:n] ;
integer i,j,k;

for k:=1 to n do
  for i:=1 to n do
    for j:=1 to n do
      if b[i,k] and b[k,j]
      then b[i,j]:=true ;
```

Figure 2.6.1 Warshall's Algorithm for Transitive Closure.

On a sequential machine, this algorithm takes $O(n^3)$ time, an improvement of an order of n over the matrix multiplication method.

Several improvements on these two algorithms for transitive closure have been presented recently. By performing some preprocessing on the incidence matrix to remove cycles, the matrix can be expressed in upper triangular form. Matrix theory is brought to bear on the problem, showing that transitive closure can be formed in $O(n^3/\log_2 n)$ time [Arlazarov70], or in the time required to perform one boolean matrix multiplication [Munro71]. Fischer and Meyer [Fischer71] show that transitive closure is at worst $O(n^{\log 7}(\log_2 n)^2)$. (The time cost of the most efficient matrix multiplication algorithm on sequential machines is $O(n^{\log 7})$ [Strassen69].)

Each of the algorithms in the preceding paragraph have made only minor improvements on the basic $O(n^3)$ time complexity of the problem. The ultraconcurrency of the tree machine provides the luxury of not having to squeeze every last drop of unnecessary activity from the algorithm in order to achieve a lower time complexity. We can and will use one of the less complicated, less efficient algorithms as the starting place for the tree machine algorithm.

A technique for multiplying matrices on the tree machine is presented in a later chapter. It can be used to implement the first transitive closure algorithm, requiring $O(n^2)$ for a single multiplication, and $O(n^3)$ time to form the closure. A tree machine implementation of Warshall's algorithm yields better results.

Before looking at the code in detail, a few words need to be said about the overall strategy. There are two key points in Warshall's algorithm. First, it is cascading. That is, newly created arcs can effect the creation of other arcs in the closure. Any realization of the algorithm on the tree machine must include this characteristic. It is not sufficient to consider only the arcs of the original graph.

Also important is the comparision between arcs. This comparison, stated in figure 2.6.1 as

```
if b[i,k] and b[k,j] then b[i,j]:=true
```

can be translated into English: if there is an arc from i to k, and another arc from k to j, then create an arc from i to j. If, instead of using the incidence matrix

representation of the graph, we represent the graph as a list of arcs, we can build a tree machine that represents all possible arcs in the graph and prune it down, by means of the arc list, to only those arcs that are in the closure.

In the graph problems of this and the next chapter, this technique of enumerating an exhaustive set of all possible answers and weeding out the wrong ones is used over and over again. As was emphasized in the discussion of complexity theory as it relates to the tree machine, we treat space (processors) as an unlimited resource in order to minimize the time required to arrive at a solution.

2.6.1 The Transitive Closure Tree

The connection plan for the tree designed to generate the transitive closure of a directed graph is given in figure 2.6.2. It is a three level tree, with closureRoot, node, and endNode processors. The closureRoot and each of the node processors have a fanout of n. The endNodes are the leaves of the tree.

```
tree closure(n);
    closureRoot ~ node ;
    node ~ endNode .
```

Figure 2.6.2 The Transitive Closure Tree.

Each node to endNode connection represents a potential arc in the closure. All possible arcs in a directed graph of n nodes are represented by the n^2 connections between node and endNode processors. As the arcs are processed the tree will be (figuratively) whittled down to include only the

arcs actually in the closure.

Each endNode processor represents an arc from its parent to the node it represents. Conceptually, think of them as little lights. If the light is off, the arc does not yet exist in G^*. If the light is on, the arc exists. As the arcs are processed, endNodes light up, new arcs are broadcast around the tree, and more lights come on. When all the lights that can be lit are lit, the algorithm is done. The endNodes that are still in the dark are the ones that have been pruned from the tree.

In the next few paragraphs the code for each type of processor will be examined thoroughly. It will become apparent that the time complexity of the individual processors is constant. The messages flowing through the tree dominate the time complexity of the problem. The time required to compute the closure is O(a), where a is the number of arcs in the closure. There are certainly no more than n^2 arcs in the graph.

2.6.2 The closureRoot Processor

The closureRoot processor begins by initializing the tree. Each node is assigned a node number, with a set message. The external port bus of the root is connected to some system bus that can provide the arcs of the graph. The closure, also in the form of arc messages, is sent out on the bus as well.

Remember that Warshall's algorithm is a cascading one. While the original arc messages come from the system bus,

once the computation gets under way new arcs are generated among the node processors. These too are received by the closureRoot and broadcast both to the system (as part of the answer) and to the tree (to effect the creation of more arcs).

The main part of the program text in the closureRoot processor is a repetitive statement with all four guards being input statements. The closureRoot wants an arc message from either the bus or any node processor, a done message from any port, or a notDone message from any internal port. The done and notDone messages control a semaphore used to determine when the generation of the transitive closure is complete. The semaphore, active, is initally equal to n+1, the number of ports in the closureRoot. The reception of a done message decrements active, and a notDone message increments it. When active reaches zero, the closure has been found. The final action of the closureRoot processor is to send a done message to the system bus.

The processor definition of closureRoot is given as figure 2.6.3, below.

```
processor closureRoot(n);
external port bus;
internal port v(i:n);
integer i,j,active;  boolean more;

"initialization - set node number"
i:=1 ; { i<=n --> v(i)!set(i) ; i:=i+1 } ;

"read arcs from bus and internal ports"
"pass up and down"
"done when ACTIVE=0"
active := n+1 ;
{ active>0 -->
    [ bus?arc(i,j) --> v(*)!arc(i,j)
    | v(*)?arc(i,j) --> v(*)!arc(i,j) ; bus!arc(i,j)
    | bus?done --> active:=active-1
    | v(*)?done --> active:=active-1
    | v(*)?notDone --> active:=active+1
    ] ;
} ;
v(*),bus!done .
```

Figure 2.6.3 The closureRoot Processor.

2.6.3 The node Processor

The first thing the closureRoot processor did was send an initialization to the processors via its internal ports. Consequently, the first action the node processor takes is to receive a set message from its external port to initialize its node number in the variable myV. Once that is done, the arc processing phase of the computation begins.

The boolean d is used to remember which message was sent last: done or notDone. Recall that these two messages are used to increment and decrement a semaphore in the closureRoot.

The other important message the node deals with is the arc message. When an arc message is received, the endpoints in

the message, i and j, are compared to myV. If the starting point of the arc is this node (i=myV), or there is an existing arc from myV to i (i\=myV and mi), the arc (myV,i) is added to the closure. This is precisely the test of the sequential Warshall's algorithm given in figure 2.6.1.

If an arc is received that does not create a new arc in a particular node processor, a done message is sent if d is false. If d is true and a new arc is created, a notDone precedes the transmittal of the new arc. The two remaining states, d=true with no new arc, and d=false with a new arc, do not cause a change in d; nor do they force the transmittal of done and notDone messages.

The program text for the node processor is given below, in figure 2.6.4.

```
processor node(n);
external port p ;
internal port v(i:n) ;
integer myV,i,j; boolean mi,mj,d,more;

"initialization - get node number"
p?set(myV) ;

"get arcs from external port"
"d=true if a done message was sent"
d:=false ;  more:=true ;
{ more -->
    [ p?arc(i,j) -->
        "if the arc is from myV or there is"
        " an arc (myV,i), set mark"
        v(i)?mark(mi) ; v(j)?mark(mj) ;
        [ (i=myV and mj) or (i\=myV and not(mi)) -->
                [ not(d) --> d:=true ; p!done
                | d --> skip ]
        | (i=myV and not(mj)) or (i\=myV and mi) -->
                v(j)!setMark;
                [ d --> p!notDone ; d:=false
                | not(d) --> skip ] ;
                p!arc(myV,j)
        ]
    | p?done --> more:=false ; v(*)!done
    ] ;
} .
```

Figure 2.6.4 The node Processor.

2.6.4 The endNode Processor

The endNode processor is very simple. It stores a boolean state variable called m. The value of m is true if there is an arc from the parent node to the endNode, and false otherwise. Only two messages are of interest: the endNode will receive a setMark instruction to set m to true, and it will send a mark message to display the state of m.

One might wonder why the endNode even exists. Couldn't a boolean array in the node processor serve the same purpose? Functionally, the effect would be the same. However, the

array would be n cells long. This introduces the size of the problem, n, into the required storage of each processor. This tie between the size of the processor and size of the problem is to be avoided at all cost. The processors are of fixed size and the characteristics of the problem are allowed to influence only the number of processors in the tree, not the characteristics of those processors.

The endNode definition is given in figure 2.6.5.

```
processor endnode;
external port p;
boolean m,more;

"mark is TRUE iff there is an arc "
"from the parent to this node"
m := false ;  more:=true ;
{ more -->
    [ p!mark(m) --> skip
    | p?setMark --> m := true
    | p?done --> more:=false
    ]
} .
```

Figure 2.5.6 The endNode Processor.

2.7 Closing Remarks

This chapter began with a survey of computation models useful in the analysis of algorithms. It ended with a demonstration of a particular model, the highly concurrent tree machine, in action. Two classic algorithms, heap sort and transitive closure, have been mapped from a sequential context to the tree machine, and analyzed in the new context.

Heap sort is an $O(n\log_2 n)$ algorithm on a sequential machine. The time complexity can be reduced to $O(n)$ on the tree machine. Transitive closure is essentially an $O(n^3)$ algorithm in the sequential world. On the tree machine it becomes $O(n^2)$. Both of these time requirements correspond to the load/unload time of the data, and, as such, represent lower time bounds. The space requirement for both heap sort and transitive closure is the same as required by their sequential machine counterparts: heap sort takes $O(n)$ space in both contexts, and transitive closure takes $O(n^2)$. There is much to be gained by treating space as an unlimited resource and concentrating on reducing the time requirement to the bare minimum.

Chapter Three

NP-Complete Problems

In this chapter the notion of NP-completeness will be introduced. This class of problems is intractable on sequential machines, requiring exponential amounts of time for a solution to be found. When the problems are mapped onto the highly concurrent tree machine, the results are startling. Given the premise of unlimited processors, the Np-complete problems can be solved in polynomial time.

The chapter begins by describing the characteristics of NP-complete problems. Three problems are mapped onto the tree machine: the clique problem, the color-cost problem, and the travelling salesman problem. The chapter ends with a discussion of the tree machine approach to solving these intractable problems.

3.1 An Introduction to NP-Completeness

Complexity theory has established a context within which it is possible to make certain statements about the inherent complexity of computations. These statements are universally couched in the terminology of sequential machines. There is, however, a class of problems for which the possibility of large scale concurrency has been addressed.

Consider a computation in which there are n decision points. At each decision point a choice must be made among q possible branches. If there is enough information available to decide which branch to take at each step, a sequential machine will be able to complete the computation in n steps.

In many computations there is not enough information available to decide which branch to take at the time the decision is made. In this case, a sequential machine must either pick a path at random or apply some heuristic. If the chosen path turns out to be wrong, the machine must backtrack to the decision point and try another route.

Imagine a machine that always chooses the right branch at a decision point, even though there isn't enough information to go on. Such a machine cannot, of course, be built with real logic operating with real programs. These imaginary machines with foresight are called nondeterministic, and can solve the computation mentioned above in linear time. Even without complete information, the nondeterministic machine can make the right choice among q branches at each of n decision points. Problems that can be solved in a polynomial number of steps on this imaginary machine are called Nondeterministic-Polynomial, abbreviated NP.

There is a class of NP problems where there are no shortcuts. Following a path to the end gives no hints about the outcome of another path. These problems are, in some sense, maximally difficult. They are called NP-complete problems.

All NP-complete problems are equivalent in the sense that a solution to a particular problem can be polynomially transformed into a solution to any other problem in the class [Cook71]. Thus if a linear solution to any one of the NP-complete problems exists, all of them can be solved in no worse than polynomial time. If an exponential solution is required, all NP-complete problems will require exponential time.

Exponential solutions to NP-complete problems exist for sequential machines. Each path must be followed individually. Since there are an exponential number of paths, the solution has exponential time complexity. On the tree machine, all the paths can be followed simultaneously.

Tree machine algorithms for three NP-complete problems have been designed. The solutions are presented in the next three sections, followed by a discussion of the programming style that the problems encourage. All three solutions follow the same general pattern: a tree is generated that represents all possible solutions for an arbitrary graph of size n. The edges of the particular graph are used to prune the tree and arrive at a solution.

3.2 The Clique Problem

A clique is a subset of nodes and edges of an undirected graph with the property that there is an edge between every pair of nodes in the subset.

> If G is an undirected graph, then C, a subgraph of G, is a clique if and only if for every pair of nodes u and v in C, there is an edge (u,v) in C.

Finding the largest clique in an arbitrary undirected graph is an NP-complete problem, as is determining if a graph contains a clique of a given size.

Before presenting the tree machine solution to the clique problem, let us look at a situation where the problem arises. Heller [Heller79] describes an algorithm for chip planning that takes a graph-theoretic approach. He describes the chip interconnections as a graph, planarizing the graph if possible. If the graph contains any subgraphs that reduce to either of the shapes in figure 3.2.1, the graph can be planarized only by adding extra nodes. Notice that one of the critical shapes is a clique of five nodes. Thus, as the graph is planarized, the question of whether the graph contain a clique of size 5 arises. While this is not an NP-complete problem (it is $O(n^5)$), it is related to the more general NP-complete problem of whether a graph contains a clique of size k.

Figure 3.2.1 Problem Subgraphs in Planarizing Technique.

The tree machine algorithm will generate all of the 2^n possible cliques of the graph. The subgraphs are generated algorithmically as shown in figure 3.2.2. A path from a leaf to the root represents one potential clique, with # meaning no node. Thus, the leftmost path, (1 2 3 4 5 6 ... n), is the whole graph, and the rightmost path, (# # # ...

#), is the empty subgraph. All paths between these two extremes are the other subsets of the nodes of the graph.

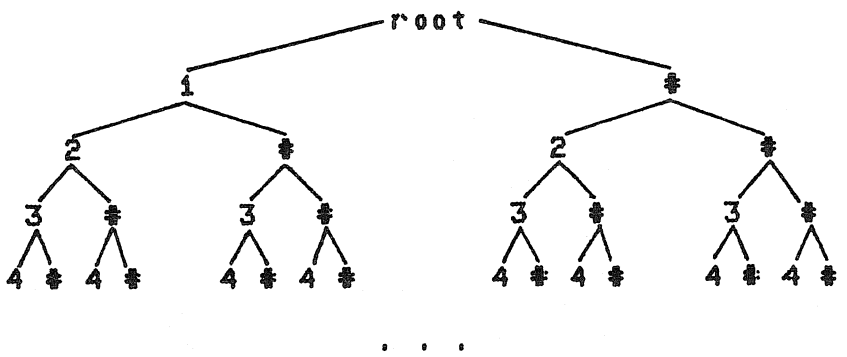


Figure 3.2.2 Algorithmic Generation of Potential Cliques.

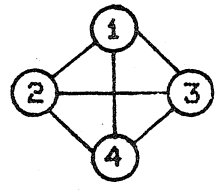
The tree of processors representing potential cliques processes the edges of the graph and (conceptually) prunes the tree down to only genuine cliques. The largest of these is the maximal clique in the graph. This same tree machine can also be used to determine if a graph has a clique of a given size.

The pruning strategy is this: each processor keeps a count of how many edges containing the node it represents pass by. This count is initialized to the number required if this path of the tree represents a clique. As an edge with an endpoint that matches this node is encountered, the count is decremented. A count of zero or less means that this node might be part of a clique. If the edge count in every processor along a given path in the tree is less than or equal to zero, the path represents a clique.

At this point you should be wondering how the number of required edges, that is, the initial edge count, is set. Doesn't it involve foresight about the makeup of each

clique, and thus the complete path in the tree? After all, each node in a clique of size m must have m-1 outgoing edges.

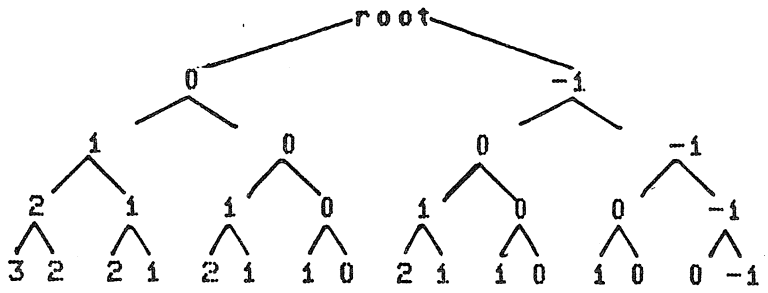
Well, as you may have guessed, there is a trick. True, the first node in a m-clique has m-1 outgoing edges, but one of those goes to the second node. the second node, then, has m-2 outgoing edges that haven't been counted yet. This counting continues with successive nodes having one fewer uncounted edges until we find that m^{th} node has no new edges at all. This edge counting is shown below for a 4-clique.



node 1 edges	node 2 edges	node 3 edges	node 4 edges
(1,2)	(2,3)	(3,4)	
(1,3)	(2,4)		
(1,4)			

Another way of looking at the edges is that there is one edge coming into node 2, two into node 3, and three into node 4 in a 4-clique. This is the way the edge counts are initialized. In any size clique, there are m-1 edges coming in to the m^{th} node in the clique.

The tree below represents the edge counts generated to find cliques in a graph with four nodes. Another way of looking at the edge counts is that each edge count is one less than the number of left branches in the path from the root. Recall from figure 3.2.2 that a count of left branches determines the size of the clique.



The initial edge counts are decremented each time an edge to the node in question is encountered. As edges are sent down the tree by the root, the endpoints are scrutinized by the receiving processors. If neither one matches this processor's node, the edge is sent further down the tree. If either endpoint matches and this processor is part of a clique (that is, is not one of the #'s of figure 3.2.2), the edge becomes a to message, and the edge count in the destination processor will be decremented. If the processor is not part of a clique (rather, is one of the #'s in figure 3.2.2) and either endpoint of the edge message references this processor, the message is thrown into the bit bucket.

Each level in the tree represents the consideration of a new node in the graph. Each path in the tree represents a potential clique. This structure makes it possible to rely purely on local information in identifying the cliques of a graph.

The textual description of the tree is given below. There are three kinds of processors defined: cliqueroot, clique, and cliqueLeaf. The tree has depth n+1. Hence there are n connections in the plan.

```
tree cliqueTree(n):
cliqueRoot ~ clique ;
n-2 ( clique ~ clique ) ;
clique ~ cliqueLeaf.
```

Figure 3.2.3 The Connection Plan for the Clique Tree.

The cliqueRoot node is responsible for initiating the initialization messages (set), for reading the edges from the system bus and passing them down the tree (edge, done), and for reporting the answer (largestClique).

The set message takes three arguments; the first is the node number, the second is a count of the number of non-# processors in this path so far. The third argument is a boolean: true means this node is in the potential clique, false means its not.

The root node has two internal ports, y and n. It expects to receive largestClique messages on these ports, reporting the largest clique found in each subtree. The larger of the two is reported to the system bus as the answer, again in a largestClique message.

The definition of the cliqueRoot processor is given in figure 3.2.4. The program has four sections: declarations, initialization, edge processing, and answer reporting.

```
processor cliqueRoot;
external port bus;
internal port y,n;
integer i,j;  boolean more;

"initialize the tree"
y!set(i,0,true) ;  n!set(i,-i,false) ;

"read the edges from the system bus"
"and pass down the tree"
more := true ;
{ more -->
     [ bus?edge(i,j) --> y,n!edge(i,j)
     | bus?done --> more:=false ;
                    y,n!done
     ]
} ;

"choose largest clique from subtrees"
"and report answer"
y?largestClique(i) ; n?largestClique(j) ;
{ i<j --> i,j := j,i } ;
bus!largestClique(i) .
```

Figure 3.2.4 The cliqueRoot Processor.

The clique processor is replicated to form all of the nodes in the tree between the leaves and the root. It receives a set message along its external port initializing the variables myN, a node number, ecnt, the number of edges from lower numbered nodes to myN that must be in the graph in order for this node to be part of a clique, and inClique, a boolean that represents this node's membership in the subgraph. The clique processor builds set messages from it's own state variables to send to the descendent subtrees only the two internal ports.

The processor will receive edge messages on its external port, terminating with a done message. If one of the endpoints of the edge is this node, the edge message is transformed into a to message containing the other endpoint of the edge. Otherwise, the edge message is passed on

unchanged. To messages provide a mechanism for letting one endpoint of an edge know that the onther endpoint has already seen the edge. When a to message is received by the other endpoint of the edge, ecnt is decremented. If ecnt goes to zero everywhere along a path in the tree machine, that path represents a genuine clique. As with the root, the largestClique messages report the answers.

The code for clique processors is given as figure 3.2.5. Again, it is presented as four paragraphs: declarations, initialization, edge processing, and answer reporting.

```
processor clique;
external port p;
internal port y,n;
integer myC,ecnt,size,i,j;
boolean inClique,more;

"receive initialize message, send to subtrees"
"myN: node number, ecnt: edge count in clique"
p?set(myN,ecnt,inClique) ;
y!set(myN+1,ecnt+1,true) ;
n!set(myN+1,ecnt,false) ;
[ inClique --> size := ecnt+1
| not(inClique) --> size,ecnt := 0,0 ] ;

"process edges: if myN in edge, change to"
"to message. if to contains myN, decr ecnt"
more := true ;
{ more -->
    [ p?edge(i,j) -->
        [ i=myN and inClique --> y,n!to(j)
        | j=myN and inClique --> y,n!to(i)
        | i=myN or j=myN and not(inClique) --> skip
        | i\=myN and j\=myN --> y,n!edge(i,j) ]
    | p?to(i) -->
        [ i=myN and inClique --> ecnt := ecnt-1
        | i=myN and not(inClique) --> skip
        | i>myN --> y,n!to(i) ]
    | p?done --> more := false ; y,n!done ]
};

"if ecnt=0 this could be a clique!"
"choose largest clique and report answer"
y?largestClique(i) ; n?largestClique(j) ;
{ i<j --> i,j := j,i } ;
{ i<size --> i,size := size,i } ;
[ ecnt>0 --> p!largestClique(0)
| ecnt<=0 --> p!largestClique(i)
] .
```

Figure 3.2.5 The clique Processor.

The cliqueLeaf processor makes up the bottom level of the tree. Its program text is simplified because it receives and sends fewer messages.

The leaf processor will receive set messages to initialize myN and ecnt as before. However, as a leaf node has no

internal ports, a cliqueLeaf processor does not manufacture set messages for its descendents.

No edge messages reach the leaves. They have all been converted into to messages. The done message terminated the edge processing phase of the computation.

The answer reporting via largestClique messages is initiated in the leaves and ripples up through the tree to the root.

The program text for the third and final processor definition is given in figure 3.2.6.

```
processor cliqueLeaf;
external port p;
integer myN,ecnt,i,size;
boolean more,inClique;

"get initialization message from above"
"no need to send it on"
"(and nowhere to send it anyway)"
p?set(myN,ecnt,inClique) ;
[ inClique --> size := ecnt+1
| not(inClique) --> size,ecnt := 0,0 ] ;

"process edges - only to and done message"
more := true ;
{ more -->
    [ p?to(i) -->
            [ i=myN and inClique --> ecnt := ecnt+1
            | i\=myN or not(inClique) --> skip
            ]
    | p?done --> more := false
    ]
} ;

"start the answers on their way up the tree"
"if ecnt=0 this is a clique, else no cigar"
[ ecnt<=0 --> p!largestClique(size)
| ecnt>0 --> p!largestClique(0) ] .
```

Figure 3.2.6 The cliqueLeaf Processor.

3.3 The Color Cost Problem

Can the nodes of an arbitrary undirected graph be colored with k colors so that adjacent nodes are different colors? This colorability question [Aho74, p.378] has attracted widespread attention in the past decade as researchers and game players alike have attempted to solve the problem for small values of k [Appel77].

The color-cost problem is a variant of the colorability problem, with k = n, the number of nodes in the graph. Here, of course, colorability is assured: each node can be colored a different color. Each color has a cost

associated with it, and the problem is to find the coloring with minimum cost. It is not necessary to use all of the colors.

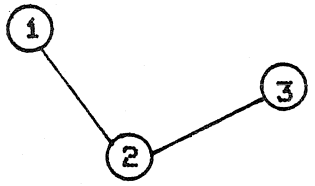
We can easily show that this problem is NP-complete by reformulating it to answer the question of k-colorability posed in the opening sentence of this chapter. If k of the n colors have zerocost and the minimum cost coloring has zero cost, the graph can be colored with k colors. This technique of showing NP-completeness by showing that the problem is equivalent to a known NP-complete problem is the standard way of adding to new problems to the class.

If the undirected graph in question has n nodes, with a pallet of n colors, the number of potential colorings, ignoring adjacency restrictions, is n^n. Thus, a sequential machine will use $O(n^n)$ time to make an exhaustive search for the answer. Given a tree machine with $O(n^n)$ processors, however, all possible colorings can be generated and evaluated in parallel.

The tree machine algorithm is paced by the number of edges in the graph. Thus the time complexity is more rightly O(e), where e is the number of edges. Because e is no larger than the number of edges in a clique of size n, that is, $e <= n(n-1)$, $O(e) <= O(n^2)$.

The coloring tree is built in the same style as the clique tree. Each level of the tree represents the coloring of a particular node in the graph, and each path from a leaf to the root is a unique coloring.

For example, consider the graph of figure 3.3.1, with three nodes and three colors. The tree machine for every graph of size three is given in figure 3.3.2. It has four levels, has a fanout of three from each non-leaf node, and is built using one colorRoot, twelve coloring, and twenty-seven colorLeaf processors.



color	cost
blue	0
green	1
red	2

Figure 3.3.1 A Sample Graph for the Color-Cost Problem.

Each of the 27 possible colorings are generated as described and shown in figure 3.3.2. As the edges are processed, various colorings are ruled out because they violate the adjacency criterion. If two nodes of the graph are adjacent, that is, if they share an edge, then they must be colored differently.

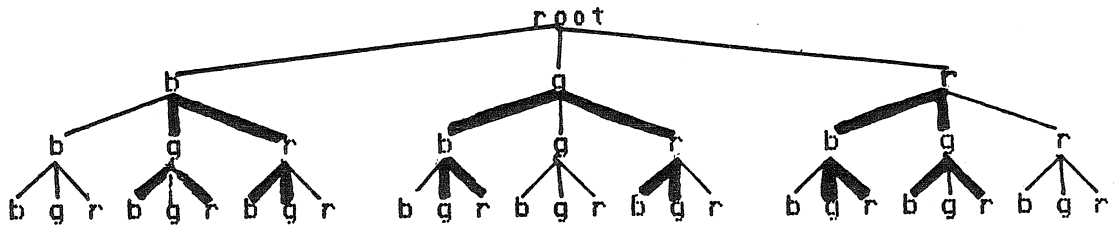


Figure 3.3.2 Solution to Sample Coloring Problem.

The edges are used to invalidate colorings that ignore the adjacency rule. Adjacent nodes must be different colors. Each processor has a validity flag, initially set to true. This flag indicated whether or not the coloring of this node is legal. Edges are sent through the tree by the root and are scrutinized by the processors. Each time a processor finds an edge with one endpoint matching the node it represents, the edge is converted into an adjacency message directed at the other endpoint. The message contains the color chosen to color the first endpoint. If the target processor has colored its node the same color, the coloring is invalidated by setting the valid flag to false. A valid coloring is one in which every processor along a path from a leaf to the root has the validity flag set to true after all the edges have been processed.

When both edges of the example graph have been processed, there remain 12 valid colorings, shown as double-lined paths in figure 3.3.2. Of these, the minimum cost coloring is the left-most one: blue, green, blue.

The algorithm utilizes three different kinds of processors: colorRoot, coloring, and colorLeaf processors. The tree is n+1 levels deep and branches n ways from each node. Thus,

there are n^n leaves in the tree, and each path from a leaf to the root represents one potential coloring of the graph. The connection plan is given in figure 3.3.3.

```
tree colorTree(n):
colorRoot ~ coloring ;
n-3 ( coloring ~ coloring ) ;
coloring ~ colorLeaf.
```

Figure 3.3.3 The Connection Plan for the Color-Cost Tree.

The colorRoot processor performs initialization and system input/output just as the corresponding processor in the clique algorithm did. The set message i initiated by the root and carries two pieces of information. The first argument is the number of the node in the graph that the receiving processor will represent. The second argument tells the receiver which color it will represent. After the set message is sent, the color costs are read from the system bus and broadcast throughout the tree.

The edge and done messages are part of the edge processing phase of the computation. Arcs of the graph are read from the system bus and sent out through the internal ports of the root node. A done message follows the last edge.

When edge processing is complete, the colorRoot is ready to receive a candidate for minimum cost coloring from each of its n internal ports, and to choose the final answer from among them.

The definition for the colorRoot is given in figure 3.3.4. As with the program text for the clique algorithm, the code is paragraphed into each phase: declaration, initialization, edge processing, and answer reporting.

```
processor colorRoot(n);
external port bus;
internal port v(i:n);
integer i,j,c; boolean more;

"initialize subtrees with costs and colors"
i:=1 ; { i<=n --> v(i)!set(i,i) ; i:=i+1 };
i,j := 1,0 ;
{ i<=n --> bus?cost(c) ;
           { c<j --> j:=c } ;
           v(*)!cost(c) ; i:=i+1 } ;
c:=n*j ;

"process the edges"
"read from system bus"
"pass to subtrees"
more:=true ;
{ more -->
    [ bus?edge(i,j) --> v(*)!edge(i,j)
    | bus?done --> more:=false ; v(*)!done ]
} ;

"choose least of n answers and report it"
i:=1 ; { i<=n --> v(i)?answer(j) ;
                  { j<c --> c:=j } ; i:=i+1 } ;
bus!answer(c) .
```

Figure 3.3.4 The colorRoot Processor.

The coloring processor resides between the root and the leaves in the tree machine solution. It has one external port and n internal ports. From the external port it receives a set message to initialize the variables myN, the node number, and myC, the color number. As the n cost messages are received, the myCth one is picked out to initialize myCost. A set message is manufactured and sent out each internal port. The costs are passed on as well.

In the edge processing phase, edge messages are scanned to see if myN matches either endpoint. If so, the message is transformed into an adj message carrying the other endpoint

and the color of this node. Otherwise, the edge is passed on unchanged. If an adj message is received with the endpoint matching myN and the color the same as myC, this coloring is not allowed. This invalidity is recorded in the boolean valid. The edge processing phase end with the reception of a done message.

In the answer reporting phase, the internal ports are interrogated for the cost to date of each coloring. An invalid coloring has a very large cost, called inf for infinite. The smallest valid cost among the subtrees is added to myCost and passed up the tree. A coloring is valid only if every processor on the path from the leaf to the root has valid=true. Thus, if all answer messages from the internal ports are inf or if valid in this node is false, an inf is reported.

The complete processor program text for the coloring processor is given in figure 3.3.5.

```
processor coloring(n);
external port p;
internal port v(i:n);
integer myN,myC,myCost,i,j,c;  boolean valid,more;

"read init  message, send some to subtrees"
"myN: node number, myC: color, myCost: cost"
p?set(myN,myC) ; valid:=true ;
i:=1; { i<=n --> v(i)!set(myN+1,i) ; i:=i+1 };
i:=1 ;
{ i<=n --> p?cost(c);
        [ myC=i --> myCost:=c | myC/=i --> skip ] ;
        v(*)!cost(c) ; i:=i+1
};

"process edges: if myN in edge, make adj message."
"if adj(myN,myC), mark as invalid coloring"
more := true ;
{ more -->
    [ p?edge(i,j) -->
        [ i=myN --> v(*)!adj(j,myC)
        | j=myN --> v(*)!adj(i,myC)
        | i>myN and j>myN --> v(*)!edge(i,j) ]
    | p?adj(i,j) -->
        [ i=myN and j=myC --> valid:=false
        | i=myN and j\=myC --> skip
        | i>myN --> v(*)!adj(i,j) ]
    | p?done --> more:=false ; v(*)!done
    ]
} ;

"report answer"
"pick least cost coloring from below"
"if valid add in myCost"
"inf means invalid coloring"
i,c:=1,inf ;
{ i<=n --> v(i)?answer(j) ;
           { j<c --> c:=j } ; i:=i+1 } ;
[ c=inf or not(valid) --> p!answer(inf)
| c<inf and valid --> p!answer(c+myCost)
] .
```

Figure 3.3.5 The coloring Processor.

The leaf processor, colorLeaf, is a simplified version of the coloring processor. Since it has no internal ports, the colorLeaf receives set and cost messages with no need to forward them. In the edge processing phase of the computation, only adj messages will be received by

colorLeaf. All of the edge messages have been converted to adj messages higher up in the tree. As with the clique algorithm, the colorLeaf's claim to fame is that it initiates the answer reporting as soon as the done message is received. See figure 3.3.6 for the definition.

```
processor colorLeaf(n);
external port p;
integer myN,myC,myCost,i,c;  boolean valid,more;

"get initialization message from above"
p?set(myN,myC) ; valid,i := true,1;
{ i<=n --> p?cost(c);
           [ myC=i --> myCost:=c
           | myC\=i --> skip ] ;
           i:=i+1 } ;

"process edges - only adj and done messages"
more:=true ;
{ more -->
    [ p?adj(i,c) -->
        [ i=myN and c=myC --> valid:=false
        | i=myN and c\=myC --> skip
        | i\=myN --> skip
        ]
    | p?done --> more:=false
    ]
} ;

"Initiate answer reporting"
[ valid --> p!answer(myCost)
| not(valid) --> p!answer(inf) ] .
```

Figure 3.3.6 The colorLeaf Processor.

3.4 The Travelling Salesman Problem

One of the classic NP-complete problems is that of the travelling salesman. Given an undirected graph, with weights on each edge, find a path that visits each vertex exactly once, returns to the starting point, and minimizes

the sum of the weights along the path.

This problem is often casually defined in terms of the cities (vertices) and highways (edges) of a roadmap. That is why it is called the travelling salesman problem. It differs from the two previous problems in that it grows as n! rather than exponentially.

The tree machine solution mirrors the n! growth. The fanout is different at each level in the tree. The root has n-1 descendents, each of which has n-2 descendents, each of which has n-3 descendents, and so on. The last but one level has a fanout of 1, to the leaf nodes. The tree that arises from n=5 is given in figure 3.4.1.

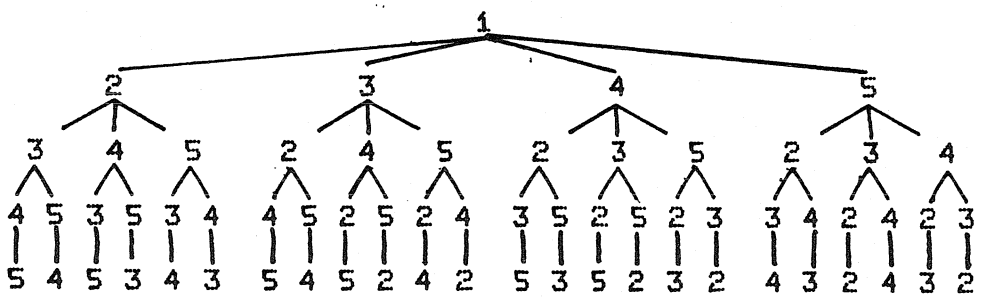


Figure 3.4.1 Sample Salesman Tree, n=5.

The travelling salesman problem maps onto the tree machine very much like the two previous NP-complete problems. The tree represents the complete set of possible cycles in a graph of n nodes. The i^{th} level in the tree represents the addition of the i^{th} node to the path. Each path from a leaf to the root in the tree represents a potential cycle, that is, a potential solution.

The processors in the tree are assigned graph nodes in the following way. The graph nodes are broadcast through the tree as a stream of node messages. Each processor has been told to notice the node occupying a particular position in the stream. The processor will remove that node from the stream, but pass all the rest on. This means, incidently, that the stream gets shorter as it approaches the leaves of the tree machine.

Each processor initializes its descendents. The notice message carries an integer argument that indicates the position of the node in the stream that this processor will represent. The notice messages are generated by way of a simple FOR loop in each processor. The first descendent will pick off the first node in the stream, the second will remove the second, and so on. Since the "noticed" node is removed from the stream, the stream is different for each processor in the tree.

The edges of the graph are thrown at this tree in order to prune this exhaustive set of cycles down to the routes that are legal in the context of the graph. Edges are broadcast through the tree, examined by the processors, and converted into distances as endpoints of the edges are matched. By the time all of the edges have been sent through the tree a legal cycle through the graph has been identified. Every path from leaf to root that represents a cycle has positive distance stored in every processor along the path.

The main subtlety to the problem is the fact that the path must be a cycle. It must end where it began. The salesmanLeaf processor holds state information about two edges: the one from its parent to itself, and the one from

itself to the starting point. The root not only passes edges on as is, but if the edg is to or from the starting point, it is duplicated as an end message as well. Thus, the cycle property is maintained in the final solution.

An example graph is given in figure 3.4.2. The distances are shown in along the edges of the graph, and are given in the Los Angeles measure of distance: minutes.

The solution tree for the travelling salesman problem is shown in figure 3.4.3. The eight double-lined paths are the legal cycles in the graph. The fastest route from Pasadena to Pasadena by way of the Hollywood, Santa Monica, Long Beach, and Pasadena freeways, at least according to this map, takes an hour and forty-five minutes. Either of these two (mirror image) routes are fastest: (Pasadena, Los Angeles, Long Beach, Santa Monica, Hollywood, Pasadena) or (Pasadena, Hollywood, Santa Monica, Long Beach, Los Angeles, Pasadena).

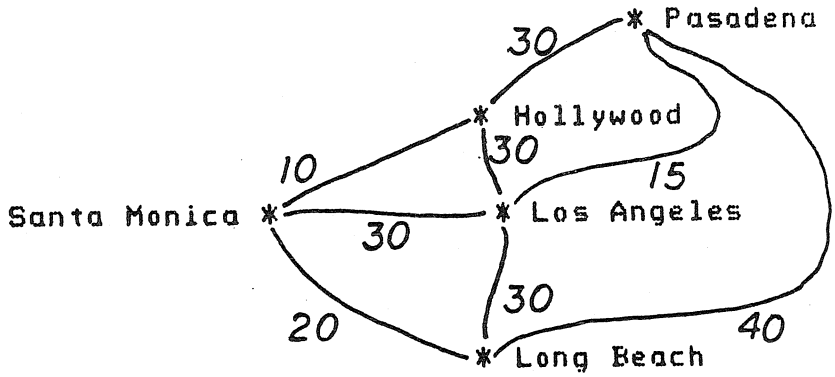
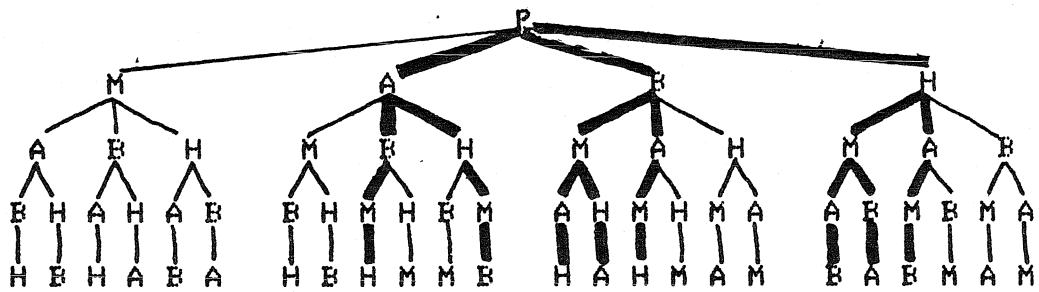


Figure 3.4.2 Travelling Salesman Example.



P: Pasadena H: Hollywood M: Santa Monica

A: Los Angeles B: Long Beach

Figure 3.4.3 Solution Tree for Example Problem.

The travelling salesman tree has a different fanout on each level of the tree. That fanout is related to the level of the tree the processor resides on. In order to represent the varying fanouts in a connection plan, a new piece of notation is introduced. The symbol # is used to mean level number in the tree, much like * or . is used to represent the current value of the program counter in traditional assembly languages. The root is level 0, so the row of salesmanLeaf's in Figure 3.4.1 has #=4. The # appears only

in connection plans, never in a processor definition.

The solution tree for the travelling salesman problem is defined in figure 3.4.4. As with the other NP-complete problems, there are three types of processors, the root, the leaves, and the ones in between. Notice that the fanout of the salesman processors is described by an expression that changes with the level number.

```
tree salesmanTree(n):
    salesmanRoot(v(i:n)) ~ salesman ;
    n-3 ( salesman(v(i:n-#)) ~ salesman ) ;
    salesman(v(i)) ~ salesmanLeaf .
```

Figure 3.4.4 The Travelling Salesman Tree.

Each of the three processors perform much of the same functions as the corresponding ones in the clique and coloring problems. The root, salesmanRoot, initiates the initialization phase, reads the edges from the system bus, and reports the answer. The middle processor, salesman, continues the initialization, processes and converts the edges, and passes answers up. The leaf processor, salesmanLeaf, is a sink for initialization and edge messages, and the source of the answer messages. The weights are loaded via distance messages in the same way that costs were initialized in the coloring problem. See figures 3.4.5, 3.4.6, and 3.4.7 for the program text.

```
processor salesmanRoot(n);
external port bus;
internal port v(i:n-i);
integer i,j,d;  boolean more;

"initialize subtree"
i:=i; { i<n --> v(i)!notice(i); i:=i+i };
i:=i; { i<n --> v(*)!node(i); i:=i+i };

"process edges"
"read from bus and pass to subtrees"
"if edge contains starting point (0)"
"make an end message too."
more := true;
{ more -->
    [ bus?edge(i,j,d) -->
            [ i=0 --> v(*)!distance(j,d);
                      v(*)!end(j,d)
            | j=0 --> v(*)!distance(i,d);
                      v(*)!end(i,d)
            | i\=0 and j\=0 --> v(*)!edge(i,j,d)
            ]
    | bus?done --> more := false ; v(*)!done
    ]
};

"report answer"
"choose shortest path from n-i subtrees"
i,d := i,inf ;
{ i<n --> v(i)?length(d,j) ;
          { j\=0 and d>j --> d:=j } ;
          i := i+i };
[ d=inf --> bus!noPath
| d<inf --> bus!shortestDistance(d) ] .
```

Figure 3.4.5 The salesmanRoot Processor.

```
processor salesman(m);
external port p;
internal port v(1:m);
integer myN,myD,i,j,d;  boolean more;

"read initialization messge to set myN"
myD:=0 ; p?notice(j) ;
i:=1; { i<=n --> v(i)!notice(i) ; i:=i+1 } ;
i:=1 ; { i<=m+1 -->
                  p?node(d) ;
                  [ i=j --> myN:=d
                  | i\=j --> v(*)!node(d) ] ;
                  i:=i+1 } ;

"process edges: if myN in edge, make distance"
"if myN in distance, set myD; if end, pass it on."
more := true ;
{ more -->
    [ p?edge(i,j,d) -->
         [ i=myN --> v(*)!distance(j,d)
         | j=myN --> v(*)!distance(i,d)
         | i\=myN and j\=myN --> v(*)!edge(i,j,d) ]
    | p?distance(i,d) -->
         [ i=myN --> myD:=d
         | i\=myN --> skip ]
    | p?end(i,d) --> v(*)!end(i,d)
    | p?done --> more:=false ; v(*)!done ]
};

"report answer: choose least distance among"
"subtrees, add myD, and pass up"
"myD=0 or d=inf means it is not a legal path"
i,d := i,inf ;
{ i<=m --> v(i)?length(j) ;
           { j\=0 and d>j --> d:=j } ;
           i:=i+1 };
[ d=inf or myD=0 --> p!length(0)
| d<inf and myD>0 --> p!length(d+myD)
] .
```

Figure 3.4.6 The salesman Processor.

```
processor salesmanleaf;
external port p;
integer myN,myD,i,d,cycle;  boolean more;

"initialize myN"
p?notice(myN) ; p?node(myN) ; myD,cycle := 0,0 ;

"process edges"
"if distance contains myN, set myD"
"if end contains myN, set cycle"
more := true;
{ more -->
    [ p?distance(i,d) --> [ i=myN --> myD:=d
                          ! i\=myN --> skip ]
    ! p?end(i,d) --> [ i=myN --> cycle:=d
                     ! i\=myN --> skip ]
    ! p?done --> more := false
    ]
} ;

"start answer trickling up the tree"
"both cycle and myD must be nonzero for valid path"
[ myD=0 or cycle=0 --> p!length(0)
! myD>0 and cycle>0 --> p!length(myD+cycle)
] .
```

Figure 3.4.7 The salesmanLeaf Processor.

3.5 NP-Complete Problems and the Tree Machine

In the previous sections, three NP-complete problems were solved on the tree machine. The three problems, though all intractable, exhibit different growth patterns, and the tree machine algorithms reflect this and other problem-specific characteristics. The similarities in solution and style are dealt with here.

All three tree machine algorithms require $O(n^2)$ time, and all three are paced by the number of arcs in the graph.

The paths in the tree machine are used to represent possible paths in the graph. Thus, state is hidden in the

connection plan of the tree. This is a departure from earlier solutions to these problems [Browning79c], and eliminates the need for problem-size dependent storage in the processors. There is another benefit as well. Each subtree has the same appearance, excepting special root functions, as a complete tree for a problem of reduced size. This is critical, since the goal of the programming exercise is to make the size of the problem influence only the number of processors in the tree, not the shape of the tree or the size of the individual processors.

In a single processor environment, space is treated as the critical resource when attempting to solve the NP-complete problems. Each of the exponential number of cases is treated individually. Thus the time required to arrive at a solution is exponential in n.

In the upcoming VLSI environment, the tradeoff can be made the other way. Time can be minimized, and space (processors) treated as an unlimited resource. The tree machine is a machine from that mold. The NP-complete problems, while still exploding exponentially in space, are solved by examining all potential solutions in parallel.

It must be pointed out that the reduction to polynomial time is realized only if there are enough processors in the tree. Otherwise, the algorithm must be simulated at some point and is still exponential. In that case, only the time constant is reduced.

There are several algorithms that yield approximate solutions to these problems in polynomial time. It might be interesting to map some of these approximation

techniques onto the tree for use when there aren't enough processors to find an exact solution.

Chapter Four

Matrix Manipulation

In this chapter two algorithms for manipulating matrices are presented: matrix multiplication and matrix inversion. In addition, several related problems are discussed: chain multiplication, solving systems of equations, and LU decomposition. The chapter ends with a discussion of the appropriateness of solving matrix problems on the tree machine.

One might wonder if matrix problems are appropriate problems for a tree machine. They seem to more naturally map onto an array machine. There are array machines that do matrix operations, and they multiply matrices very well. But matrix inversion and related problems often require that the data be reordered, that is, rows or columns interchanged. Array machines have great difficulty with this operation, called pivoting, because their efficiency relies so much on the physical order of the data. The more general structure of the tree machine is well suited to pivoting, as we will see later.

4.1 Matrix Multiplication

Consider the problem of multiplying two nxn matrices. Let A, B, and C be nxn matrices such that C=AxB. An element of C is formed by a series of multiplications and additions according to the formula given below. Each element of c is

computed from a row of A, and a column of B.

$$c_{ij} = \sum_{k=i}^{n} a_{ik}b_{kj} \quad i,j=1,2,\ldots,n$$

The tree is composed of three types of processor: root, row, and element processors. It is an n-ary tree and is described by the connectivity statements in figure 4.1.1.

```
tree MatMult(n):
        root ~ row;
        row ~ element.
```

Figure 4.1.1 Matrix Multiplication Processor Tree.

The multiplicand matrix, A, is loaded into the tree, one number per element processor, in row order. That is, the n elements of the first row of the matrix occupy the element processors connected to the internal ports of the first row processor, and so on. For example, if n=3, the tree is loaded as in figure 4.1.2.

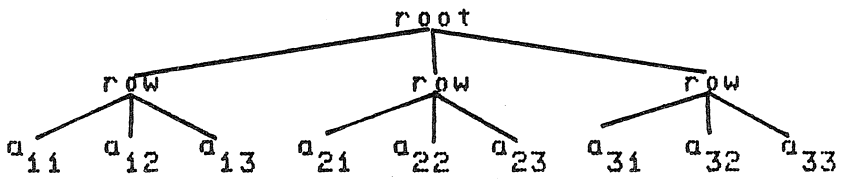


Figure 4.1.2 Sample Loaded Matrix Multiplication Tree.

The multiplier matrix, B, is available to the root in column-major order as $(b_{11},b_{21},b_{31},\ldots,b_{n1},b_{12},\ldots,b_{nn})$. Each element of B is sent to all of the row processors simultaneously. Each row processor multiplies it by the appropriate element of A and accumulates the sum. When the n elements of a column of B have been received by the root and sent to the rows, the first column of the answer matrix C is available, one element from each row processor.

The processor definitions are given in Figures 4.1.3 (the root), 4.1.4 (the row), and 4.1.5 (the element). Each is accompanied by a discussion of the salient features of the program text.

The root is largely a traffic director. The tree is first initialized with the elements of the multiplicand. The first loop in the program text does that. It remains to accept a column of the multiplier, direct it to the rows, and receive a column of the product.

The root is prepared to receive two types of messages from the environment. Load messages cause the multiplicand matrix to be loaded. Mult messages signal the availability of the multiplier matrix. The root will provide the answer to the environment via the product message. Rememeber that

the root expects the multiplicand in row major order. The multiplier and product are in column major order.

```
processor root(n);
external port bus;
internal port r(i:n);
integer i,j; real a;

"load in the multiplicand"
i:=1 ;
{ i<=n --> j:=1 ;
           { j<=n --> bus?load(a) ;
                      r(i)!load(a) ; j
                      :=j+1 } ;
           i:=i+1 };

i:=1 ;
{ i<=n -->
     "get a column of the multiplier"
     j:=1 ; { j<=n --> bus?mult(a) ;
                       r(*)!mult(a) ; j:=j+1 } ;
     "get a column of the product"
     j:=1 ; { j<=n --> r(j)?product(a) ;
                       bus!product(a) ; j:=j+1 } ;
     i:=i+1
} .
```

Figure 4.1.3 The root Processor.

The second layer of the tree is made up of row processors. The row receives the initial values of the row of the multiplicand that it represents from the root, and doles them out to the appropriate element processors. As multiplier elements are received from the root, the products are calculated and accumulated until each element in the row has contributed to the sum. The sum, an element in the product, is sent to the root.

The row will accept load and mult messages from its external port. It will send product messages in return. Outgoing traffic on the internal ports consists of load and

mult messages. The row will accept product messages from its internal ports.

```
processor row(n);
external port p;
internal port e(i:n);
integer i,j;  real a,s;

"load multiplicand"
i:=1 ; { i<=n --> p?load(a) ;
                  e(i)!load(a) ; i:=i+1 } ;

i:=1 ;
{ i<=n -->
    s,j:=0,1;
    "get an element of the multiplier"
    "form product with multiplicand element"
    "accumulate a column's worth and report sum"
    { j<=n --> p?mult(a) ; e(j)!mult(a) ;
               e(j)?product(a) ;
               s,j:=s+a,j+1 } ;
    p!product(s) ; i:=i+1
} .
```

Figure 4.1.4 The row Processor.

The third kind of processor, element, is the simplest. It waits for messages on its external port. If it receives a load message, variable a is inititalized. A mult message causes a multiplication; the answer is returned in a product message.

```
processor element;
external port p;
real a,e;

"this guy only knows how to load and multiply"
( p?load(a) --> skip
| p?mult(e) --> p!product(a*e)
) .
```

Figure 4.1.5 The element Processor.

On a sequential processor, matrix multiplication takes $O(n^3)$ time. The algorithm described above requires $O(n^2)$. There are $O(n^2)$ data elements involved in the computation and each must be loaded into or unloaded from the tree individually.

Since the tree is a recursively defined data structure, one might wonder if the divide-and-conquer technique for matrix multiplication [Aho74] is a better match of algorithm to machine. In fact, it is not. The divide-and-conquer method calls for recursively quartering the matrix until each piece is small enough to be multiplied directly, then reassembling the partial products into the whole answer. It requires $O(n^3)$ processors, an order of n more than the algorithm described above. Both take $O(n^2)$ time. The tree machine divide-and-conquer implementation is given as one of a collection of tree machine algorithms presented in Mead and Conway's Introduction to VLSI Systems [Browning79c].

With a slight modification, the matrix multiplication algorithm can be extended to multiply a series of matrices, that is, to compute

$$C = \prod_{i=1}^{m} A_i$$ where C and the A_i's are nxn matrices.

If, instead of unloading the elements of the product as they are generated, they are stored as a second value in the element processors, the tree is initialized for another multiplication. By storing two values in each element, and alternating between the two, a chain of m multiplications can be performed in $O(mn^2)$ time.

4.2 Matrix Inversion

Here is a simple way to invert a non-singular nxn matrix. First, the matrix is augmented with an n x n identity matrix to form a nx2n matrix as in Figure 4.2.1. Elementary row operations, described below, are applied to this augmented matrix until the left half (the original matrix) has been transformed into an identity matrix. The right half of the augmented matrix is now the inverse of the original matrix. Franklin [Franklin68] offers proof that this method will work.

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} & 1 & 0 & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & a_{2n} & 0 & 1 & 0 & \cdots & 0 \\ a_{31} & a_{32} & \cdots & a_{3n} & 0 & 0 & 1 & \cdots & 0 \\ & & & & & & & & \\ a_{n1} & a_{n2} & \cdots & a_{nn} & 0 & 0 & 0 & \cdots & 1 \end{pmatrix}$$

Figure 4.2.1 An Augmented Matrix

There are three elementary row operations required. They are 1) interchanging two rows, 2) dividing an entire row by a scalar, and 3) adding a multiple of one row to another. The next few paragraphs explain the algorithm in terms of these three operations.

The row interchange operation is used to minimize the computational errors due to fixed precision. The element with the largest absolute value on or below the diagonal of the selected column is chosen as the pivot element. The row that contains the pivot element is interchanged with the row that contains the diagonal element. In later discussions the selection of the pivot element is called pivoting. It is considered separately from the interchange step.

An identity matrix has ones on the diagonal and zeros elsewhere. The scalar divide operation is used to transform the diagonal elements of the matrix into ones. Each element of a given row is divided by the value of the diagonal element. Notice that this operation requires that the matrix is non-singular. The diagonal element, after pivoting, must not be zero.

The third row operation is adding a multiple of one row to another row. This operation is used to make every non-diagonal element zero. It is called elimination.

This method of forming the inverse matrix takes $O(n^3)$ operations and $O(n^2)$ storage cells on a sequential processor.

4.2.1 The Tree Machine Implementation

The algorithm described in the previous section maps very nicely onto a tree machine. Given processors that can store and manipulate a single element of the matrix, the time complexity can be reduced to $O(n^2)$ by taking advantage of the parallelism offered by the tree architecture. The time-space product of the algorithm is $O(n^5)$ on a single processor machine. It is reduced to $O(n^4)$ on the tree machine.

The following paragraphs describe the tree machine algorithm. The discussion assumes that the tree contains an augmented matrix, with an element of the original matrix and the corresponding element of the identity matrix in each of the leaf nodes.

The logical tree that will invert an nxn matrix has three different kinds of processors in it, and two different fanouts. The root processor is the control center. It issues the messages that cause pivot, interchange, and divide operations to happen. And it oversees the elimination step as well. There is no computational activity performed in the root, however, and it has no

knowledge of the actual matrix element values.

The row processors, one for each row in the matrix, direct the actual computations on the matrix elements. They select the appropriate multiplier for the elimination and divide operations, and send their pivot candidate to the root. They are directly responsible for the n element processors in each row, and as such, have a fanout of n.

The element processors are the only processors in the tree that do arithmetic. Each element processor is responsible for one entry in the original matrix, and the corresponding entry in the identity matrix. The processor will divide its data by a given scalar, and will add the product of two given scalars to its element. These are the actions required to perform the divide and elimination steps. Note that pivot and interchange do not require arithmetic, and are handled higher up the tree. The elements are the leaves of the tree.

Figure 4.2.2 is a concise textual description of the interconnect pattern between the three flavors of processors, root, row, and element. Later figures detail the programs that will run in each processor.

```
tree invert(n):
    root ~ row ;
    row ~ element .
```

Figure 4.2.2 The Matrix Inversion Processor Tree.

4.2.2 Identifying the Parallelism

Communication between processors is the costly part of any tree machine algorithm. Which of the row operations

require movement of whole rows of data?

On the surface, the interchange operation would seem to require the physical movement of two rows. This is not the case, however, if there is no dependence on a particular physical ordering of the data. If each row knows which row it is, interchanging two requires only that the identification change. Because the tree machine algorithm treats each row as an independent entity, the interchange operation requires the movement of two pieces of data, the new row assignments, not 4n (two rows). Linked list implementations of matrix manipulation algorithms on sequential processors also use this method to interchange two rows. The interchange order goes from the root to the two row processors in parallel, an $O(\log_2 n)$ operation.

How about pivoting? Is there a fast way to find the largest element on or below the diagonal in a given column of the matrix? One solution is examine the up to n elements and choose the largest one. This is a linear operation. In a later section we see that the pivot element can be chosen in $O(\log_2 n)$ comparison operations by restructuring the tree slightly.

Scalar divide of a row of the matrix is another $O(\log_2 n)$ operation. Furthermore, all n rows can do it in parallel. Each row finds its diagonal element and asks each element processor to divide its element by the diagonal value. There is no need for cross communication between rows.

The elimination operation is the only row operation that requires the movement of an entire row of the matrix. Once again, the cost of the operation can be minimized by

letting all the other rows act on the data in parallel. That is, the actual row will be broadcast to all the other row processors. Each row will calculate the necessary multiplier to eliminate their entry in the column in question. The communication of the row can be pipelined so that the operation is O(n). This is a lower bound, since O(n) elements must be moved.

None of the row operations are worse than O(n). The transformation of a given column of the matrix into the appropriate column of the identity matrix, then, is a linear operation. Inverting the matrix takes $O(n^2)$ steps. Since the loading and unloading of the matrix is also $O(n^2)$, the lower bound on time complexity is achieved. Table 4.2.3 shows the time costs for the various operations on one column and on the entire matrix. The entries in the table are counts of communications, with each message taking constant time to go from one level to the next in the tree machine.

operation	one column	n columns
Pivot	$2\log_2 n$	$2n\log_2 n$
Interchange	$\log_2 n$	$n\log_2 n$
Eliminate	$4\log_2 n + 2n$	$4n\log_2 n + 2n^2$
Divide		$2\log_2 n$
Total		$2\log_2 n + 7n\log_2 n + 2n^2$

Table 4.2.3 Time Complexity for Inverting an nxn Matrix.

4.2.3.1 The Programs

I will begin with the root processor, shown in Figure 4.2.4. I assume that the root is connected to an environment that will supply the root with the matrix via

load messages, and will accept the inverted matrix as answer messages.

The root processor begins by loading the matrix. Then, for each column in the matrix, the pivot element is chosen, the rows are interchanged, and other column entries are eliminated. When each column has been examined, the divide order is given to all rows in parallel.

The pivot element is identified by receiving pivot messages from all the row processors. The interchange message affects the interchange operation. The other column entries are eliminated via the eliminate message, which broadcasts the row elements throughout the tree. When all n columns have been treated, answer messages arrive at the root and are transfered to the system bus in the next n^2 time steps.

The root processor code is written without an enclosing loop. It can load, invert, and unload one matrix before halting. If repetitious execution is desired, a loop can be added.

```
processor  root(n);
external port bus; internal port r(i:n);
integer i,j,k,m,c; real a,b;

"set initial row assignments"
i:=1 ; [ i<=n --> r(i)!setRow(i) ; i:=i+1 ] ;

"load the matrix"
i:=1 ;  [i<=n -->
            j:=1 ;  [j<=n --> bus?load(a) ;
                              r(i)!load(a) ;
                              j:=j+1 ] ;
            i:=i+1 ] ;

"for each column ... "
i:=1;  [i<=n -->
            "select pivot"
            j,b:=i,0.0 ;
            [ j<=n --> r(j)?pivot(a,k) ;
                       [ b<a --> b,m:=a,k ] ;
                       j:=j+1 ] ;

            "interchange rows i and m"
            r(*)!interchange(i,m);

            "elimination step"
            "get row values and broadcast"
            j:=i;
            [j<=n --> r(*)?eliminate(a,b) ;
                      r(*)!eliminate(a,b) ;
                      j:=j+1 ] ;
            i:=i+1 ] ;

"unload the answer"
i:=1; [i<=n*n --> r(*)?answer(a,m,c) ;
                  bus!answer(a,m,c) ; i:=i+1 ] .
```

Figure 4.2.4 The root Processor.

The second flavor of processor, the row, is by far the most complicated one. SetRow initializes myRow. Load is used to load in the matrix, and depending on myRow, initialize the identity matrix as well. Then, for each column, we see the familiar pivot, interchange, and elimination steps. The scalar divide operation is triggered by the completion of the column massaging. Once the divide is complete, it

remains only to unload the answer, appending row and column indicators to the value received from the elements.

```
processor row(n);
external port p; internal port e(i:n);
integer myRow,i,j,r; real a,b,f;

"initialize myRow" p?setRow(myRow);

"load the augmented matrix"  i:=1 ;
{ i<=n --> p?load(a) ;
        [ i=myRow --> e(i)!load(a,1)
        | i/=myRow --> e(i)!load(a,0) ] ;
        i:=i+1 } ;

"for each column ..."  i:=1 ;
{ i<=n -->
    "send up pivot element"
    [ myRow<i --> p!pivot(0)
    | myRow>=i --> e(i)?absVal(a) ; p!pivot(a) ] ;
    "interchange"
    p?interchange(j,r) ;
    [ j=myRow --> myRow:=r
    | r=myRow --> myRow:=j
    | j/=myRow and r/=myRow --> skip ] ;
    "elimination"
    [ i=myRow -->
        j:=i; { j<=n --> e(j)?bothVal(a,f) ;
                         p!eliminate(a,f) ; j:=j+1 }
    | i/=myRow --> skip ] ; j:=1 ;
    {j<=n --> p?eliminate(a,b); e(i)!elimValue(a,b);
        [j=i --> f:=a | j/=i --> skip]; j:=j+1 };
    e?val(a) ;
    [ i=myRow --> e(*)!eliminate(0)
    | i/=myRow --> e(*)!eliminate(a/f) ] ;
    i:=i+1 } ;

"divide en mass"
e(myRow)?val(a) ; e(*)!divide(a) ;

"unload the answer"
i:=1 ; { i<=1 --> e(i)?answer(a) ;
                  p!answer(a,myRow,i) ; i:=i+1 } .
```

Figure 4.2.5 The row Processor.

The simplest processor is the element at the leaves of the tree. The actions that result from the eight messages the element processor will accept are easily understood.

```
processor element;
external port p;
real a,i,r,s,t;

{ p?load(a,i) --> skip
| p!absVal(abs(a)) --> skip
| p!bothVal(a,i) --> skip
| p?elimValue(s,t) --> skip
| p?eliminate(r) --> a:=a-s*r ; i:=i-t*r
| p!val(a) --> skip
| p?divide(r) --> a:=a/r; i:=i/r
| p!answer(i) --> skip
} .
```

Figure 4.2.6 The element Processor.

4.2.4 Column vs Row Organization

One might think that the elimination step could be simplified by storing the elements by column rather than by row, as described above. Instead of row processors, define column supervisors, and load the matrix in column major order. The three operations on the columns to produce the inverse are analogous to those on the rows. We will look at the column operations individually to see if the time cost improves.

The pivot selection involves only one column and the subtree of element processors below it, but the interchange operation affects every column in the tree. Thus, pivot selection has the same cost and interchange is more expensive than in the row organization. The pivoting cannot be done in parallel in all the columns. The elimination of a column affects the choice of the next pivot element because it changes all of the elements in the matrix

The elimination step no longer requires the broadcasting of the 2n elements in a row. Instead, the n factors that are used to eliminate the column must be transmitted throughout the tree. Thus, the cost of the elimination step is reduced by a factor of two because of the column organization.

The divide step becomes costly in a column organization because each row is scattered throughout the tree. The rows can no longer be divided in parallel They must be treated sequentially instead. However, pipelining can be employed to overlap the broadcasting of the n scalars to all of the columns.

The time complexity for each operation is given below, as well as an expression for the overall complexity. Notice that column organization, like row organization, results in an $O(n^2)$ algorithm, albeit with a smaller constant. The row organization has the advantage of being more intuitive.

operation	one column	n columns
Pivot	$2\log_2 n$	$2n\log_2 n$
Interchange	$3\log_2 n$	$3n\log_2 n$
Eliminate	$4\log_2 n + n$	$4n\log_2 n + n^2$
Divide		$6\log_2 n + n$
Total	$6\log_2 n + n$ +	$9n\log_2 n + n^2$

Table 4.2.7 Time Complexity for Column Organization.

4.2.5 More about Pivoting

The interchange/pivot operation described above chooses the largest element in the column to replace the diagonal element. This is partial pivoting, and is adequate for

most matrices.

Full pivoting extends the scope of the pivot candidate search to all elements in the submatrix whose upper left corner is the diagonal element in question. This can be implemented on the tree machine by adding the linear selection code from the root processor to the row processor code. The largest element in each row is selected by the row processor. The largest of these is selected by the root. Two interchanges are required. First two columns are interchanged, then two rows. Pivot selection and interchange operations are still $O(n)$ and $O(\log_2 n)$ operations, though the constant factor increases.

The pivot step, either partial or full, involves choosing from among n things. By defining the three level tree of figure 4.2.2, we force the pivot operation to be linear in n. But choosing the largest of n numbers can be done incrementally with pairwise comparisions. Comparing two numbers is a particularily appropriate operation for a binary tree. We define a fourth kind of processor, max, that forms a binary tree between the root and rows. Partial pivoting takes $O(\log_2 n)$ time per column. Full pivoting requires the use of max processors between both the root and the rows and the rows and the elements. Program text for matrix inversion using max processors is given in a memo called "Matrix Inversion on the Tree Machine" [Browning79b].

4.3 Solving Ax = y and AX = B

Suppose the matrix A is non-singular. Then

$$Ax = y \quad ==> \quad A^{-1}Ax = A^{-1}y \quad ==> \quad x = A^{-1}y$$

The preceding paragraphs describe an $O(n^2)$ algorithm for finding A^{-1}. The product $A^{-1}y$ can be found in $O(n)$ time with the tree machine algorithm for matrix multiplication described in the previous section. The solution vector x is found in $O(n^2)$ time.

The same method is used to solve the more general problem AX = B, where A, X, and B are matrices. If A is nxn, and X and B are nxm, then the solution requires $O(n^2+nm)$ time steps. A must be non-singular, since the solution is arrived at by inverting A, then multiplying the inverse by B.

Franklin [Franklin68] suggests that A be augmented with B instead of the identity matrix. The steps that transform A to the identity also transform B into the solution X. This method also has time complexity $O(n^2+mn)$. The cost of each operation is given below.

operation	one column	n columns
Pivot	$2\log_2 n$	$2n\log_2 n$
Interchange	$\log_2 n$	$n\log_2 n$
Eliminate	$2\log_2 mn + m + n$	$2n\log_2 n + mn + n^2$
Divide		$2\log_2 n$
Total	$2\log_2 n + 3n\log_2 n + 2n\log_2 mn + mn + 2n^2$	

Table 4.3.1 Time Complexity for Solving AX=B on the Tree.

4.4 LU Decomposition

Matrix inversion is only one way of solving Ax = y. A more widely used method is direct solution of the linear system. The matrix A is first decomposed into an upper and a lower triangular matrix, U and L, such that A = LU. Given L, U, and y, the solution vector x can be found with a set of subtractions and divisions [Isaacson66, p. 30].

U is found using the same row operations described above, that is, by Gaussian elimination. L is a by product of calculating U. Below the diagonal of L are the factors used to eliminate the lower triangular elements of A. The diagonal elements are ones. To put it more precisely,

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}$$

$$l_{ij} = u_{jj}^{-1}(a_{ij} - \sum_{k=1}^{i-1} u_{kj}l_{ik})$$

LU decomposition is more widely used than matrix inversion to solve linear systems. While both methods require $O(n^3)$ operations, matrix inversion takes three times as many operations as LU decomposition on a sequential machine. Isaacson and Keller do the analysis in detail [Isaacson66, p.34-37].

On the tree machine, however, there is no time saved by doing the elimination only below the diagonal. Elimination of a whole column is done in parallel. Both matrix inversion and LU decomposition are $O(n^2)$.

Pivoting complicates the process of solving for x once L and U are found. The vector y must be permuted the same way that A was rearranged by the pivoting. A permutation matrix, P, must be calculated and applied to y.

$$PA = LU \Longrightarrow LUx = Py$$

If only one solution x is desired, A can be augmented with y, with pivoting affecting the augmented matrix. The need for a permutation matrix is obviated.

4.5 Matrix Problems on the Tree

The preceding paragraphs have presented several matrix problems and their tree machine solutions. While these solutions represent an improvement over the traditional single processor solutions, they are outshown by the results obtainable on special purpose machines like the one described by Kung and Leiserson in Introduction to VLSI Systems [Kung80]. The main advantage of the tree machine is that it can do other things as well as manipulate matrices.

Another advantage of the generality of the tree machine structure is that pivoting does not introduce complications into the algorithm. The machine described by Kung and Leiserson relies on the data flowing through their "systolic array" in a prescribed manner. Pivoting plays havoc with that order by rearranging the data throughout the computation. While some matrices that arise in the course of solving a system of equations can be inverted without requiring pivoting, many cannot be solved without it. The rigid structure and data flow requirements of the systolic matrix machine make multiplication a linear operation, but cannot adapt to the general matrix inversion problem.

Chapter Five

The Processor Architecture

In the first chapter a programming notation for the tree machine was introduced. The notation was used in the previous three chapters to program a variety of algorithms for the tree machine. In this chapter the machine underneath the notation is explored. The size requirements are discussed, an instruction set is defined and used as a target for compilation of the notation, and the loading protocol is described. Finally, the question of floating point is addressed. Should it be built into the hardware of each and every processor, into a select few, or simulated in software?

The important idea behind the processor design discussed here is that the job that any one processor will do is only a small part of the complete algorithm. Thus the processor need not be everything to everybody. There is no need for a seemingly unlimited store or a rich instruction set. Indeed, the more luxurious the processor, the less the tendency of the programmer to fully exploit the concurrency available in the algorithm. Thus, the important tradeoff is between the number of processors required to solve a problem, and the individual capability of the processors. When a feature is added to the processor design, this question must be posed. Is the increased functionality worth the increase in chip area?

5.1 An Overview of the Processor

The tree machine processor has four main parts: a program store, a bank of registers for storing data, an ALU, and some communication handlers. The control and data paths run between these components, aided by three special purpose registers. The I register holds an instruction, the PC register points to the instruction in the program store that will be fetched next, and the AC, or accumulator, is a source and sole destination for the ALU. The address calculation logic is extremely simple in this machine. Only an adder is required, and is provided in addition to the ALU.

Figure 5.1.1 gives a block diagram of the machine. Control lines are indicated as dashed lines, and the data paths are solid lines.

The program store, I-register, PC, and address logic represent one functional unit in the machine. The PC is used to fetch a byte from the program store and into the I-register; following the fetch, the PC is incremented to point to the next byte. The instruction in the I-register is decoded and executed, either affecting operations in the ALU or interupting the sequential flow of the program.

The ALU, AC, and registers comprise a second functional block. The ALU is equipped to perform all the usual arithmetic and logical functions. The AC is the primary operand source for the ALU. The registers will supply a

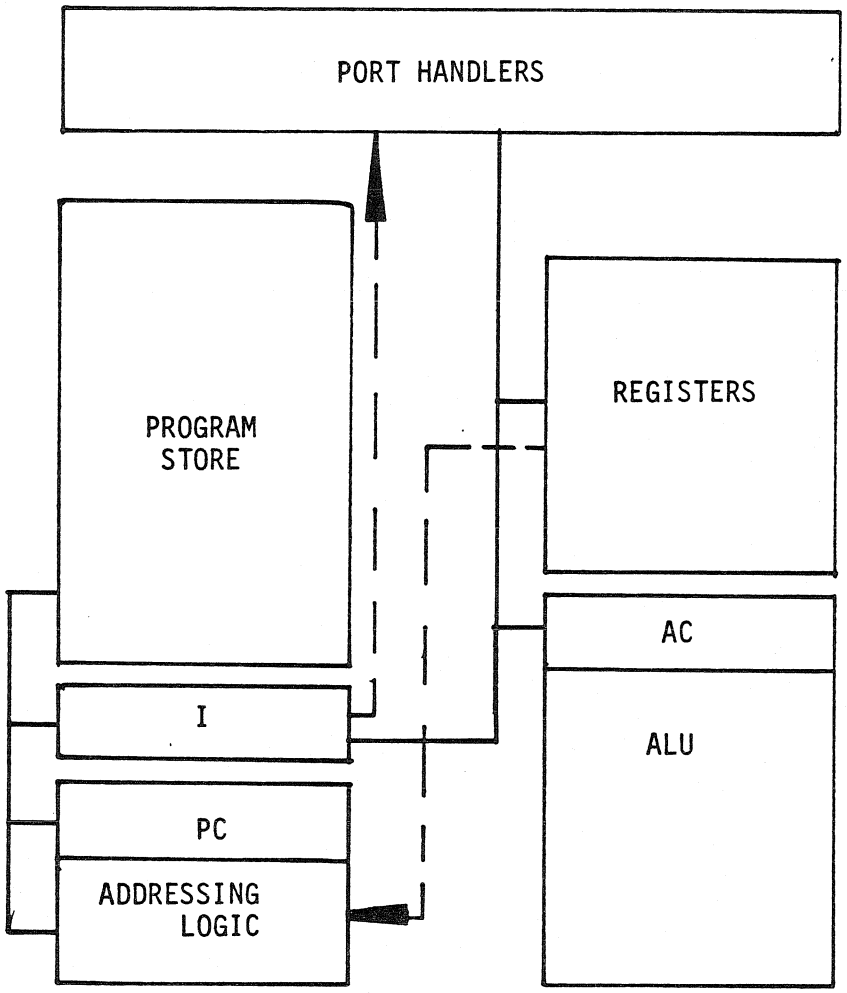


Figure 5.1.1 A Block Diagram of the Processor.

second operand if necessary. The AC is the sole destination of the ALU functions.

The third group of related components are the communication handlers. There are three of them, one for each of the three ports. They provide the interface to the outside world, handling message traffic, loading the program store, and passing code through to their descendents. The actual communication requirements, and thus the details of the structure of the handlers, are discussed in a later section.

The next few paragraphs describe the actual dimensions of the program store, the data registers, and the data path. A discussion of the instruction set that exercises the machine follows. The chapter concludes with a look at the question of execution speed versus chip area: should floating point logic be built into the processor?

5.2 The Size of Each Processor

There are four important numbers involved in the dimension of the processor: the number of bits of program store, the word size of the program store, the number of data registers, and the width of the data path. In order to encourage the distribution of the algorithm through a large number of processors, these numbers should be as small as possible while maintaining the usefulness of the processor. After all, a processor with a 1-bit program store and no registers is not very interesting.

While intuition about what these numbers should be will firm up as programming experience grows, there is a considerable body of programs in the previous chapters. The program and data requirements of these programs have been tabulated, and summarized in figure 5.2.1. The initial recommendation for the processor size is based on these sample programs. I will present the results here; the reasoning behind these decisions is given in the text that follows.

First, the program store dimensions: a four bit nibble is appropriate, with 512 nibbles of memory in each processor, for a total of 2048 bits. The target machine language of the compiler is defined in the next section. Because of the relatively small size of the program store, density is stressed in the choice of instructions to implement. The number of unused bits in the semi-precious program store is minimized.

The initial tree machine processor will have a bank of sixteen 8-bit registers for holding data. Some of these are assigned to user-defined storage. The others can be used by the compiler for temporary storage.

The size of each processor is also related to the kind of arithmetic functionality that is built in hardware. The simplest (and smallest) processor has only integer addition and subtraction, with multiplication and division requiring repetitive shift and add/subtract operations. It is this simple processor that is described here. Other features, like built in multiplication and division and floating point hardware, will be left to future designers and versions of the tree machine.

Algorithm	nibbles of program store	data registers
Vector Sum		
vSumRoot	250	4
vSum	350	7
vSumLeaf	88	6
Heap Sort		
Sort	258	3
sortLeaf	19	1
Transitive Closure		
closureRoot	147	4
node	194	7
endNode	58	2
Clique		
cliqueRoot	138	3
clique	399	7
cliqueLeaf	162	6
Color Cost		
colorRoot	215	4
coloring	412	8
colorLeaf	183	7
Travelling Salesman		
salesmanRoot	285	4
salesman	394	6
salesmanLeaf	198	5
Matrix Multiplication		
root	164	3
row	137	4
element	40	2
Matrix Inversion		
root	289	6
row	426	7
element	162	5

Figure 5.2.1 Space Requirements of the Tree Machine Algorithms.

5.3 The Instruction Set

Programs for the tree machine are written in the high-level language presented and used earlier. The instruction set and machine definition presented here exist solely for the benefit of the compiler. The instruction set does not

provide myriad ways to do something or a rich set of capabilities. It contains only instructions used in code generation.

The novel characteristics of the instruction set arise from the peculiarities of the tree machine processors. For example, the only data store available is the set of registers. There is no need for lavish memory addressing capability in the instruction set since four bits to address a register will suffice.

Because of the concurrency available in the tree machine, we need not concern ourselves with making the individual processors fetch and execute instructions quickly. It is more important that the code be dense, that is, lots of bang for the bit. Each bit of program store increases the size of the processor, and reduces the number of processors per chip.

The instructions are composed of one or more 4-bit nibbles. The first nibble is the opcode, of which there are sixteen. The instructions fall into three categories: control flow, communication, and data flow. If an instruction requires operands, they are provided in the nibbles that follow the opcode.

There are seven control flow instructions. ABORT and SKIP correspond directly to the like-named statements in the notation: ABORT is an abnormal termination statement, and SKIP is a no-op. In addition, HALT is used for normal termination. The other four control flow instructions implement branches. JFL and JFS are conditional jumps (Jump if False) and JAL and JAS are unconditional jumps (Jump

Always). These instructions assume two forms. The long form (JFL and JAL) is followed by an 8-bit offset into a jump table. The jump table contains 12-bit offsets that are added to the PC to arrive at the address of the next instruction to be executed. The jump table begins at location 0 in the program store and may contain up to 86 3-nibble entries. The stream of instructions that make up the program text follows the table. The short form (JFS and JAS) is followed by a 4-bit positive offset that is added directly to the PC. All backward jumps require the long form, but forward jumps can use either one. Since the testing of guards in the conditional and repetitive statements often requires skipping over only a statement or two, the short form is a space saver.

One might wonder why there is no short backward jump provided in the instruction set. Isn't five nibbles a high price to pay for each backward jump? Remember that this instruction set is designed as the target of compilation for the tree machine notation. Forward jumps are needed after the evaluation of each guard in both the conditional and loop statements, and at the end of each statement body in the conditional statement. Backward jumps appear only at the end of statement bodies in loop statements, and these jumps tend to be long jumps anyway. Since there is a limited number of opcodes, it is wise to concentrate on forward jumps.

Figure 5.3.1 is a summary of the control flow instructions. Addressing is to the nibble. The program counter is advanced after each fetch to the next nibble.

mnemonic	# extra nibbles	
ABORT	0	
SKIP	0	
HALT	0	
JFS	1	
JFL	2	(+3 in jump table)
JAS	1	
JAL	2	(+3 in jump table)

Figure 5.3.1 Control Flow Instructions.

There are four communication instructions. There are two for input and two for output. Since message statements can appear either as declarative statements or as guards, there must be two forms of the instruction: one to do the communication if it is appropriate, the other to do it anyway. That is, if the message statement is a guard, the communication is appropriate only if the processor destined to send or receive the message on the other end is ready. The declarative form of the message statement is not so picky: the communication is done even if it means waiting for the other processor to get ready, and control does not return to the processor until the communication is completed. Thus the instructions ? and ! do input and output if it is appropriate (message guards) and ?W and !W will wait for successful completion (message statements). The mechanics of synchronizing two processors by way of a message statement is discussed in detail in the next section. In all cases, if the communication is successful the value true is loaded into the AC. Otherwise, the AC is set to false.

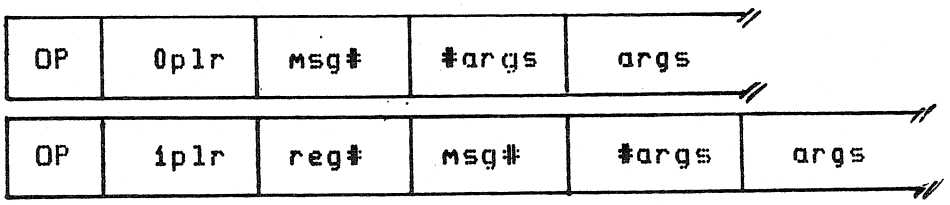
All four of these instructions take a fairly complicated list of operands that include the port, the message name, the number of arguments in the message, and a list of registers that contain those arguments. The message

p?load(c) from the sort processor definition of figure 2.5.2 might be written as

```
?  4,1,1,3  ;input,external port
            ;msg #1,one arg,put in reg #3
```

when it appears as a guard. If the statement is the declarative form, the opcode becomes ?W. The operand list is unchanged.

The nibble representing the port is encoded to allow the specification of a single port, a list of ports, or a register that contains the port name. The high order bit is the indirect bit. If it is zero, the next three bits indicate the ports involved in the communication. The low order bit represents the right internal port, the second bit the left internal port, and the third bit is the external port. One or more of the ports can be named in the communication. On output instructions, the message is to be sent to all the named ports. On input, any one of the named ports may receive the message.



OP: one of !, ?, !W, or ?W
plr: parent, left, and right ports

If the indirect bit of the port specification is a one, the next nibble in the instruction stream names a register that contains a port specification. The register contents are assumed to be encoded in the same manner, though indirection through yet another register is not allowed. The contents of the register are ORed with the original

port specification to arrive at the final specification of the ports involved in the communication.

Figure 5.3.2 is a table of the communication instructions. The limitation on the number of message and arguments is 16. This limitation does not present a serious problem, given the size of the program store. The limit is not even approached by any of the algorithms of the preceding chapters.

I = value of the indirect bit
n = number of arguments

mneumonic	# extra nibbles
?	3 + I + n
?W	3 + I + n
!	3 + I + n
!W	3 + I + n

Figure 5.3.2 Communication Instructions.

The third class of instructions are the data flow instructions. The machine has an accumulator that is the source and destination of all the arithmetic instructions. Three of the instructions are for loading and unloading the accumulator. They are LOAD, LOADI, and STORE. LOAD moves the contents of a register to the accumulator, LOADI moves the 8-bit literal in the next two nibbles to the accumulator, and STORE puts the contents of the accumulator in the specified register.

The 4-bit opcode allows sixteen instructions to be defined. The instructions described above use up 14 of the available codes. The two remaining are used as escape codes. They require another nibble to fully specify the operation. One of them, BOP, is used for binary operators. These use the

accumulator as one source and a register, specified in the third nibble, as the other. The destination is always the accumulator. The binary operators include addition, subtraction, multiplication, division, and, or, exclusive or, and the binary tests: greater than, greater than or equal, less than, less than or equal, equal, and not equal. The last opcode is called UOP and is used as an escape to a set of unary operators: not, two's complement, clear, increment, decrement, set to 1, set to -1, and the tests even, odd, positive, and negative. All of the unary instructions use the accumulator as the sole source and destination of the instruction. The test instructions in the repertoire of both the BOP and UOP instructions leave either true or false in the accumulator, based on whether the condition is satisfied or not. Register sources are unaffected by the binary operators.

Figure 5.3.3 summarizes the last group of instructions, the data flow opcodes. Note that these instructions require no fancy addressing modes. The data is either in the accumulator, requiring no addressing at all, or in a register that requires a 4-bit absolute address. The absence of complex data structures is the result of the limited capacity for data and program storage in each processor.

mneumonic	# extra nibbles	
LOAD	1	(reg #)
LOADI	2	(literal)
STORE	1	(reg #)
BOP	2	(one of +,-,*,/, and,or,xor,),)=,(,(=,=,/= ; reg #)
UOP	1	(one of -,0,1,-1,abs, inc,dec,not,compl pos,neg,odd,even)

Figure 5.3.3 Data Flow Instructions.

The instruction set defined here is intended as the compilation target of the high level notation described in Chapter 1. The code generation strategy is the subject of the next section.

5.4 Code Generation

The previous section defined an instruction set for the tree machine processor. In this section we will see how the high level notation described in the first chapter is translated into the instuction set. I have written a program that does this translation. This is only one part of the complete compilation task, and is the simplest part. A design for the complete compiler is proposed in the next chapter. There are some difficult problems involved in immplementing the other parts of the compiler. I will point them out in the next chapter as topics that need to be investigated.

The first few paragraphs explore the general problem of converting each of the different language constructs into

machine code. Following that the marble sorter example will be compiled into the machine code of the tree processor.

5.4.1 The Assignment Statement

The syntax of the assignment statement is given below, in figure 5.4.1. Note that a list of variables, and a corresponding list of expressions whose values will be assigned to the variables, can be specified in a single assignment statement. The semantics of the statement say that all the expressions are evaluated before any assignments are made. That is, the order in which the expressions are evaluated is not allowed to matter. Thus, the statement

```
num,c := c,num
```

for the heap sort algorithm requires the use of a temporary register to hold a copy of num while the value of c overwrites the original. The semantics of the statement are, and in fact must be, identical with those of the trio of statements below.

```
temp:=num ; num:=c ; c:=temp
```

In situations where there are dependencies among the expressions in the list, the compiler assigns free registers to hold intermediate results. In the more common situation where all expressions can be evaluated and the assignments made without fear of modifying values used later, the task is simple. The compiler will evaluate the

expression and store it in the selected variable.

```
⟨assignment⟩ ::= ⟨identifier list⟩ ':=' ⟨expr list⟩
⟨identifier list⟩ ::= ⟨identifier⟩
                    | ⟨identifier⟩ ',' ⟨identifier list⟩
⟨expr list⟩ ::= ⟨expr⟩ | ⟨expr⟩ ',' ⟨expr list⟩
```

Figure 5.4.1 The Assignment Statement Syntax.

Figure 5.4.2. shows the machine instructions that are generated when an assignment is compiled. There are two cases: 1) the expressions are independent, and 2) the expressions overlap. The example shows an assignment with identifier and expression lists containing two items. The technique extends in an analogous way to longer lists.

a,b := ⟨expr$_1$⟩ , ⟨expr$_2$⟩	⟨expr$_1$⟩
(⟨expr$_1$⟩ does not involve b and	STORE r_a
⟨expr$_2$⟩ does not involve a)	⟨expr$_2$⟩
	STORE r_b
a,b := ⟨expr$_1$⟩ , ⟨expr$_2$⟩	⟨expr$_1$⟩
(⟨expr$_1$⟩ is a function of b and	STORE r_t
⟨expr$_2$⟩ is a function of a)	⟨expr$_2$⟩
	STORE r_b
	LOAD r_t
	STORE r_a

Figure 5.4.2 Code Generation for the Assignment Statement.

If the compiler runs out of registers to use as temporary holding places for dependent expressions, it will issue a diagnostic message telling the user to simplify the assignment statement.

5.4.2 The Message Statement

The message statement translates directly into one of the four communication instructions. Each of the messages is assigned a number the first time it is encountered. Message arguments can be either named variables of expressions. Variables have already been assigned a register, and that register appears in the argument list of the message instruction. Expressions must be evaluated, the result stored in a temporary register, and that register inserted into the argument list.

A communication opcode is chosen to fit the situation: ! and ? for imperative send and receive, !W and ?W for guarded send and receive. The instruction is completed with an indication of the port, the number assigned to the message, the number of arguments, and a list of registers that contain the arguments. Figure 5.4.3 shows the technique for a message with two arguments, one a variable, the other a literal.

```
p?msg(a,50)                    LOADI 50
                               STORE r_t
                               ?W    port#,msg#,2,r_a,r_t

p?msg(a,50) --> <body>         LOADI 50
                               STORE r_t
                               ?     port#,msg#,2,r_a,r_t
                               JFx
                               <body>
```

Figure 5.4.3 Code Generation for the Message Statement.

5.4.3 The Repetitive and Conditional Statements

The syntax of the conditional and repetitive statements is given in figure 5.4.4. They are alike except for two things: what happens when none of the guards can be satisfied, and what happens after the statements following a guard are executed. The conditional statement aborts the process if the evaluation of all its guards generate false. The repetitive statement terminates the loop. After the statements following a guard are executed in a conditional statement, it terminates. A repetitive statement goes back to the top and loops.

```
⟨iteration⟩ ::= '{' ⟨guard list⟩ '}'
⟨conditional⟩ ::= '[' ⟨guard list⟩ ']'
⟨guard list⟩ ::= ⟨guarded body⟩
                 | ⟨guarded body⟩ '|' ⟨guard list⟩
⟨guarded body⟩ ::= ⟨guard⟩ '-->' ⟨body⟩
⟨guard⟩ ::= ⟨boolean expr⟩ | ⟨message⟩
```

Figure 5.4.4 Conditional and Repetitive Statement Syntax.

Code generation is easy for both of them. The guards are tested in order. When one is selected, the accompanying body of statements is executed and a jump is performed either to the top of the loop or around the other cases. The guards are chained together with JFx (Jump False long or short) instructions; the chain ends either with an ABORT instruction (conditional statement) or falls through to execute the statement following the loop (repetitive statement). Figure 5.4.5. shows the code generated by the compiler for both the repetitive and conditional statements.

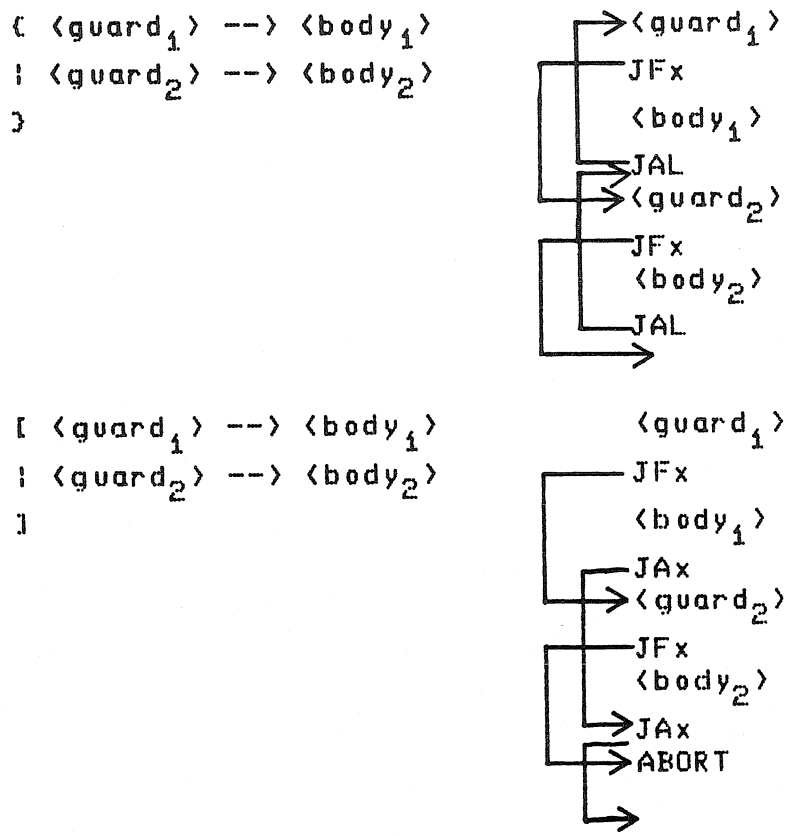


Figure 5.4.5 Conditional and Repetitive Statement Code Generation.

5.4.4 skip and abort

The code generation for the remaining statements, skip and abort, is trivial because there is a direct equivalent of each in the machine instruction set. Thus, skip is SKIP and abort is ABORT.

5.4.5 An Example of Code Generation: marbleSorter

In this section the binary tree version of the marbleSorter tree is compiled. Figure 5.4.6 is the compiled newColorSorter and figure 5.4.7 shows the code generated for the padding and bin processors. In each case the register and message assignments are shown.

```
registers: color=0
messages: marble with one arg=0, marble with no args=1

Jump Table
 0: 18
 3: 57
 6: -65

 9:      ?   4,0,1,0          ;{ In?marble(color) -->
14:      JFL 3
17:          UOP SET1         ;    [ color=1 -->
19:          BOP =,0
22:          JFS 12
24:              UOP SET1     ;        L!marble(1)
26:              STORE 1
28:              !W 2,0,1,1
33:              JAL 0
36:          LOADI 2          ;    | color=2 -->
39:          BOP =,0
42:          JFS 12
44:              LOADI 2      ;        L!marble(2)
47:              STORE 2
49:              !W  2,0,1,2
54:              JAS 15
56:          LOADI 2          ;    | color>2 -->
59:          BOP >,0
62:          JFS 6
64:              !W  1,1,0    ;        reject!marble
68:              JAS 1
70:          ABORT            ;    ]
71:      JAL 6                ;}
74:      HALT                 ;.
```

Figure 5.4.6 The newColorSorter Processor.

```
registers: r=0
messages: marble=1

Jump Table
 0: -14

 3:      ?   4,0,1,0          ;( p?marble(r) -->
 8:      JFS 7
10:          !W  (0),0,0      ;   c(r)!marble
15:          JAL 0            ;)
18:      HALT                 ;.
```

(a) the padding processor

```
registers: cnt=0
messages: marble=1

Jump Table
 0: -16

 3:      UOP CLEAR            ; cnt:=0 ;
 5:      STORE 0
 7:      ?   4,0,0            ;( p?marble -->
11:      JFS     9
13:          UOP SET1         ;   cnt:=cnt+1
15:          BOP +,0
18:          STORE 0
20:          JAL 0            ;)
23:      HALT                 ;.
```

(b) the bin processor

Figure 5.4.7 The padding and bin Processors.

5.5 Loading the Tree Machine

The question of loading code into the program store has been treated rather lightly in the preceding discussion. It is time to rectify the situation by looking at it in detail.

The special capabilities for loading code are part of the external port handler. Since code is loaded only from the bus connected to the external port of the root processor of the tree, it travels down to the leaves, never up.

The code streams are preceded by a header that directs the code to a particular processor, or to a class of processors. This header is prepared by the compiler.

The header contains five fields, as shown in figure 5.5.1. There is a two-bit opcode, a number describing the length of the address, in bits, that follows, the address of the destination processor, the initial value of the PC, and the length, in nibbles, of the code stream. The initial PC value is included so that the jump table, which begins at location 0, is skipped over.

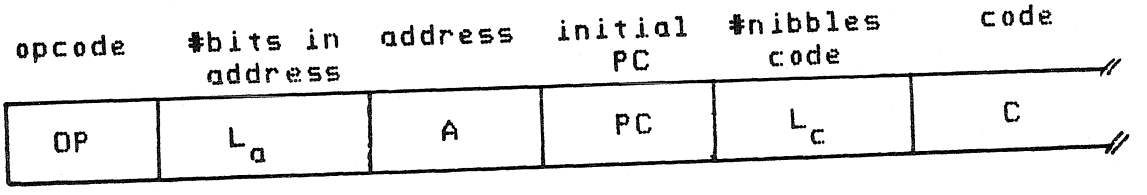


Figure 5.5.1 Program Header.

The two-bit opcode can take on four different values: ONE, TREE, LEVEL, and YOU. ONE means that the address points to the specific processor that will receive the code. TREE is used to load an entire subtree, rooted with the processor addressed in the A field of the header, with code stream. Similarily, the LEVEL function directs the stream to all processors on the level specified in the address. (The root is level 1.) The fourth operation, called YOU, is used by a parent processor to force the loading of a

particular descendent. The address is ignored by the receiving processor. This operation is used in testing the tree machine processors. While the effect of a ONE operation with a zero length address is, the same, the YOU instruction requires that less circuitry/microcode be working.

The processors are uniquely identified by a bit string that grows with the depth of the tree. The root is called 1, its descendents are 10 (the left child) and 11 (the right child), and so on. Figure 5.5.2 shows the generating pattern.

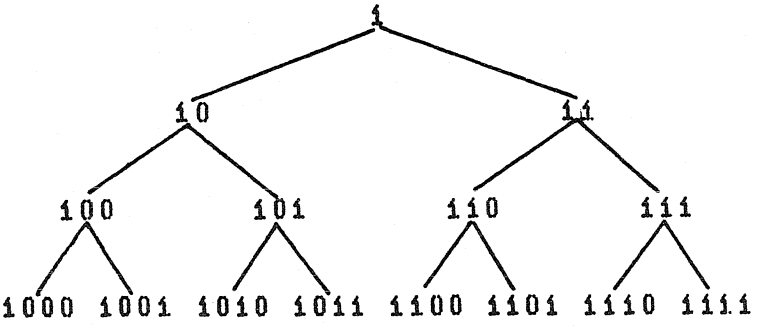


Figure 5.5.2 Assigning Unique Addresses to Processors.

Notice that the processor name, and thus the number of bits it takes to store it, grows linearly with the depth of the tree. This linear growth requirement runs counter to the programming philosophy of Chapter 3. But look again at the way the addresses are generated. The address of a processor differs from that of its parent by only one bit, the extra one on the end. There is no need to store the duplicate bits: state information is hidden in the paths of the tree just as it was in the algorithms that solve the

NP-complete problems on the tree machine. Thus the address becomes a single bit of routing information: 0 means the left, and 1 the right port. A more accurate picture of the addressing is given in figure 5.5.3.

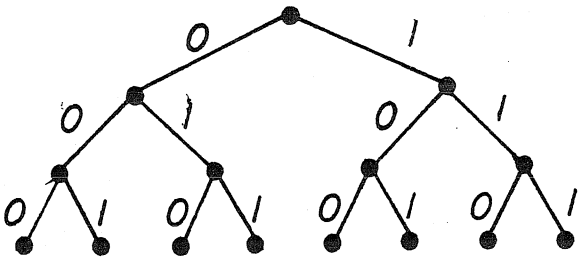


Figure 5.5.3 Path-Oriented Address Assignment.

As each processor examines the header of a code stream, it looks first at the opcode. If YOU is specified, it skips over the address and loads the number of bits specified in L_c into the program store. If LEVEL is the opcode, the processor looks at L_a. If L_a is nonzero, it is decremented, the leading bit is stripped off A, and the modified header and code are passed down the tree via both the right and left ports. If L_a is zero, the code is loaded into the program store.

If the opcode is either ONE or TREE, the address of a processor or group of processors is included in the instruction. If the address has a nonzero length ($L_a>0$), the code is intended for a processor deeper in the tree. The leading bit of the address is stripped off and used as routing information. If it is zero, the modified header and code go to the left, else to the right. If the length of the address is zero, the code stream is loaded into the program store. If the opcode was ONE, that is all that

happens. If, however, TREE was specified, the entire subtree beneath the processor must be loaded with the same program. The header, with TREE and L_a=0, and code are sent to both the right and left. The descendent processors, following the algorithm given above, will notice the zero length address, load the code, notice the opcode of TREE, and pass it to their descendents.

Figure 5.5.4 summarizes the opcodes and the actions they invoke in the external port handler. The internal port handlers merely pass code stream messages through to the external port of the descendent processor. The action invoked by each opcode is written symbolically in the tree machine notation. This only to provide a description of what happens. In fact, these operations are part of the hardware or microcode of the external port handler.

opcode	function
YOU	load.
LEVEL	[L_a=0 --> load ! L_a>0 --> L_a:=L_a-1 ; A:=leftShift(A) ; 1,r!code(LEVEL,L_a,A,L_c,C)] .
ONE	[L_a=0 --> load ! L_a>0 --> L_a:=L_a-1 ; R:=msb(A) ; A:=leftShift(A) ; [R=0 --> 1!code(ONE,L_a,A,L_c,C) ! R/=0 --> r!code(ONE,L_a,A,L_c,C)]] .
TREE	[L_a=0 --> load ; 1,r!code(TREE,0,,1_c,C) ! L_a>0 --> L_a:=L_a-1 ; R:=msb(A) ; A:=leftShift(A) ; [R=0 --> 1!code(TREE,L_a,A,L_c,C) ! R/=0 --> r!code(TREE,L_a,A,L_c,C)]] .

Figure 5.5.4 Code Stream Opcodes.

Let us look at the effect of the different opcodes given a particular load stream. Figure 5.5.5 shows four load streams differing only in opcode. The address and code fields are the same in each case. The tree machines show where the code is loaded, given a particular opcode.

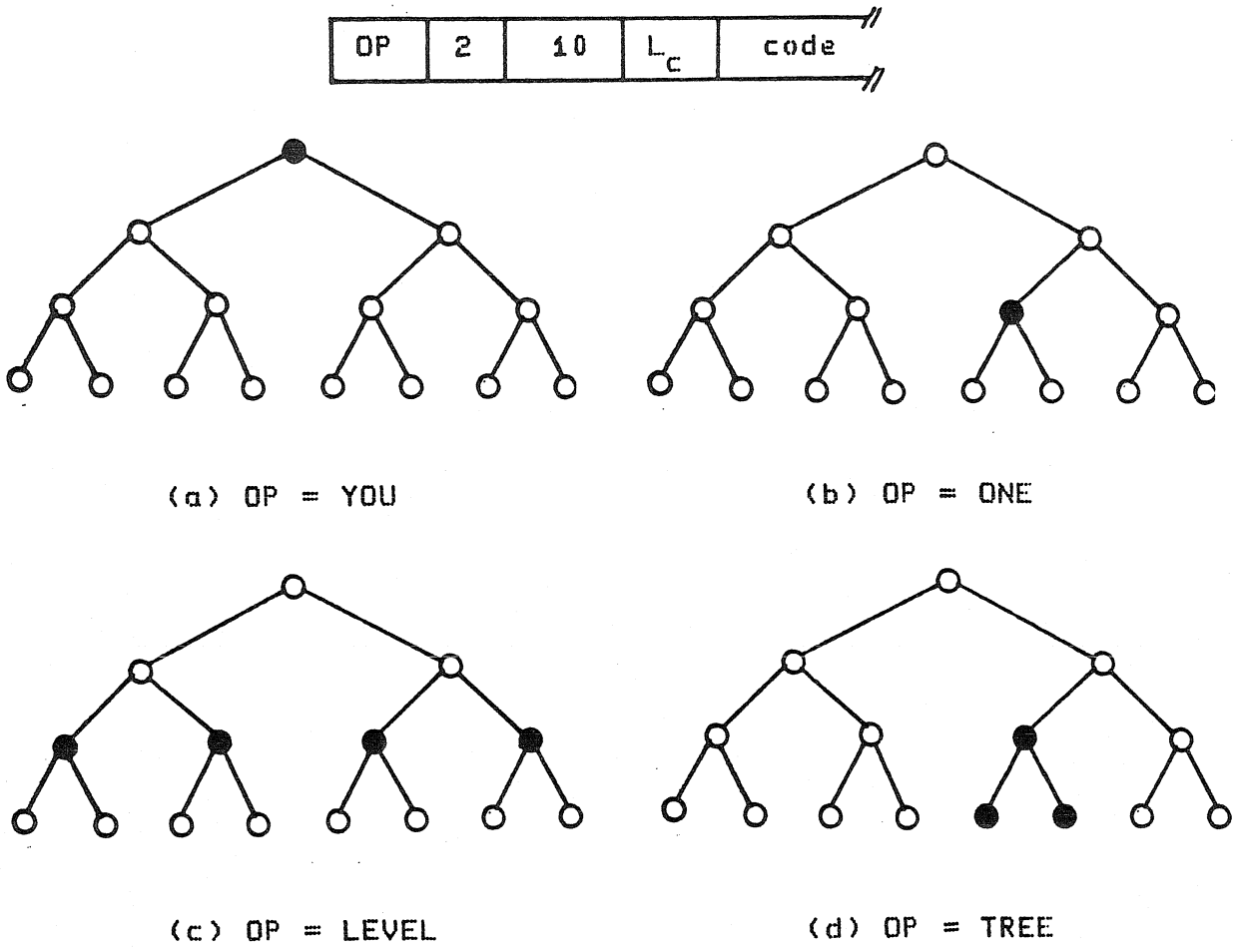


Figure 5.5.5 Loading a Tree Machine.

5.6 Floating Point

A collection of matrix manipulation algorithms was presented in the fourth chapter. They all declared real variables and performed floating point arithmetic on them. In the machine description presented earlier in this chapter only integer arithmetic was provided. In this

section we will describe the problems peculiar to floating point arithmetic, discuss the tradeoffs between building floating point hardware and simulating the arithmetic in software using integer arithmetic, and close with a look at numbers that are wider than the 8-bit data path of the processor.

5.6.1 A Brief Look at Floating Point Arithmetic

A floating point number is represented as a sign, a fractional part and an exponent. The exponent is applied to a base of 2. That is, the number ten might masquerade as the floating point number $+.625 \times 2^4$. Internally, this might be stored as shown below.

sign	exp	fraction
0	0100	101

Arithmetic operations are applied to the exponent and fractional part, or mantissa, separately. The two are treated together in the normalization step at the end of each arithmetic operation. Normalization is the process of adjusting the exponent until all the significant bits are to the right of the "binary point" and the leftmost bit of the mantissa is nonzero.

The familiar rules for performing arithmetic on numbers in scientific notation apply. That is, two floating point numbers can be added or subtracted only if their exponents are the same. The typical practice is to make the one with the smallest exponent match the larger one by shifting the mantissa right and incrementing the exponent until agreement is reached. This way, only the least significant

bits are lost. Multiplication and division can be performed on numbers with disparate exponents: the exponents are added in multiplication and subtracted in division. The mantissas are multiplied or divided as appropriate. Before any of the four arithmetic operations are completed the answer must be normalized.

There are a collection of exceptional, i.e. bad, conditions that can arise during the arithmetic. The mantissa can overflow (no big deal). The exponent can overflow or underflow (a disaster). If the mantissa overflows, the least significant bit is shifted out, the exponent is incremented by one, and the processing continues. Exponent overflow (too big) or underflow (too small) is a different matter. The number simply cannot be represented in the number of bits provided. Flores [Flores63] gives a lucid description of when and why these exceptional conditions arise.

With this introduction to floating point, albeit brief, let us get on with the business at hand. How shall a tree machine processor manage floating point numbers?

5.6.2 Hardware vs. Software

If the floating point logic is built into the hardware, it makes the processors bigger, thereby decreasing the number of them on a chip. If the floating point is simulated in software, less of the semi-precious program store is available to the user. Or, alternatively, the program store must be increased, again increasing the chip area occupied by each processor. Another alternative is to use

several processors to do the task.

There are a couple of ways to reduce the cost of providing floating point support. The arithmetic operations can be made slow but simple, for example, bit serial. This is currently under consideration by Peggy Li of Caltech as a Master's project. Alternatively, a select few of the processors can be endowed with floating point hardware. It is possible to write the matrix algorithms of Chapter 3 so that the floating point arithmetic is confined to the leaf processors. This increases the number of messages and the length of those messages, but reduces the amount of chip area devoted to floating point functions by one half. That is, only half of the processors, the leaves, have floating point hardware. The matrix inversion algorithm described in my memo "Matrix Inversion on the Tree Machine" [Browning79b] is an example of an algorithm with floating point required only in the leaves.

The ideal solution is to supply floating point capability with a combination of both software and hardware. The leaves can be built with the hardware capability. Messages asking to borrow the floating point unit for a cycle or two travel the right direction in the tree: toward the greatest bandwidth. If processors higher in the tree have sufficient program store to absorb the software package that simulates floating point, it can be loaded with the rest of their program, thereby reducing the trafficking in messages.

5.6.3 When Eight is Not Enough . . .

The data path of each tree machine processor is eight bits wide. Floating point representation that uses a total of eight bits for each number is not very interesting: given the representation shown above using one bit for the sign, four bits for a signed exponent, and the remaining three bits for the mantissa means that numbers between 2^{7} and 2^{-8} can be represented with approximately one digit of precision. In general, floating point numbers should be represented by more than eight bits.

Widening the data path to 16, 32, or 64 bits not only increases the area of the processor, but forces a lot of waste in operations that can proceed quite happily with smaller word sizes. There are a couple of ways the wider data path can be simulated that require no modification to the hardware described earlier in this chapter.

First, if there are enough free registers in a given processor, the compiler can allocate several of them to a real variable. This technique is possible with the composite processors discussed in the first chapter. These composites exist whenever the fanout from a node is greater than the two-way branching provided by the physical structure of the tree. Since most of the processors in the composite simply pass messages on, they have lots of unused registers.

A related technique is to distribute the number in several processors. Each processor performs arithmetic on its piece of the number, sending and receiving carry information from its neighbors. This scheme is an

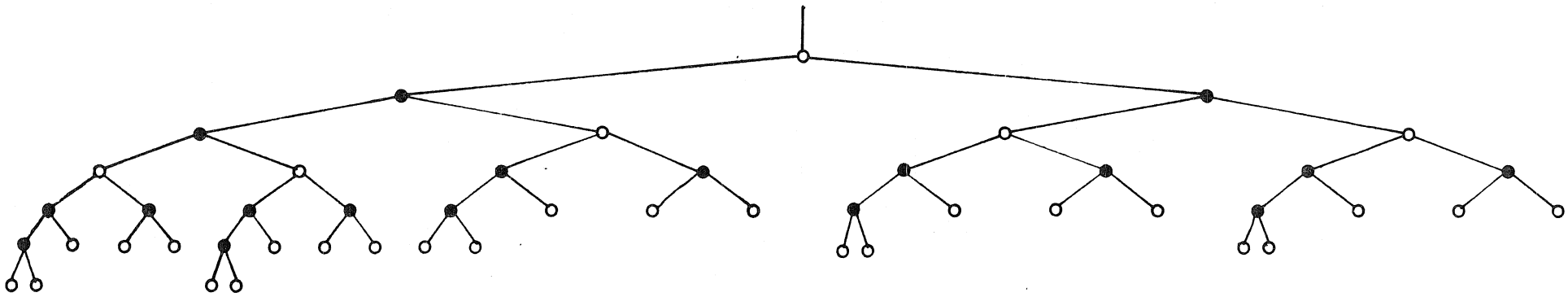
interesting way to do long addition, subtraction, and multiplication because the carry chain penalty can be postponed and done for 8-bits at a time. Long division, however, is blazingly slow. Nonetheless, the technique deserves a closer look.

The matrix multiplication and inversion algorithms have fanouts from the root and row processors of n. Thus, there are n-1 processors in each composite processor. These padding processors can be used to hold the bits and pieces of the real variables.

Figure 5.6.1 shows the tree machine for multiplying or inverting a 5x5 matrix. The root, row, and element processors are shown as solid circles, the fill-ins as open circles. There is plenty of room to provide real numbers with eight bit sign and exponent, stored in the row and element processors, and 24-bit mantissas, stored eight bits at a time in the padding processors. An extension of the tree beneath the element processors is required.

How is the floating point arithmetic simulated in software? We look closely at each of the four arithmetic operations in the next few paragraphs.

First we will look at addition and subtraction. There are three basic steps: 1) make the exponents match, 2) do the addition or subtraction of the mantissas, and 3) normalize the result. The processor that has the exponents directs the first step. The smaller exponent is made to agree with the larger one, keeping track of the number of times the mantissa should be shifted right. The high order bits of the mantissa are shifted first, with the spillover handed



○ = root, row, or element processors

● = padding or max processors

Figure 5.5.1 Matrix Multiplication/Inversion Tree for n=5.

to the processor responsible for the middle portion of the fractional part. This shifting continues until the processor with the low order bits of the mantissa is done. The spillover from the last shift operation is lost. Keep in mind that the shifting is done to a copy of the original number: none of its precision is lost.

Once this setup is completed, the addition or subtraction can proceed. In order to minimize the cost of the carry chain, all of the processors except the ones with the exponent and low order bits of the mantissa do the addition twice: first assuming a carry in, and then with no carry. The low order bits are added, generating the first carry information. That carry is used to select one of the two sums in the next processor, and the carry from that sum is passed to the processor representing the next highest order set of bits, and so on.

The final step is to normalize the result. The exponent is incremented or decremented appropriately and the shift ripples either from the high order bits to the low order bits, or vise versa, until the leftmost bit of the mantissa is nonzero. A special check for a zero result must be made, since its rather difficult to normalize zero.

How about multiplication? Figure 5.6.2 shows the way a product is composed from partial products. The two partitioned numbers are multiplied in pieces and the pieces combined with addition to give the final product. The various multiplications are overlapped so that the only synchronized action is the carry chain of the addition in the final assembly. Again, we must normalize the result, but there is no setup step analogous to reaching exponent

agreement in addition. The exponents are simply added together to yield the exponent of the (unnormalized) product.

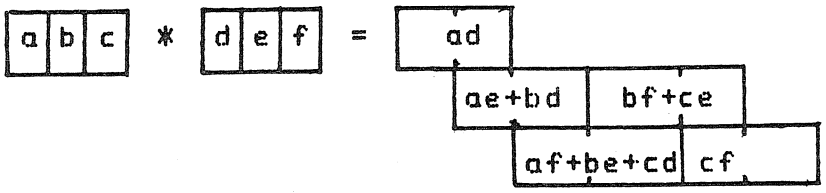


Figure 5.6.2 Multiplication: The Sum of Partial Products.

Division, on the other hand, cannot be done in a distributed way, though some overlap is possible. Each piece of the dividend can be divided by the whole divisor independently, as shown in figure 5.6.3. But the whole divisor must be assembled in each processor, and the division of the high order bits must be done to the full precision, shown by the overlapped parts below. Again, the final step of the computation is normalization.

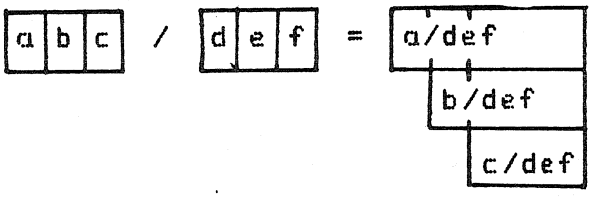


Figure 5.6.3 Division: Blazingly Slow.

Let's take a look at the cost of each of the floating point operations, assuming worst case. Addition and subtraction require exponent massage, add with carry, and normalization. These are all O(m) operations where m is the number of bytes used to represent the mantissa. Multiplication requires m^2 multiplications and 2m-3 sums. The multiplications are overlapped so that m of them are

done at a time. The partial products can often be formed in processors that are appropriate for the additions. Since normalization remains O(m), multiplication is also O(m). Division requires the transmittal of m pieces of the divisor to all the mantissa processors, m(m+1)/2 divisions done m at a time (roughly), followed by normalization. Thus, it would appear that division is O(m) as well.

These cost functions are unrealistic, however, because they assume that integer addition, multiplication, and division of 8-bit quantities all have unit cost. This is most assuredly false in the simple tree machine processor described in this chapter. Multiplication requires repetitive shift and add operations and takes roughly eighty times as long as simple addition. (There are about ten instructions in the loop, and it is executed 8 times.) Similar costs apply to division.

The bottom line is this. Floating point can be simulated in software. So can wider word widths. If the tree machine user wants fast floating point arithmetic on numbers with lots of precision, however, it should be built into the hardware.

Chapter Six

"... As lovely as a tree ..."

In their recently published book, Introduction to VLSI Systems, Carver Mead and Lynn Conway suggest that we let examples of complex systems in nature aid in the design of complex computer systems [Mead79,p.264].

> "Human organizations, like computer organizations, suffer if communication costs are high or if concurrent processing cannot be exploited. In fact, a human brings to an organization what VLSI brings to a circuit: both combine processing and memory effortlessly! Analogies with human structure may help to suggest the kinds of behavior we might achieve in computational structures."

The most durable of human organizations is the hierarchical management structure of large corporations, and it is this structure that we mimic in the tree machine. Each processor has a restricted set of communication paths available to it, communication is strictly local, and there are no cross links to dilute the hierarchy.

Chapter One presents a body of physical evidence to support the notion that a tree machine is a practical and interesting architecture to expoit the properties of VLSI. In this chapter, I will argue that the tree machine is a good model for understanding and organizing concurrent programs as well.

The chapter has three sections. The first two sections, titled Concurrency and Communication, reiterate the theme of the previous six chapters. If we are to take advantage

of VLSI, we must build a new theoretical foundation for designing and analyzing algorithms. The tree machine model of computation, its notation, and the emerging body of algorithms provide a beginning. The final section discusses possible extensions to the results presented here and new directions the research might take.

6.1 Concurrency

In his presentation at the VLSI Conference at Caltech in January of 1979, Martin Rem [Rem79a,p.57] made the following statements:

> "The design of ... an ultraconcurrent computation is not an easy task. In this respect VLSI came too early: we are beginning to understand the theory of sequential programming, but still have only a rudimentary knowledge of concurrent programming."

It is not easy to map a sequential algorithm onto a tree machine. It is not even clear how worthwhile that activity would be. The programmer must go back to the original problem, throw away any preconceived notions about the form of the program to solve the problem, and examine the problem with a fresh outlook. As a result of mapping several problems onto the tree machine, I have developed some rudimentary guidelines for designing tree machine algorithms.

There are a couple of intellectual hurdles that must be crossed in the early stages of programming a problem for the tree machine. The program must first be broken into modules that will run in the tree machine processors. This division is usually as fine as the smallest unit of data:

the individual numbers to be sorted, the individual elements in a matrix, or the individual nodes of a graph. Then the control flow of the problem must be partitioned so that each module is as independent of the others as it can be. Any natural structuring of the data should also be reflected in the tree: the rows of the matrix, for example, or the paths in a graph.

The most helpful metaphor is the object-oriented nature of the Simula class, described briefly in section 1.2. Each class, or module, or processor, is viewed as an independent, autonomous unit. It is not manipulated by external processes. It is asked to manipulate itself. Thus, locality is emphasized. All operations are local ones, and all requests for action can be scrutinized before effecting a change.

Like programming the tree machine, programming in Simula's object-oriented world does not come naturally. There is a significant difference between the recursive subdivision of problems encouraged by the Algol-Pascal family of languages that most programmers learn first, and the modularity of Simula and its derivative languages. Nonetheless, object-oriented programming is the best possible experience to have before trying to design an algorithm for a tree machine.

The second major step in designing a tree machine program is the distribution of control through the entire tree of processors. For example, in the matrix inversion algorithm of Chapter 4, the row processors expect the pivot, interchange, and eliminate sequence to happen n times. After that, they initiate the divide step themselves: the

root is not involved in the computation, so it is not involved in the control either. In the NP-complete problems of Chapter 3 the leaves initiate the answer reporting phase because that is where the answer is. If knowledge of the control flow of a program is built into all of the processors in the tree, fewer synchronizing messages need to be exchanged, thus reducing the execution time.

The key to finding and exploiting the concurrency available in an algorithm depends on two things: proper choice of modules, and proper distribution of control flow. The subdivision into modules should be guided by the natural organization of the data. The control flow should be distributed to minimize the number of sequencing messages.

E.W. Dijkstra in Structured Programming [Dahl72,p.6] said that "the art of programming is the art of organising [sic] complexity". That statement is particularily apropos to the tree machine.

6.2 Communication

In their 1977 article "Microelectronics and Computer Science", Ivan Sutherland and Carver Mead [Sutherland77,p.212] suggest that foundations of Computer Science will be reconsidered as VSLI becomes a reality.

> "... we feel justified in describing as revolutionary the effect of integrated-circuit technology both on the design of computing machines and on the intellectual framework within which such machines are exploited."

They point out that because wires and communication, not logic and memory, are expensive in VLSI, the criteria we use to measure the cost of algorithms must change. The second chapter of this dissertation presented a model of computation in which communication is the dominant cost. In fact, in all of the algorithms presented here, the problem is decomposed into modules in such a way that the computational cost within a module is insignificant. I believe that computational models and cost functions like the one presented here will replace the time/space costs and models of sequential machines.

One of the effects of replacing the theoretical foundations of the theory of algorithms is that notations will change as well. If communication is to provide the basis of the cost analysis, the notations must accommodate explicit communication statements. After all, the algorithm designer must be able to count something to arrive at a cost for the algorithm. Chapter One presented a notation that contains communication primitives and the notion of a multiprocessor system. While the notation is biased toward tree machines with a strict hierarchy, the chapter concludes with some comments about how the notation can be extended to other structures. It is notations like the one given here that will be the programming languages of the next generation of computing machines.

Given that we want to exploit concurrency and count communication costs, why should we do it on a tree machine? One of the most persuasive reasons was given by Martin Rem [Rem79a,p.62]:

> "A hierarchy is the only way to build complex systems with a high confidence level. They enjoy the nice property that we can prove assertions about the system by recursion over the hierarchy: assuming that the assertion holds for the subprocess we can prove that it holds for the process itself."

Entire subtrees are characterized by the communication through the external port, without regard to the substructure beneath the root of the subtree. In making and proving assertions about the behavior of a processor in the tree, you never have to look further than one level into the substructure. Thus, as Rem concludes, "What is mathematically attractive turns out to be physically attractive as well." [Rem79a,p.62].

6.3 Future Directions

The development of a theoretical foundation for the study of algorithms that is appropriate to VLSI is a very fertile area for research. I have presented a model of computation for hierarchical machines. A more general model, not tied to a particular interconnect structure, should be developed. In addition, more understanding of how to design an algorithm for an ultraconcurrent machine is needed. With this understanding comes the development of proof techniques for concurrent programs. Work has been started in all of these areas. At Caltech, Young-il Choo and James T. Kajiya are looking at proof techniques, including notations that encourage correctness by construction. Many of the investigations by Caltech's Silicon Structures Project into how to manage the complexity of implementing a design in VLSI will influence the study of programming complexity. At Carnegie-Mellon University, a group of people led by H.T. Kung are looking

at implementing algorithms directly in silicon. One of them, Clark Thompson, has proposed a cost function for algorithms based on chip area [Thompson79]. Since wires represent the bulk of the area on a VLSI chip, one might argue that he, too, is counting communication cost. Since his analytic model, an extension of work begun by Carver Mead and Martin Rem [Mead79,p. 313-329], is independent of a particular interconnection pattern, perhaps it will become the de facto standard.

In addition to forays into the theory of algorithms, there is more work to be done on the tree machine itself. The obvious next step is to build one. Peggy Li will probably do just that. The implementation of interprocessor communication will be the most critical part of building a tree machine. The individual processors are asynchronous modules. All synchronizing between processors is accomplished via communication statements. The communication takes place if one of the two processors is willing to receive the message that the other is trying to send.

Communications can be specified either as statements or as guards. As long as at least one of the processors involved is communicating via a message statement, synchronizing the two processors is easy. One of the processors will suspend further processing until the communication is completed. But what if both of the processors are communicating from guards? The consequences of message guards need to be fully explored. They are clearly useful, possibly even necessary. But they greatly complicate the implementation of a communication protocol. Perhaps the work in self-timed logic being done by Charles L. Seitz and Charles

Molnar, among others, will provide us with hints about how to implement the communication structure safely.

Another implementation issue is the problem of unbalanced trees. If a pure binary tree is implemented, the user will define an unbalanced program tree whenever the logical fanout is not a multiple of the physical fanout. James T. Kajiya is looking at a structure that has more than three physical ports on each processor. That is, as well as parent and children links, the processor might also have access to some of its sibling processors, and to selected children of siblings. This structure is still planar, but has the advantage that the extra links allow the program to remain balanced regardless of program fanout. Because tree machine programs define static structures, a tree machine compiler could lay out a binary tree on Kajiya's peculiar structure. Thus, the programmer would still be dealing with a pure tree machine, and the compiler would be optimizing the program's processor usage.

Speaking of compilers, there is much more work to be done on the compiler for the tree machine notation. I have written a program that translates from the notation to the instruction set I defined in Chapter 5. I envision this as the second phase in the compilation process. It should be preceded by a mapping phase that takes processor definitions with arbitrary fanouts and generates new processor definitions with only two internal ports. The code generation is followed by the tree assembly phase in which a load stream for the tree machine is constructed using the protocol described in Chapter 5. Figure 7.3.1 shows the three phases of compilation that tree machine programs might undergo.

The most difficult of these compilation steps is the first one. A set of customized processor definitions must be manufactured by the compiler to pad out the tree. The original composite processor is modified to communicate with the padding processors. Messages that pass through the padding are altered to include their final destination in addition to the other arguments. It is hard to modify the original processor definition to communicate via only two ports while preserving the original semantics. It is even harder to produce padding processors that don't add deadlock situations to the program. The next few paragraphs give an infantile strategy for the mapping. It is by no means a solution to the problem, but provides a starting point for future work.

My perception of the mapping problem is that it proceeds recursively. The composite processor is used to generate three processor definitions: a modified version of the processor that has only two internal ports, and two padding processors. Each padding processor has half as many internal ports as the original processor. These padding processors are themselves composite processors, and must undergo the mapping process.

There are two tasks that are involved in mapping a composite processor onto a binary tree. The first one involves changing the original processor definition to communicate along only two internal ports. The second step is to generate the padding processor definitions and change the connection plan accordingly. I will look at each part of the problem, pointing out potential pitfalls. Martin Rem and his students are working on a general solution to

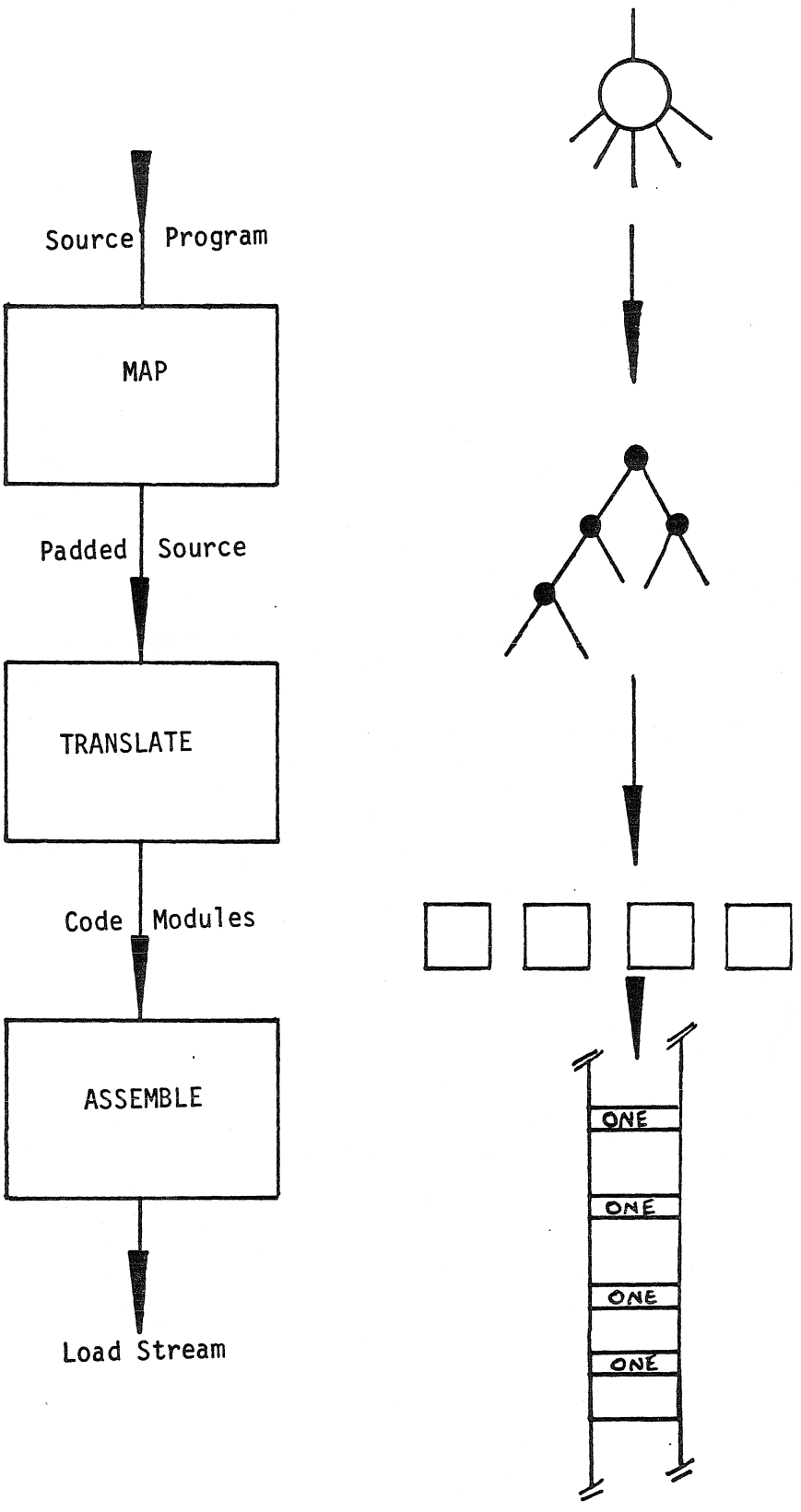


Figure 7.3.1 The Compilation Process

this problem at the Technical University of Eindhoven, Holland. Some of the pitfalls mentioned are ones I found myself. Others were pointed out by Martin.

First, a processor definition is modified to communicate along the two physical internal ports. The definition is parsed by the compiler in order to count the number of internal ports defined and to identify the message statements and guards. The internal ports are mapped onto the two physical ports of a real processor. Half are assigned to the left port and the others are mapped onto the right port. The port assignments alternate between the left and right side. Thus if an array c(1:5) of internal ports is specified and c(1) is assigned to the left port, then all of the internal ports selected with an odd index (c(1), c(3), c(5)) will use the left port, and the others will use the right port.

After all of the logical ports are mapped onto physical ports, the message statements and message guards are massaged to reflect the use of physical ports. The transformation of message statements is straightforward. If the specified port has a simple name, like B, C, and reject, the mapped port is used directly. If broadcast mode communication is specified, all of the appropriate ports are selected. If an expression is used to select one of an array of ports, the message statement is replaced with a conditional statement. For example, if the message statement c(i)!foo is encountered and c(1) has been assigned to the left port, the following conditional statement replaces the original message:

```
[ odd(i) --> L!foo((i+1)/2)
! even(i) --> R!foo(i/2)
]
```

Notice that an argument has been appended to the message foo. This argument allows the message to be routed to its original destination. It is appended when the ports have simple names as well. Broadcast communication, on the other hand, does not require a routing argument, since it involves all of the processors.

The transformation of message guards is more difficult. Suppose we continue the technique described above and replace the single message guard with a pair:

```
[ odd(i) and L!oops((i+1)/2) -->
! even(i) and R!oops(i/2) -->
]
```

How do we get the argument to message foo into the act? Communication with a specific port is desired, yet the port is encoded as data in the message. It cannot influence the control of the statement, that is, the guard.

Another problem with either form of communication arises when broadcast mode is used for input. The message c(*) ?foo means that the message foo can come from any one of the internal ports. In particular, the message will be accepted from only one of the ports. Yet these five ports are spread out over two physical ports. It is easy to guarantee that only one of the ports mapped onto a particular physical port will respond, but how do we make sure that only one among all of them is accepted? This problem and other related mutual exclusion problems are difficult.

In the preceeding discussion we looked at the mapping of a composite processor definition onto a physical processor. But we have yet to produce any padding processors. The second step of the mapping process is to generate some source code for the padding. The compiler makes two additional processor defintions, one for each of the right and left physical ports. Each one has a logical fanout equal to the number of processors mapped onto its parent port in the previous step. The body of statements in each padding processor definition consists of a loop with message guards. There is one guard for each communication on the parent port. Messages sent or received through the parent port may have an appended argument for routing purposes. Messages that involve the internal ports of the padding processor will not have the routing information contained as data in the message. Since the internal ports are once again logical ports, the message is routed directly by port selection.

The result of producing composite padding processors is that they too must be mapped onto physical processors. That is, the mapping is done recursively. The advantage of this technique is that a composite processor can be mapped onto a group of physical processors without knowing the characteristics of those processors that will eventually be connected to its internal ports. The messages that use the internal ports of the composite at any stage in the mapping are identical to those specified in the original definition. The extra routing argument is never present in messages through the visible layer of internal ports.

The third step of compilation produces a load stream. This is a straightforward process, requiring the parsing of the

revised connection plan to identify which physical processors should be loaded with each kind of code segment. I have implemented this phase, together with the code generation phase, in a working compiler.

The last section of Chapter One mentions some extensions that might be made to the notation. I will reiterate only the most important one here. There is no way to capture the overall strategy of the algorithm. The notation is very good at the detail level. We can write programs for each kind of processor, and we can hook the processors together into a tree machine. But how do we know what the program is supposed to do without resorting to reading all the code with paper and pencil in hand. I might add that this problem is not unique to tree machines. But, because the control of a tree machine program is distributed throughout the processors, the intent is harder to capture than in an ordinary programming language. Thus, I believe the next significant contribution to our understanding of concurrent programming will be to find a way of concisely describing what an algorithm is supposed to do. This may be a flow graph of the communication, similar to the work being done by Marina Chen, Eric Barton, and John Williams at Caltech. It may be a by-product of improved proof techniques for concurrent programs. Regardless of the method chosen to provide a functional specification, the key to the specification will be the hierarchical nature of tree machine programs.

There are some hard system problems that go along with having a real machine of fixed size instead of a computational model that is as big as it needs to be. What do you do when the tree isn't big enough? What do you do

when a processor isn't big enough? How do you debug it? How do you tell when one of the processors is broken? All of these questions must be answered when the tree machine moves from the theoretical world to the real world. It should be a very exciting time.

References

[Appel77]
Kenneth Appel and Wolfgang Haken
"The Solution to the Four-Color-Map Problem"
Scientific American 237:4 October, 1977
p. 108-121

[Aho74]
A.V. Aho, J.E. Hopcroft, and J.D. Ullman
The Design and Analysis of Computer Algorithms
Addison-Wesley, Reading, Massachusetts, 1974

[Arlazarov70]
V.L. Arlazarov, E.A. Dinic, I.A. Faradzev
"On Economic Construction of the Transitive Closure
of an Oriented Graph", Soviet Math Dokl. 11:5(1970)
p. 1209-1210

[Armstrong77]
Phillip N. Armstrong
An Investigation of Sorting and Self-Sorting Memory"
Final Technical Report on Smart Memory Structures
California Institute of Technology, 1977

[Backus78]
John Backus
"Can Programming be Liberated from the vonNeumann Style?
A Functional Style and its Algebra of Programs"
C.ACM 21:8, August, 1978 p. 613-641

[Bently79]
Jon Bently and H.T. Kung
"A Tree Machine for Searching Problems
(A Preliminary Description)"
Computer Science Dept.
Carnegie-Mellon University, 1979

[Birtwhistle73]
G.M. Birtwhistle, O-J Dahl, B. Myhrhaug, K. Nygaard
Simula Begin
Petrocelli, New York, 1973

[Browning79a]
Sally A. Browning
"Computations on a Tree of Processors"
Proc. Caltech Conf. on Very Large Scale Integration
January, 1979, p. 453-478

[Browning79b]
Sally A. Browning
"Matrix Inversion on the Tree Machine"
Memo #2806, Computer Science Dept.
California Institute of Technology, May, 1979

[Browning79c]
Sally A. Browning
"Algorithms for the Tree Machine"
in [Mead79], p. 295-313

[Choo80]
Young-il Choo
Hierarchical Design Nets
MS Dissertation, Computer Science Dept.
California Institute of Technology, 1980 (in progress)

[Cook71]
S.A. Cook
"The Complexity of Theorem-Proving Procedures"
Proc. 3rd Annual ACM Symp. on Theory of Computing
1971, p. 151-158

[Dahl72]
O-J Dahl, E.W. Dijkstra, and C.A.R. Hoare
Structured Programming
Academic Press, New York, 1972

[Denny79]
W. Michael Denny, Ernest R. Buley, and Earl Hatt
"Logic-Enhanced Memories: An Overview and Some Examples
of Their Application to a Radar-Tracking Problem"
Proc. Caltech Conf. on Very Large Scale Integration
January, 1979, p. 173-186

[Dijkstra76]
E.W. Dijkstra
A Discipline of Programming
Prentice-Hall, Englewood Cliffs, New Jersey, 1976

[Fischer71]
M.J. Fischer, A.R. Meyer
"Boolean Matrix Multiplication and Transitive Closure"
Conf. Record, IEEE 12th Annual Symp. on
Switching and Automata Theory, p.129-131

[Flores63]
Ivan Flores
The Logic of Computer Arithmetic
Prentice-Hall, Englewood Cliffs, New Jersey, 1963

[Franklin68]
Joel N. Franklin
Matrix Theory
Prentice-Hall, Englewood Cliffs, New Jersey, 1968

[Heller79]
William R. Heller
"An Algorithm for Chip Planning"
SSP Memo #2806, Computer Science Dept.
California Institute of Technology, May, 1979

[Hoare78]
C.A.R. Hoare
"Communicating Sequential Processes"
C.ACM, 21:8, August, 1978, p. 666-677

[Isaacson66]
Eugene Isaacson, Herbert Bishop Keller
Analysis of Numerical Methods
John Wiley & Sons, Inc., New York, 1966

[Keller78]
Robert M. Keller, Gary Lindstrom, Suhas Patil
"An Architecture for a Loosely-Coupled Parallel
Processor", Department of Computer Science
University of Utah, UUCS-78-105, October, 1978

[Knuth73]
Donald E. Knuth
The Art of Computer Programming, vol.3
"Searching and Sorting"
Addison Wesley, Reading, Massachusetts, 1973

[Kung79]
H.T. Kung and Charles E. Leiserson
"Algorithms for VLSI Processor Arrays"
in [Mead79], p. 271-292

[Leiserson79]
Charles E. Leiserson
"Systolic Priority Queues"
Proc. Caltech Conf. on Very Large Scale Integration
January, 1979, p. 199-214

[Locanthi79]
Bart N. Locanthi
"A Taxonomy of Interconnect Structures"
Memo #2764, Computer Science Dept
California Institute of Technology, April, 1979

[Locanthi80]
Bart N. Locanthi
The Homogeneous Machine
PhD Dissertation, Computer Science Dept.
California Institute of Technology, 1980

[Mago79]
Gyula A. Mago
"A Cellular, Language Directed Architecture"
Proc. Caltech Conf. on Very Large Scale Integration
January, 1979, p. 447-452

[Mead79]
Carver Mead and Lynn Conway
Introduction to VLSI Systems
Addison-Wesley, Reading, Massechusetts, 1979

[Munro71]
Ian Munro
"Efficient Determination of the Transitive Closure
of a Directed Graph"
Information Processing Letters 1(1971) p. 56-58

[Rem79a]
Martin Rem
"Mathematical Aspects of VLSI Design"
Invited presentation, transcript included in
Proc. Caltech Conf. on Very Large Scale Integration
January, 1979, p.55-64

[Rem79b]
Martin Rem
personal communication, spring 1979

[Strassen69]
V. Strassen
"Gaussian Elimination is not Optimal"
Numer. Math. 13(1969) p.354-356

[Tarjan77]
R.E. Tarjan
"Reference Machines Require Non-Linear Time to Maintain Disjoint Sets"
Proc. 9th Annual Symp. on Theory of Computing, 1977

[Tarjan78]
R.E. Tarjan
"Complexity of Combinatorial Algorithms"
SIAM Review, 20:3, July, 1978

[Thompson79]
C.D. Thompson
"Area-Time Complxity for VLSI"
Proc. Caltech Conf. on Very Large Scale Integration
January, 1979, p.495-508

[Thurber75]
Kenneth J. Thurber
"Associative and Parallel Processors"
Computing Surveys, 7:4, December, 1975, p. 215-255

[Turing36]
A.M. Turing
"On Computable Numbers, with Application to the Entscheidungs Problem"
Proc. London Math. Society, 2:42(1936), p. 230-265

[Warshall62]
S. Warshall
"A Theorem on Boolean Matrices"
J.ACM 9:1, January, 1962, p.11-12

[Wilner79]
Wayne T. Wilner
"Recursive Machines"
Xerox Palo Alto Research Center
(submitted to IFIP '80)