Tree Machines: Architectures and Algorithms
A Survey Paper

Hussein A. H. Ibrahim

CUCS-119-84

Department of Computer Science
Columbia University

Abstract

Recent advances in very large scale integrated (VLSI) circuit technology have lead to a surge in research aimed at finding new computer organizations that support a great deal of concurrency. Computer organizations based on tree structures appear well-suited to several kinds of parallel computations. In this paper we will discuss the performance of tree machines as well as issues related to their implementation in VLSI. Examples of tree machines are presented, with an emphasis on the way the processing elements communicate in the machine. A taxonomy of tree algorithms based on a taxonomy of parallel algorithms proposed by Kung in 1979 is introduced. Examples of tree algorithms are also given.

Table of Contents

List of Figures

1. Introduction

Recent advances in very large scale integrated (VLSI) circuitry allow us not only to make current processors faster and smaller in size, but also to embed a number of processing and memory elements within a single chip in a cost-effective manner. This has lead to a surge in research directed toward finding radically new computer organizations that support a high level of concurrency. In conventional von Neumann machines, all communication between the processing site (usually a single processor), and the memory site proceeds along a relatively low bandwidth communication path, which is a serious bottleneck in conventional machines [Back 78]. Intermingling data and processing elements in VLSI technology allows many local computations to be executed concurrently where the data elements reside. Consequently, the bottleneck of conventional machines is eliminated, and the amount of data that can be processed in a given period of time is greatly increased.

Certain computational tasks arising in many applications may be divided into smaller cooperating tasks that can be performed concurrently. The results computed by these smaller tasks may then be combined together, usually in a hierarchical manner, to compute the required result. Tree-connected processing elements (PE's) provide a structure that allows us to exploit the hierarchical nature of this class of computations. Hierarchical systems also enjoy the properties of local communication and regular interconnection patterns which are desirable for VLSI implementations. A hierarchical communication system is also necessary for efficient global communication in VLSI technology when the number of elements connected to the global communication system is very large [Mead 79]

A tree machine can be informally described as a collection of processing elements interconnected to form a complete binary tree. Tree machines, besides enjoying the performance characteristics of hierarchical structures, also capitalize on the properties of VLSI technology. They can be laid out optimally on the plane, and they do not suffer from pinout limitations, as we will see later in this paper.

In 1970, Lipovski proposed a tree-organized associative machine [Lipo 70] consisting of small PE's connected in the form of a binary tree, each with a small amount of

memory and logic. A description of this machine will be given in Section Four. Berkling [Berk 71] later proposed another architecture for a computing machine that implemented in hardware a hierarchical execution of programs that was modeled after Turing tree machines. With the recent emergence of VLSI technology, tree machines have begun to attract attention again as an interconnection scheme for multiprocessor systems. Several projects now underway at different universities are directed toward the construction of tree machines. Among these are the X-Tree machine project at the University of California at Berkeley, the tree machine project at California Institute of Technology, the NON-VON Supercomputer and DADO machine at Columbia University, the Stony Brook tree machine at the State University of New York at Stony Brook, the tree machine at the University of North Carolina at Chapel Hill [Toll 81c], the DDMn Machine at University of Utah, and the DAC project at the University of Southern California [Horo 79].

In Section Two we discuss some of the most important performance issues related to tree machines. In Section Three some implementation issues will be considered. Section Four presents some of the architectures that have been proposed for implementing tree machines, and discusses their differences. Certain special purpose tree machines will also be described in this section. In Section Five, we discuss some of the algorithms that run efficiently on tree machines.

2. Performance Considerations

The tree machine architecture is capable of providing a rather general purpose computing environment. A tree machine can be programmed to execute a varied collection of algorithms that take advantage of the tree structure explicitly, along with many other algorithms that exploit its less obvious advantages in the realization of large scale computational concurrency. We will introduce some examples of these algorithms in this section. A detailed discussion of some of the algorithms that run on tree machines will be presented in Section Five. In the following examples we assume that the tree machine consists of processing elements connected together in the form of a binary tree, and that each PE contains some memory storage and can perform simple arithmetic and logical operations on data stored in its local memory. Each PE can send data to and receive data from its children and its parent. The PE at the root of the tree can communicate with the external world.

Among the algorithms that run efficiently on tree machines are algebraically associative operations such as the computation of the sum, the maximum, or the mean of n numbers. Such operations can be performed in $O(\log n)$ time, provided the data elements are already present in the tree. In these algorithms, data elements are stored initially in the PE's at the leaves of the tree (*leaf processors*). The answer is obtained by letting each PE combine the values found in its children using the algebraically associative operator (adding two numbers in the sum algorithm, for example). This step is repeated $\log n$ times (the number of levels in the tree), after which the root of the tree contains the final result (the sum of the n numbers, in the sum algorithm).

Search, insert and delete operations on sets can also be performed in $\log(n)$ steps where n is the number of data elements in the set [Bent 79a]. In the case of the search algorithm, for example, the algorithm starts by inspecting the data held by the root processor. If the target of the search is found, the algorithm is terminated. Otherwise, the root processor initiates the search in both of its children. In the worst case, this operation is repeated $\log(n)$ times. Note that the tree connections are used explicitly in such algorithms to obtain the required results.

A tree machine in which all processors simultaneously perform the same instruction on their respective local data items is said to be executing in single instruction stream, multiple data stream (SIMD) mode [Flyn 72]. Similarly, algorithms performed in this mode are called SIMD algorithms. Tree machines are capable of the very rapid execution of SIMD algorithms, which require this kind of global broadcast. The efficiency of tree machines in executing SIMD algorithms drives from the fact that time required for global broadcast in trees is proportional to the number of levels in the tree, and is thus logarithmic in the number of PE's. Few other architectures share this property. Two dimensional orthogonal meshes, where PE's are connected together in the form of a rectangular grid, have a global broadcast time of $O(n^{1/2})$, where n is the number of PE's in the mesh. For ring and linear array architectures, the broadcast time is of $O(n)$. The global broadcast time in trees can be reduced further by propagating the data coming from the parent to the children without latching it in a clocked manner. This in effect will turn the tree bus into a combinational logic circuit during global broadcast. A global broadcast of this kind was proposed by Lipovski for his Associative Tree Machine [Lipo 70] and is being implemented in the NON-VON Supercomputer [Shaw 82]. It is expected that the time needed to broadcast a NON-VON instruction to all PE's in a tree of 20 levels (one million PE's) will be about one microsecond.

Highly efficient SIMD algorithms exist for a wide range of important tasks. Examples include content-addressable (or *associative*) operations, relational database primitives such as SELECT, PROJECT, and JOIN, and many numerical tasks drawn from such diverse areas as signal processing, physical simulation, and low-level computer vision. We will now show how a content-addressable search can be performed using this kind of global broadcast. We assume for simplicity that each PE holds a record of data elements in its local memory. We globally broadcast the sequence of instructions for finding the data element we are looking for, and all the processors look for this data element concurrently. The search is successful if at least one processor finds the data element in its own local memory. We will defer discussion of how it may be determined whether a given search has terminated successfully.

Many other algorithms require a time complexity proportional to the number of data elements in the problem when performed on trees. Sorting, for example, can be done in linear time [Song 81], and many matrix operations in $O(m)$ [Brow 79], where m is the number of elements in the matrix. Algorithms have also been proposed to simulate three-dimensional physical systems on trees in $O(n^{2/3})$ time [Shaw 82] where n is the number of points in a discrete approximation of the space being simulated. Problems that are solvable by divide-and-conquer techniques may also be well suited to tree machine architectures, as are many problems having natural recursive definitions [Rem 79].

Algorithms that require extensive input and output operations can suffer from delays, if the input and output operations are confined to the root processor. There are several solutions to this problem, which involve the distribution of input/output operations between many PE's, as will be shown when tree machine architectures are described. Also, algorithms that require a significant amount of communication between arbitrary PE's, such as arbitrary permutation of data elements, are not amenable to efficient execution on tree machines. The reason behind this is that exchanging data elements between two PE's in the tree requires the data elements to travel up the tree until the lowest-common-ancestor of the two PE's is reached. In case of extensive communication between PE's, the highest nodes in the tree may become the bottleneck of execution. The worst case occurs when the data elements in the right and left subtrees, rooted by the root processor, are to be exchanged. The root processor becomes the bottleneck in this case, as all the data elements have to pass through it. One solution to this problem is to add extra connections between PE's to reduce the traffic going through higher level PE's, as will be described when we discuss the X-tree machine.

Another issue that relates to the performance of concurrent systems is the time needed to access a specific element in the system. Access time for any element in the tree is O(log n), and the maximum communication time between any two individual elements in the tree is of the same order. Horowitz [Horo 81] presented an algorithm to route messages in a binary tree machine. In his algorithm he introduced extra links to shorten the average path length between the tree nodes, and to provide for fault tolerance in case one of the tree processors is not functioning.

Tree structures also enjoy the property that many assertions about the system can be proven by induction over the hierarchy [Rem 79]. Tree machines are also easily testable if a single processor is testable [Mead 79]. First, we test the root PE, and if it is working correctly we load the same test program in its children and exercise them. This process is repeated at each level of the tree.

3. Implementation Issues

VLSI devices have been decreasing in size and increasing in speed at a rapid rate over the last decade. It is estimated that in the late 1980's, we will have chips containing millions of devices. As the dimensions of circuit devices scale down, communication delays in wires that carry control and data to functional blocks in an integrated circuit will become a dominant factor. In many cases, much of the area on the chip is likely to be occupied by these communication lines. Another important factor in current VLSI technology is the relatively small (and only slowly increasing) number of pins each package can have. Tree machines are compatible with these properties of VLSI by virtue of their local communication, regular interconnection pattern, area-efficient layout, and limited number of external ports. We will start our discussion of the VLSI implementation of tree machines by showing how a tree machine may be laid out on chip. Next, we will discuss the pinout limitations imposed by current technology, and show how tree machines avoid these limitations. We end the section by presenting a technique proposed by Leiserson [Leis 81] for implementing tree machines using just one kind of chip.

3.1 Layout of Tree Machines

In Figure 3.1, we have illustrated a space-economical layout proposed by Mead and Rem [Mead 79] for a complete binary tree. This layout is known as the *hyper-H* embedding of the complete binary tree. The hyper-H embedding is highly regular, and the silicon area occupied by the tree is proportional to the number of PE's per chip. Horowitz presented an algorithm for embedding arbitrary (possibly incomplete) binary trees in the plane [Horo 81]. The algorithm assumes the wires connecting the processors to be straight, and of unit width. The algorithm also assumes the processors to be of unit area. In the case of a complete binary tree this algorithm yields the hyper-H embedding of the tree. The area required to layout the tree in this case is proportional to n, where n is the number of PE's in the tree. The worst case for the Horowitz algorithm is a binary tree skewed to the right or the left as shown in Figure 3-2. The algorithm embeds this skewed tree in area $O(n^2)$, where n is the number of PE's in the tree. The above discussion assumes that it is sufficient to allow the tree to communicate with the external

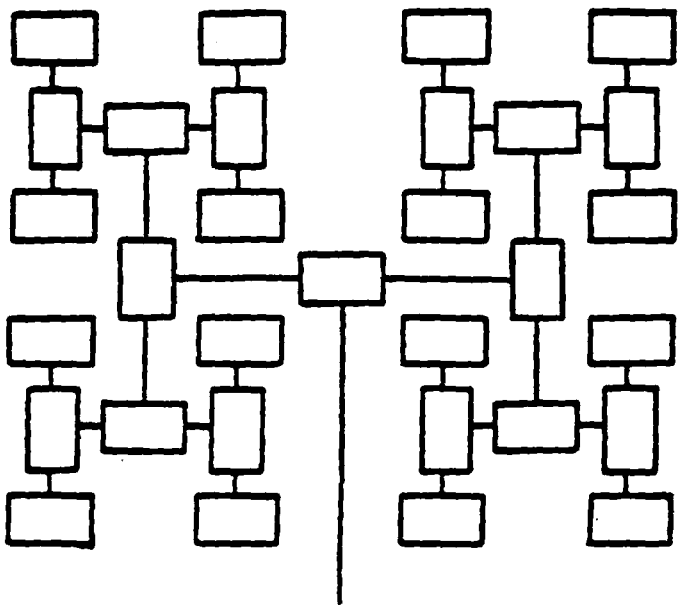

Figure 3-1: Hyper-H Embedding of a Binary Tree

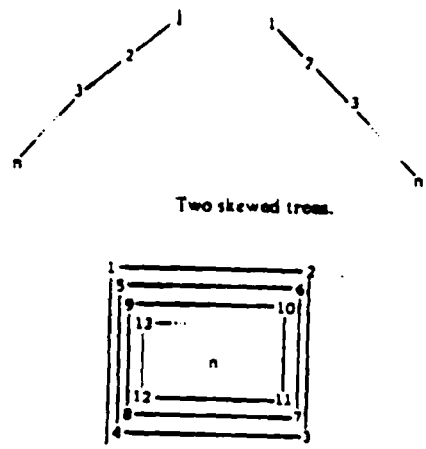

Figure 3-2: The Mapping of a Right Skewed Binary Tree

world through the root only. The hyper-H embedding is optimal in this case. If we assume that the tree must also have its leaves accessible along the perimeter of the chip, then the area needed to embed the tree is at least $O(n \log n)$ as shown by Brent and Kung [Bren 79].

By way of comparison, the two-dimensional orthogonal mesh-based architectures, in which each PE is assigned to one of the points of a rectangular mesh and connected to its four neighbors, are as area efficient asymptotically as trees, but suffer from pinout limitations, as we will see in the next subsection. Shuffle-exchange and cube-connected cycles architectures [Ston 71], to cite another example, require an area of at least $O(n^2/\log^2 n)$ [Thom 80].

3.2 Pinout Limitations and Tree Machines

Over the coming years, the number of pins per chip package is not expected to increase at the same rate as the number of active devices within the chip. The limited number of pins per package, places a severe constraint on the bandwidth available for communication with the external world, and thus represents a physical bottleneck. Architectures having a small fixed number of external ports per chip, independent of the number of PE's per chip, are for this reason particularly suitable for VLSI implementation. Binary trees are quite attractive in this regard. Every processor in a binary tree machine communicates with exactly three other processors: its parent, left child, and right child.

If we were to build the tree machine using a single PE per chip, the number of ports per chip would be three. If the tree machine is to have more than one processor per chip, at least two implementation methods are possible. The first uses two kinds of chips, as shown in Figure 3-3. Chips that contain the leaf nodes (leaf chips) will have only one port. The leaf chips are not pin-limited, and as devices on chip scale down, we can embed unlimited number of PE's on each chip Chips that contain only internal nodes (internal chips), on the other hand, will have a number of ports depending on the number of nodes they contain. These internal chips are thus pin-limited, and can not contain more than a fixed number of processors, even as dimensions scale down.

Another way to implement tree machines, suggested by Leiserson [Leis 81], employs only one kind of chip, which is discussed and illustrated in Section 3.3. In the Leiserson scheme, every chip has exactly four ports, independent of the number of processors it contains. Tree machines implemented using this scheme have a fixed

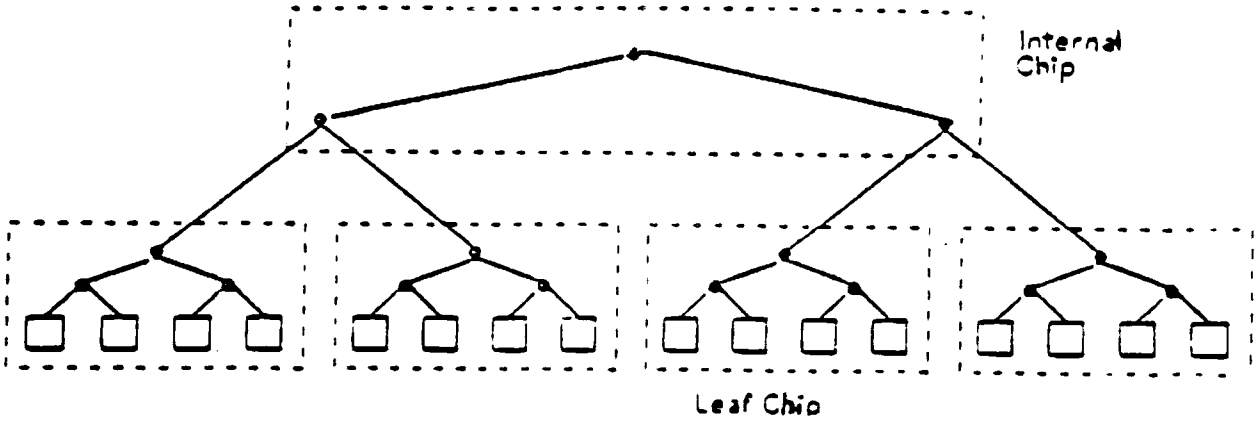

Figure 3-3: Implementing Tree Machines Using Two Kinds of Chips

number of pins per chip, and can have as many PE's as the dimensions of VLSI devices allow. Linear array machines, which consists of an ordered set of PE's, each connected to its immediate predecessor and successor, share this property with tree machines. Two-dimensional orthogonal mesh-based architectures have a number of external ports equal to $4\sqrt{n}$ where n is the number of PE's. Shuffle exchange and cube-connected cycles architectures have even more external ports asymptotically than meshes.

3.3 The Leiserson Scheme for Tree Machine Layout

The tree architecture gives rise to IC's that have a highly regular interconnection structure, local communication, and many repetitions of a single processor design. These IC's can then be assembled in regular patterns at the printed-circuit (PC) board level to construct machines with thousands to millions of processors. A scheme for implementing binary trees using a single type of chip and regular inter-chip connection pattern was suggested by Leiserson. In this scheme, a complete subtree and a single interior node are embedded on each chip as shown in Figure 3-4. There are four ports per chip labeled T, F, L, and R. The T connection leads to the root of the subtree, while F, L, and R connect the single interior node to its parent, left child, and right child, respectively. A simple recursive procedure allows the construction of a complete binary tree as shown in Figures 3-4 and 3-5. The area required for routing wires within the PC board is proportional to the number of chips per board, leading to efficient use of PC boards of arbitrary size.

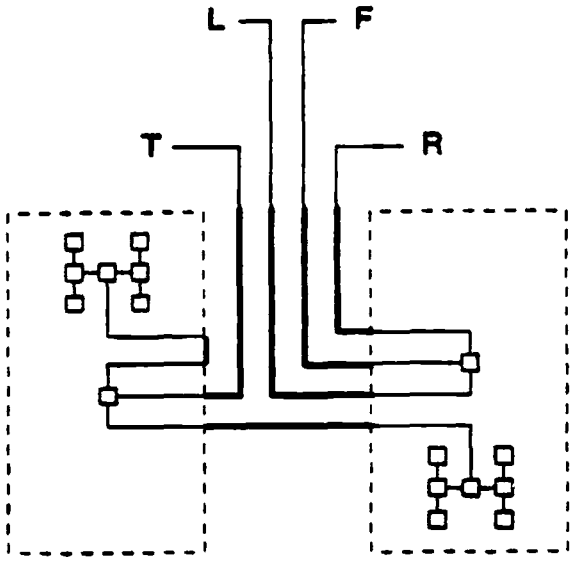

Figure 3-4: Interconnection of Two Leiserson Chips

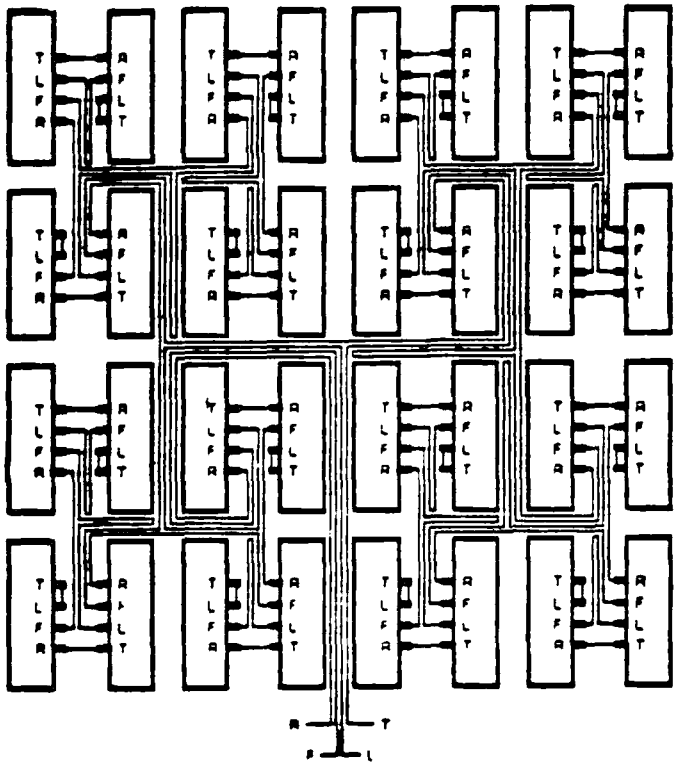

Figure 3-5: Leiserson Layout: The Printed Circuit Board

4. Some Tree Machines

In this section we will discuss some of the most important tree machines designs that have been proposed, with an emphasis on their differences. Some of these are, at present, merely "paper machines"; others are in various stages of implementation.

4.1 The Tree Channel Processor

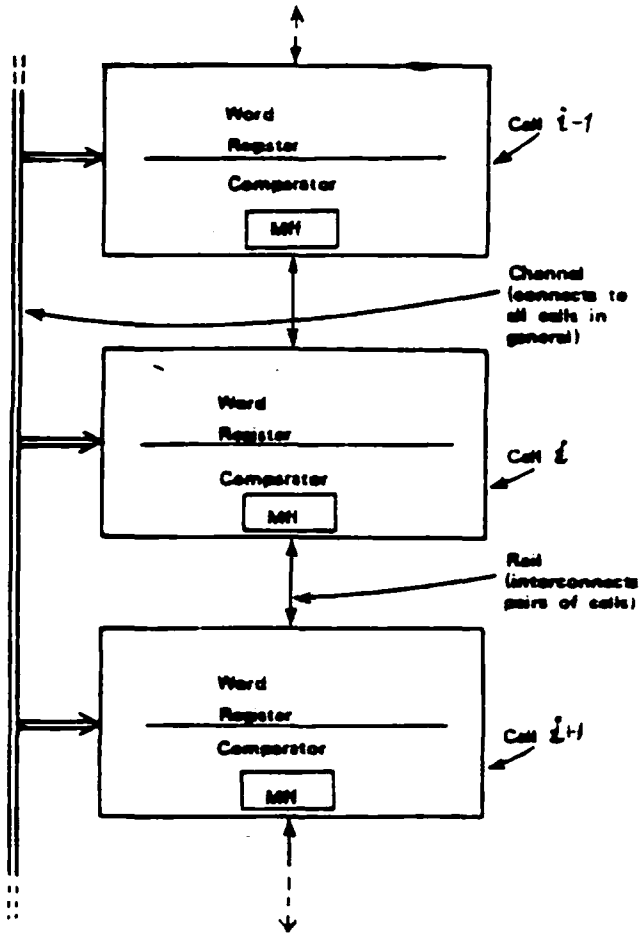

Figure 4-1: A Linear Array Processor

The Tree Channel Processor (TCP), proposed by Lipovski [Lipo 70], is an associative machine that was designed to emulate a class of machines the author called "linear array processors", which are based on a linear array of associative memory cells. Figure 4-1 shows a linear array processor. Each associative cell has a single fixed-size word of memory and a comparator. The linear array processor

was intended primarily for information storage and retrieval. "Rails" that connect consecutive cells are used to broadcast data and instructions, and to aggregate the results of associative search. The linear array, however, suffers from excessive propagation delay, difficulty of segmenting the processors to execute subprograms, and a susceptability to faults resulting from rails stuck at one logical level.

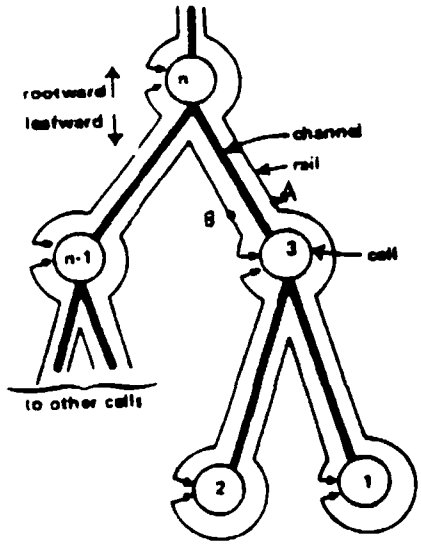

Figure 4-2: A Subtree of the Tree Channel Processor

Lipovski designed the TCP interconnection scheme of figure 4-2 to solve these problems. The processor has a single "channel" and two identical rail complexes. Global broadcast to all cells is performed simultaneously via the channel in the tree branches. Each cell amplifies the channel signal and propagates it to its two children. The rail communication makes the processor cells appear to the programmer as an ordered one-dimensional array able to detect subsets and substrings, to count elements, etc. The maximum delay through the channel or the rails is of $O(\log n)$. That delay determines the cycle time for the tree.

Communication with the external world is performed through input and output channels that are connected to selected leaf nodes. The tree can be partitioned into separate modules called instruction domains (ID's). An instruction domain is established when a cell is disconnected from the channel by setting one of its mode flags. The whole subtree rooted by the separated cell becomes a new ID. The

different ID's normally act independently. They can issue commands to delimit the channel or reconnect it. If a cell in a given ID needs to use an I/O channel connected to a leaf cell in another ID, the two ID's must be reconnected and whatever programs they are executing must be halted until the request is satisfied. A switching network at the root of the tree is used to accommodate these functions.

The cells at the tree nodes can function in three different modes, which are invoked by setting mode flags within each cell. All cells in a specific ID operate in the same mode. The three modes are:

1. *Run mode*, which involves normal execution of instructions for information retrieval.

2. *Transfer mode*, which provides for efficient loading and unloading of the tree.

3. *Supervisor mode*, which enables channel-delimiting cells to be set up or changed.

The TCP presented solutions to several problems associated with architectures having a very large number of processors. The machine is segmented to allow different problems to run concurrently. It has a small propagation delay, and provides for fast loading and unloading of data in parallel from different I/O channels

4.2 The Caltech Tree Machine

The Caltech tree architecture comprises a collection of small, identical processors, each with some local storage, connected to form a binary tree [Brow 80]. The machine relies on local communication between parents and children only; there are no global communication paths. The machine operates in multiple instruction stream, multiple data stream (MIMD) mode [Flyn 72], with PE's independently executing programs stored in their respective local memories. Processors communicate by means of message-passing protocol. We will start by describing the processor architecture, and will then show how communication between processors is accomplished.

4.2.1 The Processor Architecture

Each processor contains a small amount of program memory, a few data registers, some communication handlers, and some logic to provide general arithmetic and logical capabilities.

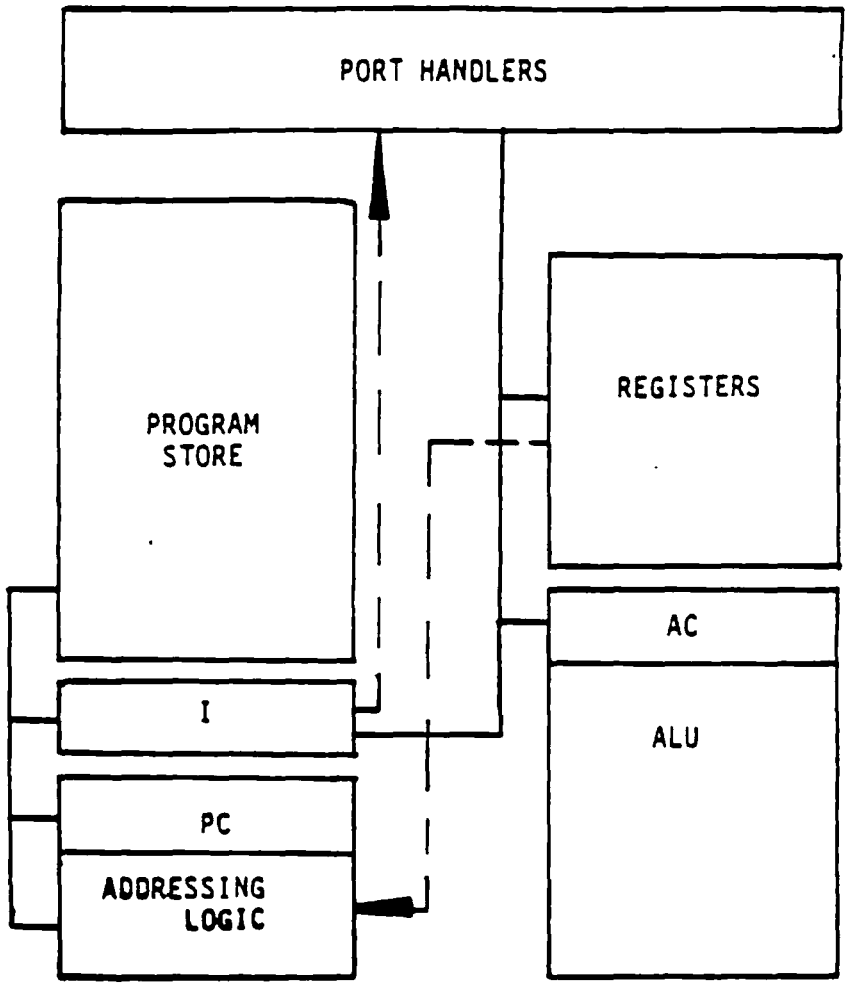

Figure 4-3: The Caltech Tree: Processor Architecture

Figure 4-3 illustrates the processor architecture. Underlying this architecture is the belief that increased functionality does not justify the requisite increase in chip area, and that there is a tendency on the part of the programmer not to exploit all available concurrency if there is a rich set of instructions and a powerful processor

[Brow 80]. The processor has 256 bytes of program store and 16 byte registers for storing data. Only addition, subtraction, shift operations, and logical operations are provided by the ALU. Multiplication, division, and floating point operations are all performed in software. The accumulator, AC, is a source, and the only destination, for the ALU. The I register holds the instruction being executed, while the PC register points to the next instruction to be fetched for execution. The instruction set provides only a direct addressing scheme. There are three main categories of instructions in the machine: control flow instructions, communication instructions, and data flow instructions. There are seven control flow instructions: HALT for normal termination of the program, ABORT for abnormal termination of the program, SKIP for skipping an instruction, and four instructions to implement conditional and unconditional branching within the program. Four communication instructions are provided: two for input and two for output. Data flow instructions are used to transfer data between the data registers and the accumulator, and to perform arithmetic and logical operations. There are 14 instructions in this category.

The tree machine is programmed in a high level language that resembles Hoare's "Communicating Sequential Processes" (CSP) notation [Hoar 78]. The Tree Machine Programming Language (TMPL), as proposed by Browning [Brow 80] allows the programmer to write programs for trees with arbitrary fanout. During compilation these programs are transformed into source code for a binary tree by a program called MAP [Brow 80]. The basic building block of TMPL is the processor definition. Each definition describes a self-contained computational unit that communicates with other processors through named external ports [Brow 81].

The tree machine is loaded through the external port of the root processor and the loaded code travels down the tree to the leaves. The code stream consists of a header and the code itself. The header specifies the length of the code, the length of the destination processor address, the address of the destination processor, the initial value to be placed in PC, and an instruction code (opcode). The PE's are uniquely identified by a bit string that grows in length with the depth of the tree. The addresses are assigned in such a way that a child address differs from its parent address in only one bit, so that there is no need to store the PE's address in its own local memory. Instead the parent is able to decide where to direct the

code message on the basis of the value of a specific bit in the target address. The opcode can be any of the following:

1. ONE, which means the code is to be loaded in a specified processor.

2. TREE, which means the code will be loaded in the entire subtree, rooted by the processor that is addressed by the opcode.

3. LEVEL, which directs the code to all PE's at the level specified by the processor address in the header.

4. YOU, which is used by a parent to force the loading of its children.

During loading each PE looks at the opcode. If it is YOU, the PE will load the code into its own memory. If it is TREE or ONE, the PE will look at the address, and if it has a nonzero length, will remove the leading bit of the address and pass the header and the code to one of its two children depending on the value of this bit. If the length is zero, then it will load the code stream into its own memory. In the case of TREE, the entire subtree beneath the processor must be loaded with the same program. This is done by passing the code and the address field equal to zero to the children of the processor. If the opcode is LEVEL the PE will do the same thing it did in the case of TREE, except that it will examine the length of the address rather than the address itself.

4.2.2 Communication in the Caltech Tree Machine

In each PE there are three groups of communication handlers, one for each of the PE's three ports. These handlers manage message traffic, load programs, and pass code to their descendants. The definition of the processor in TMPL includes a named *external* port to communicate with its parent, and an arbitrary number of named *internal* ports to communicate with its children. Communication statements in TMPL specify the port name through which the message is to pass instead of naming target processors. Inter-process communication can be specified in two ways: either by an imperative statement or a conditional expression [Brow 81]. The conditional form appears as a part of a loop or within a case statement. On executing these statements, the processor is blocked until communication is successfully terminated. The conditional form is executed only if both PE's

communicating along the specified port are ready to exchange a message. The general form for a message statement is the same as that of an expression in CSP:

[port or list of ports] [? or !] message(arguments)

where ? indicates input and ! indicates output. Two processors will communicate when an output request to a port from one PE matches an input request for the same message from the other processor. In order to avoid deadlock, the restriction is made that either the output or the input can be done conditionally, but not both

The type of message, *imp* for imperative and *cond* for conditional, must be specified so that the compiler can detect illegal communication operations. For examples and more discussion the reader is referred to [Brow 81].

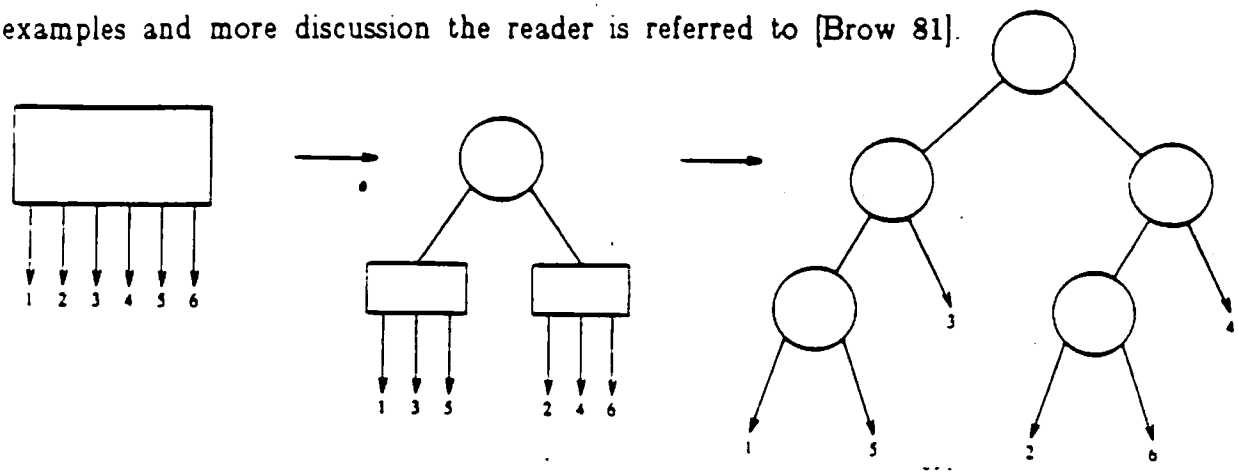

Figure 4-4: Mapping Arbitrary Fanouts onto a Binary Tree

Arbitrary fanouts are mapped into the binary tree by using several layers of the tree to provide the required number of descendants, as shown in Figure 4-4. The intermediate PE's are called *padding* PE's, and are provided with a skeletal program that allows them to simply pass messages between parent and children.

Some of the algorithms that run on the Caltech tree machine will be described in Section Five.

4.3 The X-Tree Machine

The X-Tree machine architecture was formulated at the University of California at Berkeley. It is organized as a full binary tree of multiprocessors known as X-nodes. The objective of the X-Tree project is to define a modular component from which general-purpose computing systems of arbitrary size could be constructed [Sequ 79].

A binary tree was chosen as the most advantageous way to interconnect these components. The binary tree is enhanced with extra links to form a half-ring or full-ring tree as shown in Figure 4-4. These extra links are employed to provide fault tolerance, to shorten the average path length between tree nodes, and to provide a more uniform distribution of message traffic throughout the tree.

Each of the X-Tree PE's was intended to have as much memory as could fit on a single chip with the processor itself. Each X-node thus consists of a single chip computer with as much memory as technology permits. This memory is used for local storage, acting as a cache for the secondary memory, which is connected to the leaf nodes only. Having a large memory in each node minimizes the need for swapping pages of memory to and from the secondary memory.

We will start by discussing communication in the X-Tree, and then discuss the architectural features of a single X-node.

4.3.1 Communication in the X-Tree

All communication in the tree is in the form of messages. To obtain a specific block of data from that portion of the secondary memory which is attached to a given leaf node a message is sent to this node, requesting the desired data block. A channel is then established between the two nodes using a special message header. After communication is ended, the channel is disabled by another special command. Addresses are assigned to the X-nodes such that the children of node n have node addresses $2n$ and $2n+1$.

The half- and full-ringed trees give rise to simple routing algorithms. Decisions about routing messages are made locally at each node depending on the destination address and the current node address. Starting from the root, a message can travel anywhere in the tree by letting the nodes examine the sequence of bits in the destination address, compare them to their own local address, and route the message depending on the result of this examination. To go from any arbitrary node to another node in a binary tree the message will move up in the tree until a common ancestor of the two nodes is found. This common ancestor shares its node address bits with the leading bits of the two nodes. In the case of a full-ring tree it

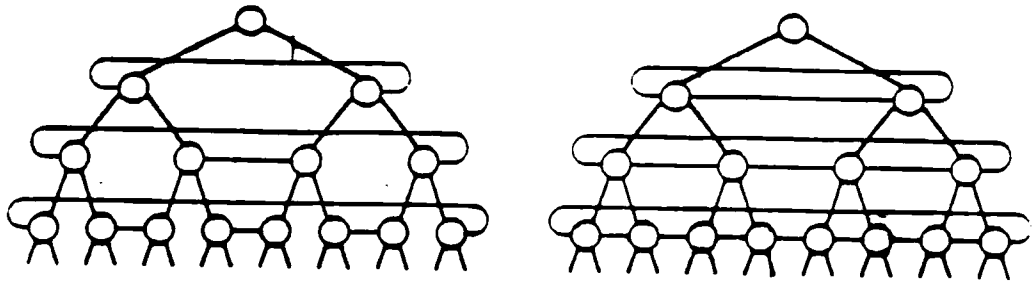

Figure 4-5: A Binary Tree with Full- and Half-Ring Connections

is always shorter to travel along the ring connections when the horizontal distance between nodes is four or less; otherwise, the message moves up the tree. There is a routing controller at each node, which handles in software the messages coming into the node. The routing controller in a full-ring tree compares the address of the incoming message to its own node address, and then acts in the following manner:

1. If the two addresses agree, then the node is the intended destination and the message has reached its destination. The destination node examines then incoming message to determine what action to perform.

2. If the destination lies higher in the tree, then the message is routed upward. This is the case if the destination node address is shorter than the current node address.

3. If the destination node lies at the same level or lower, then the horizontal distance is computed by subtracting the current node address from the destination node address. If this distance is less than four, the message is routed to a ring connection. If the distance is greater than four, the message is routed upward. When the horizontal distance is zero, the message is routed downward.

In case the message can not be routed in the direction of its primary choice because that link is broken, the routing controller will try the second and third choices that are predefined for it. This fault tolerance scheme was simulated and found to yield reasonable results. To prevent the possibility of a message circulating forever, the number of rejections the message encounters is counted and the

message is purged once this count reaches a predetermined number. The count is kept in a byte that travels with the message.

Routing in half-ring trees can be done using the same algorithm as for full-ring trees, except that when a full-ring link is non-existent, the second choice is used to route the message. This algorithm for half-ring trees is slightly less than optimal [Sequ 78]. An optimal algorithm for half-ring trees is shown in [Sequ 78].

Whenever the tree is expanded, a storage device or a terminal may have to be moved, thus changing its node address. A mechanism is thus required to keep the current addresses of data items in this case. To achieve this, messages to leaf nodes are tagged, and leaf processors are marked differently from non-leaf PE's. New nodes are attached the left child positions of, existing leaf nodes. When a message directed to a leaf node reaches a destination node that is not a leaf node, the destination node directs it to its left child. This process continues downward along the left child chain until the message reaches the leaf node.

4.3.2 The X-node architecture

The X-node is a simple microcomputer that communicates with four or five nearest neighbors. In addition to the processor, each X-node contains a self-controlled switching network with its own I/O buffers and controllers. This enables computation to occur concurrently with communication. The processor is attached to the switching network in the same manner as are the other nearest X-node neighbors, as shown in Figure 4-6.

The switching network consists of a time-multiplexed bus with five or six attached ports. Each port consists of an input buffer, an output buffer, and the finite state machines necessary to control them. The internal communication bus consists of a data bus and an address bus that carries slot and port addresses. The bus is allocated in a fixed round-robin manner to all attached ports.

The X-node processor architecture is intended to support high level languages directly in hardware [Patt 79]. This leads to smaller programs, and consequently to more efficient use of the on-chip memory. The X-node processor is also intended to

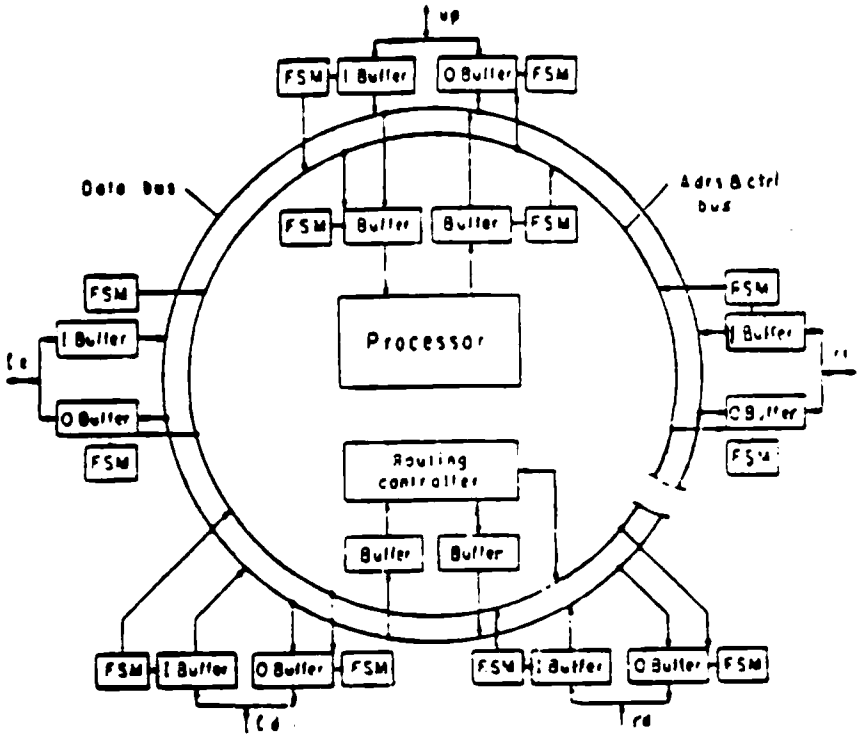

Figure 4-6: Block Diagram of an X-node

support dynamic microprogramming. This helps optimize the execution of code for certain applications in particular nodes. The microcode is stored in the on-chip memory.

The X-node processor requires a substantial amount of memory to minimize the frequency of page faults and subsequent paging traffic between the X-node and the secondary storage [Patt 79]. A high density implementation of the node memory is thus required. Programs, data, and microcode are stored in different areas inside the X-node memory. Separate caches are employed to handle these three types. Thus, parallel memory references to data, programs, and microcode can be achieved. This on-chip memory hierarchy also includes a cache attached to a separate dedicated ALU designed for address calculations.

Sequin and Fujimoto proposed adding a separate chip at each node dedicated to performing the communication needed between the X-nodes. They called the new node the Y-component. For more information on the Y-component the reader is referred to [Sequ 82].

4.4 The NON-VON Supercomputer

The NON-VON (non-von Neumann) Supercomputer [Shaw 82] is currently being implemented at Columbia University. Its architecture includes a tree-structured Primary Processing Subsystem (PPS) based on custom nMOS VLSI circuits, along with a Secondary Processing Subsystem (SPS) based on a bank of intelligent disk drives. Figure 4-7 shows the top level organization of a simplified initial version of the NON-VON machine that is now under construction.

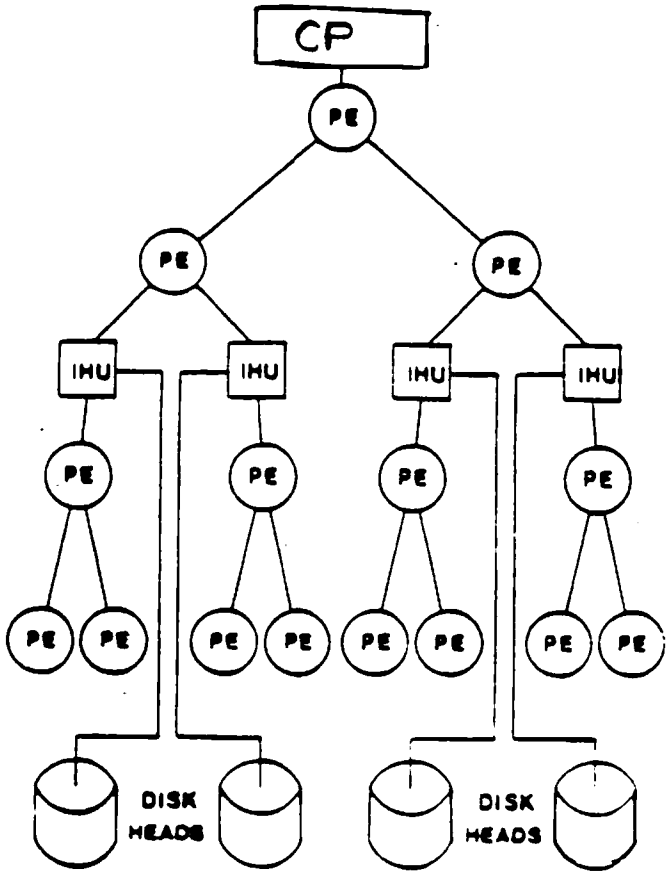

Figure 4-7: Top Level Organization of NON-VON

The PPS is configured as a binary tree of PE's. Each PE comprises a small RAM (64 bytes in the prototype), a modest amount of processing logic, and an Input/Output (I/O) switch. The I/O switch can be set for global bus communication, for communication between parents and children (tree neighbor

communication), or to reconfigure the binary tree as a linear array of processors (linear neighbor communication). All PE's are identical except for minor differences in the "leaf nodes". At the root of the tree is a special processor called the control processor (CP), which is responsible for coordinating different activities within the PPS. The CP is capable of broadcasting instructions to be executed in SIMD mode in all active PE's. The SIMD execution of globally broadcast instructions in NON-VON is compatible with a large number of much smaller and less powerful PE's than has been proposed in other tree machines. This is because NON-VON has no need for local program memories or area-expensive processing and communication hardware. The area occupied by the processor embodied within a single PE is approximately equal to that required for its 64 bytes of local memory. This makes it feasible to effect a "nearly one-to-one" association of logical records with physical PE's. This aspect of NON-VON is central to its processing power in large-scale data processing applications.

The first version of NON-VON, called NON-VON 1, will contain chips with only one PE for the purpose of testing certain electrical and timing characteristics. This chip has already been fabricated and tested. A modified version of the chip with eight PE's is currently being implemented, and may serve as the basis of a later prototype called NON-VON3. The NON-VON3 chip uses the same amount of area per PE, but is considerably faster and has a more powerful instruction set. Currently a new more advanced architecture, called NON-VON 4, is being designed with the goal of significantly expanding the range of applications that can be executed by NON-VON1 and 3 in a highly parallel fashion. The most important change incorporated in NON-VON 4 is that a number of PE's (perhaps 256 to 1K) Large Processing Elements (LPE's), interconnected with a high-bandwidth interconnection network, will be incorporated within the top portion of the PPS tree. Each of these LPE's is capable of serving as a control processor for the subtree of which it is the root. This enables NON-VON4 to function in MIMD and "multiple SIMD" modes. In multiple SIMD mode, each LPE broadcasts instructions from its own local memory to be executed in SIMD mode by its own subtree.

Figure 4-8 shows the design of a single NON-VON 1 PE. A PE actively executes the instructions broadcast by the CP as long as its enable bit is set. If the enable

bit is reset, then the PE is disabled and only an ENABLE instruction will activate it again. Two internal buses constitute the data path. The first bus is eight bits wide, and is used to transfer data between the byte accumulators and other byte registers. The memory and the Memory Address Register (MAR) are both connected to this byte-wide bus. The other bus is one bit wide and is used to transfer data between the one-bit accumulators and the one-bit registers.

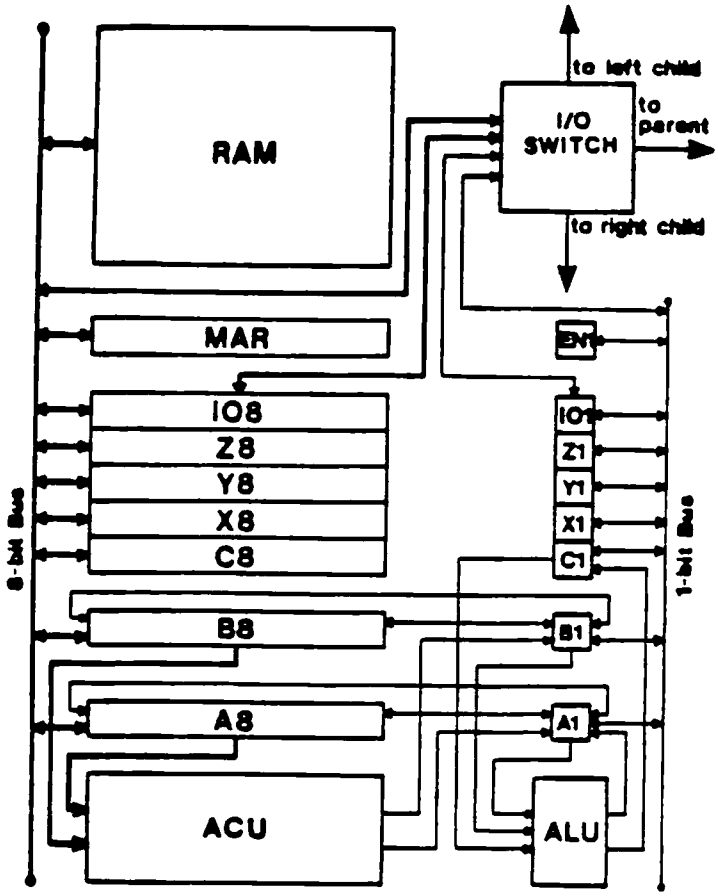

Figure 4-8: NON-VON. Block Diagram of the Processing Element

Figure 4-8 shows also the main functional blocks of the PE data path. They are the one-bit Arithmetic/Logical Unit (ALU), the Arithmetic Comparison Unit (ACU), the byte accumulators A8 and B8, an array of byte registers, the one-bit accumulators A1 and B1, an array of one-bit registers, the MAR, and the memory. The ACU is an eight bit comparator that compares the contents of the two accumulators A8 and B8, setting A1 whenever A8 = B8, and setting B1 whenever

A8 > B8. The ACU is used for content-addressable operations. A8 and B8 can be rotated, left or right, through A1 and B1, respectively. All arithmetic and logical operations other than the 8-bit arithmetic relational operations performed by the ACU are performed bit-serially using the one-bit ALU.

The NON-VON 1 architecture incorporates an SPS based on a number of rotating storage devices. Associated with each disk head in the SPS is a separate sense amplifier and a small amount of logic capable of dynamically examining the data passing beneath it. These Intelligent Head Units (IHU's) are also assumed to be capable of performing general computations (hash coding, for example), and of serving as control processors. This supports parallel transfer of data between the PPS and SPS which is necessary to avoid I/O becoming a bottleneck, and allows NON-VON1 to function as an independent collection of SIMD machines (this execution mode, also employed by other parallel architecture researchers, has come to be referred to as multiple SIMD, or MSIMD).

4.4.1 Communication in NON-VON

The NON-VON I/O switch supports three modes of communication:

1. Global bus communication, supporting both broadcast by the CP to all PE's in the PPS as required for SIMD execution, and data transfers from a single selected PE to the CP. No concurrency is achieved when data is transferred from one PE to another through the CP using the global communication instructions. An instruction called RESOLVE can be used to disable all but a single PE chosen among a specified set of PE's. This is an example of a hardware *multiple match resolution* scheme, in the terminology of the literature of associative processors. (The CP, on executing a RESOLVE instruction, is able to determine whether the operation resulted in any PE being enabled or not). The REPORT instruction transfers data from the single chosen PE to the CP using the global bus communication.

2. Tree communication, supporting data transfers among PE's that are physically adjacent within the PPS tree. Instructions support data transfers to the Parent (P), Left Child (LC), and Right Child (RC) PE's. Full concurrency is achieved in this mode, since all nodes can communicate with their physical tree neighbors in parallel.

3. Linear communication, supporting data transfers to the Left Neighbor

(LN), or Right Neighbor (RN) PE's in a particular logical linear sequence. This mode is useful for applications that require a predefined total ordering of data. Figure 4-9 shows how the linear logical sequence is mapped onto the tree structured physical topology of PPS by inorder enumeration [Knut 73]. The path needed to transfer data between linear neighbors in the tree concurrently are shown in Figure 4-9. Two phases are needed to complete the linear communication cycle. Note that every other element in the inorder sequence is a leaf node. In the first phase, data is transferred along the arrows originating from the leaf PE's, while in the second phase, data passes along the black arrows terminating at the leaf PE's.

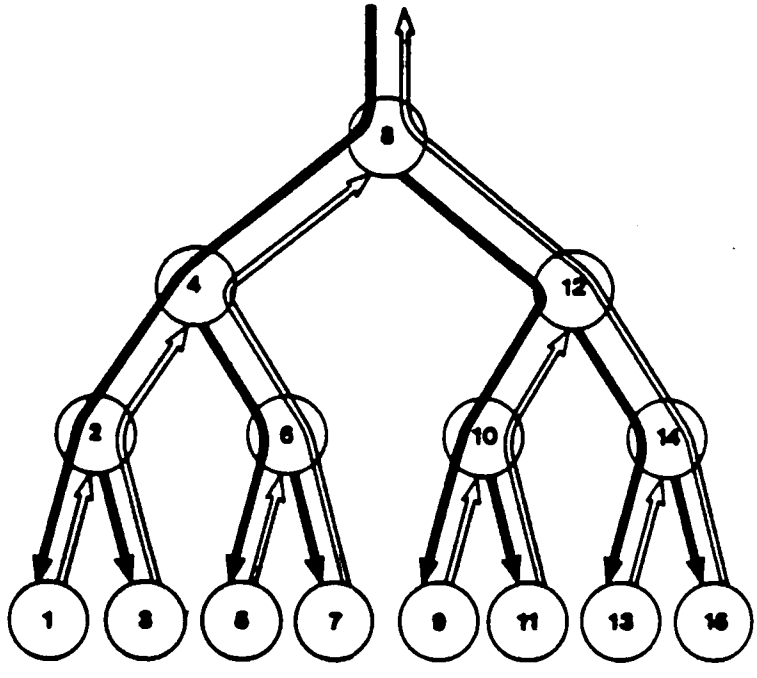

Figure 4-9: Inorder Embedding of the Linear Array

The IO8 and IO1 registers are used for communication with other PE's. There are two types of instructions for tree and linear communications. They are of the form SEND <PE>, and RECEIVE <PE>, where <PE> can be P, LN, RN, LC, or RC. There is no SEND P however, since children must not compete to send messages to a common parent. On executing a SEND instruction, the contents of A8 are transferred through the I/O switch into the IO8 register of the receiving PE. On executing a RECV instruction, the contents of IO8 are transferred into the IO8 register of the receiving PE.

NON-VON is designed to support the massively parallel manipulation of data records stored in its PE's. A data record stored in a single PE is, in effect, capable of manipulating its own contents. Because of this observation, the metaphor of an "intelligent record" is suggested by [Shaw 82]. In real data processing applications however, records may require just a few bytes of RAM, or may be too large to fit in the RAM of a single physical PE. In the first case, more than one record can be packed in a single physical PE, while in the latter case, the record must be split into pieces and stored in several PE's. This mapping between logical records and physical PE's is invisible to the user. Thus a programmer views his records, regardless of their size, as stored one per "virtual" PE. Two schemes are suggested to allocate storage to records that do not fit within a single physical PE. The first scheme, referred to as *linear allocation* method, splits each record among several linearly adjacent (logical) neighbor PE's. The other scheme, referred to as bush allocation, stores each record in a distinct "tree-shaped" cluster of physically adjacent PE's called a bush. For more details on the allocation of logical records in NON-VON, the reader is referred to [Shaw 82] and [Shaw 83].

4.5 The Stony Brook Tree Machine

The Stony Brook tree machine is a tree-structured multicomputer machine that is being implemented at the State University of New York at Stony Brook. The tree machine has the topology of a "double tree structure". The machine consists of two interlocking trees as shown in Figure 4-10. The first tree incorporates user programmable modules (P-modules). The P-modules are programmed in a superior-subordinate mode. The processors at the nodes of the P-tree (P-modules) have their own local memories. They communicate using hand-shake techniques, and the commands and parameters directly communicated between them are of limited length.

The second tree comprises transactional modules (T-modules) that are not accessible to the user. The T-modules are used to provide the necessary communication between P-modules and external storage devices. The T-modules will be used to transmit large data blocks and program segments to the P-modules from the external storage devices. These communication links appear as dotted lines in

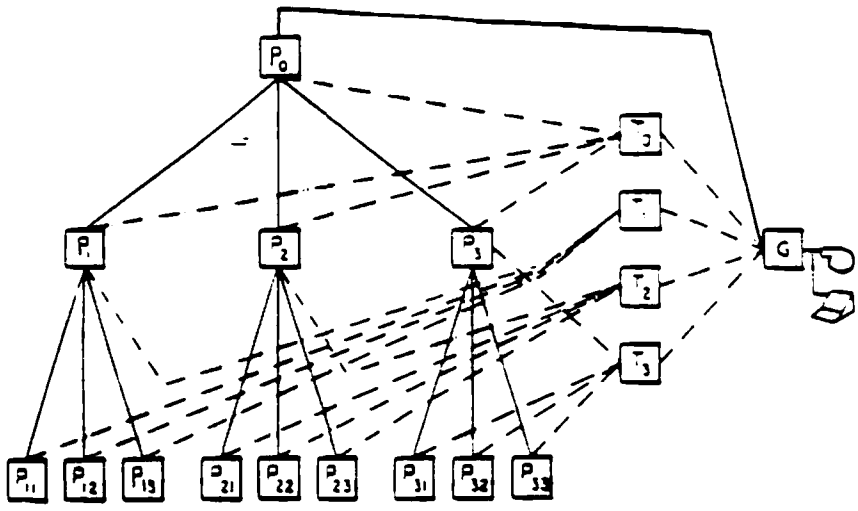

Figure 4-10: Example of a Double Tree Network

Figure 4-10. Data transfers are performed under the control of the T-modules, which are not accessed by the user. Each superior and its immediate subordinates are connected to the same T-module. The T-modules are distributed to permit expandability and to exploit the low cost of small computer modules. In effect the T-modules act as a sort of cache for files that are stored in the central system file storage, labeled G in Figure 4-10.

The role of G is to monitor the whole system for experimental purposes. Programs for the whole system are developed on it. In the following section we will discuss communications in the machine in more detail.

4.5.1 Communication in the Stony Brook Tree Machine

The lines shown in figure 4-10 represent communication pathways of different kinds. The *control links*, shown by solid lines in the figure, permit exchange of short messages between the P-processors under program control. The control links provide also mechanisms by which a superior can assert control over a subordinate. Each control link consists of two independent unidirectional channels, each of which is

capable of buffering one word at a time. Each channel has a single word buffer that can be loaded under program control by the processor on one side. The processor on the other side can read this buffer. A flag for handshake purposes is maintained by each channel. It is set when a buffer is being loaded, and is reset when the buffer is read. An interrupt occurs when the incoming channel buffer to a processor is full. The interrupted processor disables interrupts and reads the message into its local memory by polling the flags. The control link provides two mechanisms that allow a superior to assert control over its subordinates. The first is a *reset* mechanism which is used by a superior to abort the task running in its subordinates, and to prepare for a new assignment. The second facility provided by the control link is a *halt* mechanism, by which a superior can cause the subordinate to enter the halt state. In this state a superior can examine and alter the state of its subordinates. This is useful for interactive debugging, and for bootstrapping/restarting in the P-tree [Bhat 79].

The *data link*, shown using dotted lines in Figure 4-10, can support both exchange of short messages and efficient transfer of large blocks of data. The data link can work in either single-word handshake mode or in a self-sequenced direct memory access mode (DMA mode). For a P-processor to access a file found in the T-processor to which it is connected, it must first engage in protocol in which short messages are exchanged with the T-processor. At this point, DMA mode is enabled and the actual data transfer takes place. The T-processor has no control over the P-processor. The G-T tree was designed to provide services to requests generated by the P-tree processors. For more information about the communication and protection mechanisms in Stony Brook tree machine the reader is referred to [Bhat 79].

In the prototype, currently under construction at Stony Brook, DEC LSI-11's are being used as P- and T-processors, while the G-processor is a DEC PDP-11/60. The mass storage and all peripheral devices have been concentrated on the PDP-11/60 for reasons of economy. The P-tree machine is well suited to applications involving problem-solving by decomposition [Kieb 79]. The P-tree also can be divided into subtrees performing separate computational tasks, using the ability of superiors to initiate and interrupt the execution of subordinate tasks. We will describe some of the algorithms that can run on this tree machine in Section Five.

4.6 Special Purpose Tree Machines

In this section we will describe several special purpose tree machines that implement particular algorithms in a highly efficient manner. These machines are non-von Neumann computing devices that might typically be used as peripherals to conventional computers.

4.6.1 The DADO Tree Machine

DADO [Stol 82] is a parallel tree-structured machine designed for the highly efficient execution of production systems. Production systems have most often been used in artificial intelligence applications to represent a body of knowledge about specific tasks in the real world -- medical diagnosis, for example. A production system consists of a set of rules, or *productions*, which form the *Production Memory* (PM), together with a database of assertions about the real world called the *Working Memory* (WM). A production rule is a statement of the form: if the *condition* holds then this *action* is appropriate. The *condition* is the Left Hand Side (LHS) of the rule, while the *action* is the Right Hand Side (RHS). The WM serves as a "focus of attention" for the production rules. The LHS of each rule represents a condition that must be present in the WM before the action of its RHS is fired. The action can change the WM by adding assertions to it or deleting existing ones.

The production system operation consists of a repeated match-select-act cycle. In each cycle, rules are matched against the current WM assertions. One of the matching rules is selected according to some predefined criteria. The action in the RHS of this rule is then performed. This cycle continues until a goal action is taken or there are no more rules that are matched by the WM.

PE's in DADO are connected in the form of a complete binary tree; their number would be on the order of thousands using today's technology. Each PE contains its own local memory (2K bytes in the current version), a specialized I/O switch allowing global broadcast in addition to tree neighbor communications, and its own processor. DADO is similar in many aspects to NON-VON. The most important differences are the processor granularity and the mode in which each PE can function. DADO has a "coarser granularity" than NON-VON; that is, its

"smallest" processing elements are based on more powerful processors and much larger memories. Each DADO PE can operate in either of two modes. In the first (SIMD mode), it executes instructions broadcast by some ancestor in the tree. In the second, called MIMD mode, the PE executes instructions stored in its local memory. In this mode, the PE is disconnected from its parent and can broadcast instructions to its descendants, providing they are in SIMD mode.

In what follows we will briefly explain how a production system is executed on DADO. In this discussion we assume productions whose LHS and RHS are conjunctions of predicates in which all first order terms are composed of constants and existentially quantified variables [Stol 82].

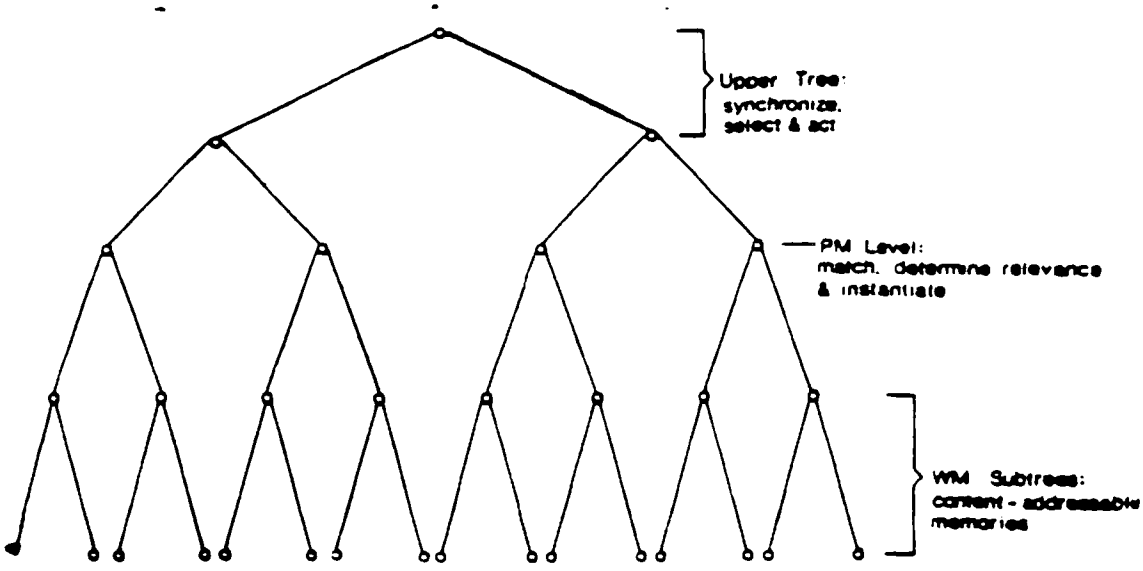

Figure 4-11: Functional Division of the DADO Tree

The tree machine is dynamically divided by the I/O switches into three conceptually distinct components, as shown in figure 4-10:

1. The *PM-level* consists of all PE's at a particular level within the tree that are used to store the productions. Each production is stored in a single PE at the PM level. The PM level in the tree is chosen to be the lowest level having enough PE's to hold all productions.

2. The *upper portion* of the tree comprises all PE's above the PM level. They are used primarily to synchronize the select and act phases of the execution cycle.

3. The *lower portion* of the tree consists of all PE's found below the PM level. Each subtree that is rooted by a PE in the PM level will store that portion of the WM that is relevant to the production stored in this PE. These subtrees are referred to as WM-subtrees.

The execution cycle of the production system consists of three phases:

1. The matching phase: In this phase all PE's at the PM level enter the MIMD mode and simultaneously match the LHS of their productions against the contents of their respective WM subtrees. The matching in each subtree is performed associatively. After the matching phase, only those PE's containing a ground literal that matches the LHS of the rule are left enabled; all other PE's are disabled. Every PE at the PM level, on finishing the matching operation, sets a flag to indicate that it has finished. It also sets a certain word to a value that depends on whether there was a match for the production, and on some criterion that assigns a priority for the PE. The upper part of the tree computes the conjunction of the flags set by the PM-level PE's, and when this conjunction is true, the select phase starts.

2. The selection phase: The-PM level PE's are switched to SIMD mode at the start of this phase. The PE's in DADO's PM level have a unique identifying tag. The upper part of the DADO tree is then used to compute the maximum value of the priorities assigned to PE's on the PM level and to identify a PE having this value. This computation requires $O(\log n)$ steps. The root processor uses the result of this computation to enable that PM-level PE having the maximum priority value. This selected PE then sends the RHS of the rule it contains to the root processor.

3. The action phase: The whole tree is enabled again, and the action in the RHS of the winning PE is executed. This typically involves either adding items to or deleting items from the WM.

This cycle is repeated until a desired state is reached or until no more matchings are found in the WM.

A DADO prototype incorporating 15 PE's is currently operational at Columbia University. A larger prototype, DADO2 which comprises 1023 PE's, is under construction, with each incorporating an Intel 8751 microcomputer chip and an Intel

2168 8K x 8 RAM chip. A high-level language called PPL/M, a variant of Intel's PL/M, will be used to implement DADO's software. The PE will be able to execute the entire Intel 8751 instruction set in MIMD mode, and to broadcast the SIMD instructions to its descendants. The SIMD instruction set is a superset of PL/M. Two additions have been made to PL/M for programming the SIMD mode of operation of DADO. The SLICE attribute defines a variable or a function that is defined to be resident in each PE for which the declaration applies. The second addition is a control construct, called the DO SIMD block, which defines instructions to be broadcast to all SIMD PE's. For further detail of the DADO prototype the reader is referred to [Stol 82b].

4.6.2 A Tree Machine for Searching Problems

This special purpose tree machine was proposed by Kung [Kung 1979a] to efficiently search and maintain a file of fixed-format records. We must be able to insert a new record in the file, delete an existing record, update records, and answer queries about the file. One searching problem that is of interest is called *member search*. In this problem we maintain a set of data elements, and the problem is to determine if a specific element is a member of the set or not. Usually finding the element is followed by other operations in real applications such as reading information associated with this element.

The architecture of Kung's searching machine is shown in figure 4-12. The machine consists of two back-to-back binary trees that share leaf nodes. The machine is based on 16-bit instructions and 32-bit data words. There are three kinds of nodes in the machine:

- *Circle nodes* at the non-leaf nodes of the upper tree, which are used to broadcast a stream of data and/or instructions to the square nodes where the execution of instructions takes place in parallel. The top data paths (links from circle nodes) are 16 bits wide. A circle nodes must update an internal counter in the case of insert and delete operations, and direct the coming data and instruction to one of its two children while sending a no-operation code to the other child in the case of insert operations.

- *Square nodes* at the common leaf nodes, which are used to hold the data and compute results to be combined by triangle nodes. Data records are

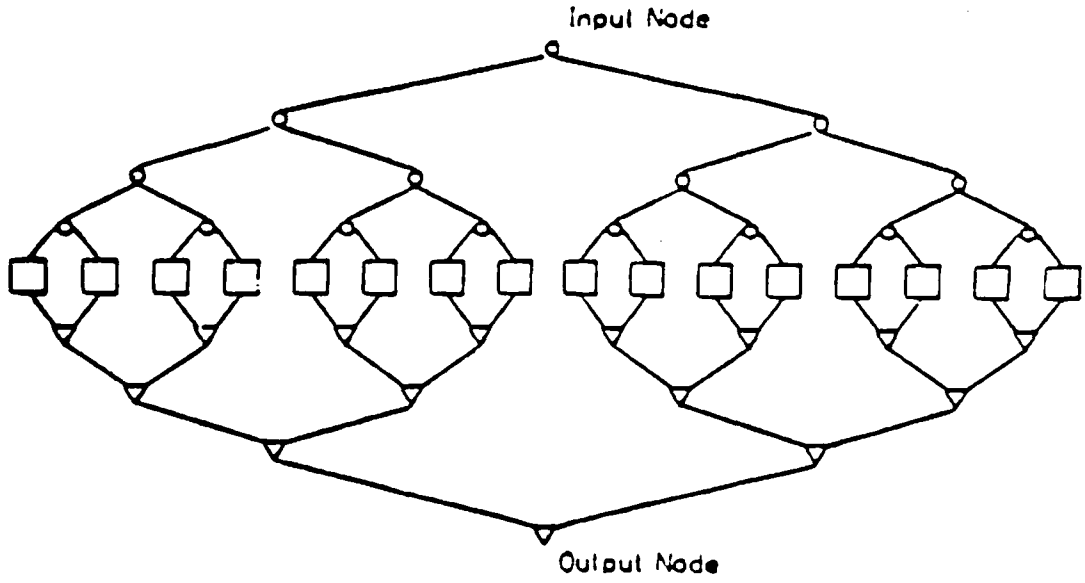

Figure 4-12: Structure of The Search Tree Machine

stored in the tree by placing each record in one of the square nodes. Every square node is a very small von Neumann machine that receives its instructions from an external 16-bit stream and sends 32 bits of result data to its output port. Figure 4-13 shows the architecture of the square node. The processor contains sixteen 32-bit words of memory, two 32-bit general purpose registers (RA,RB), a vector of eight single bit flags (F(0),F(1),..F(7)), a byte register for set identification (SetID), and an instruction register. The processor can be disabled by resetting F(0). The results of comparing RA and RB are stored in two flags(F(1),F(2)). The square node may pass the incoming instruction to the triangle node that is connected to it, if a single bit field in the instruction is reset.

- *Triangle nodes* at the internal nodes of the lower tree, which are used to combine the the results produced by the square nodes and produce answers to queries. The triangle nodes operate on 80 bit packets containing one 16 bits instruction, and two 32 bit data elements coming from the square nodes.

Data flows in one direction only, top to bottom, in a synchronous manner, with operations performed and data transmitted to the next node during each major cycle. Thus the flow of data and computations can be pipelined. There is also a controller at the top of the input node to perform high level functions such as

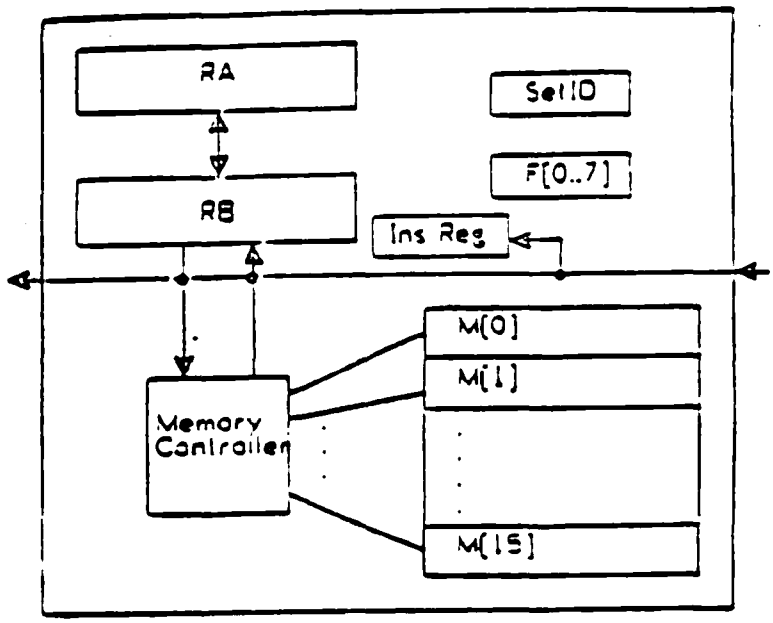

Figure 4-13: Components of the Square Node

loading the tree, creating and deleting sets, defining and handling records, etc. The controller relieves the CPU of this overhead and lessens the traffic on the system bus.

We now describe how the member search problem is solved on this tree. Assuming a set of N elements are stored in the machine, the algorithm starts by inserting the value of the element we are looking for at the input node and broadcasting this value to every square of the tree through the circle nodes. After $\log n$ steps the data element reaches all the square nodes. At that point each square node compares this value with its own data element. If they are equal then the square node sets a flag bit to one; otherwise, it sets the flag to zero. The bits are then combined through the bottom part of the tree (triangle nodes) by letting each triangle node compute the disjunction of its two inputs. After another $\log n$ steps a single bit emerges from the output node telling whether the search was successful or not. The total number of steps to solve the member search problem is thus $2 \log n$.

If the number of data elements that match the value being broadcast is of interest, then the triangle nodes will sum their two inputs instead of just disjunction them. In the same number of steps the machine obtains the number of matching elements

emerging from the output node. If the application calls for the list of records that contain the matching data elements, then the triangle nodes compute the union of their two inputs and send the result of this union in a pipelined fashion to their parents.

In general this tree machine can solve any problem that consists of computing some function over every element in the set, and then combining the values of these results by some associative, commutative binary operator.

Other special machines that have the same architecture were proposed to perform database operations [Song 80], sorting [Song 81], and for the maintenance priority queues [Leis 79].

5. Algorithms on Tree Machines

In this section we will discuss some of the algorithms that run on tree machines presented in this paper. These algorithms represent a subset of parallel algorithms that have attracted researchers' attention since the early sixties [Kung 79b]. A parallel algorithm can be defined as a collection of independent task modules which can be executed in parallel, and which communicate with each other during the execution of the algorithm [Kung 79b].

We will adopt Kung's taxonomy of parallel algorithms [Kung 79b] for the case of tree machine algorithms. In his taxonomy Kung classified parallel algorithms based on their relation to parallel computer architectures. Three important attributes of parallel algorithms were used for the classification. They are *concurrency control, module granularity, and communication geometry*. Concurrency control enforces the desired interaction between task modules to ensure correctness of the algorithm's execution. Synchronous concurrency control is found in algorithms where all task modules execute the same instruction broadcast by a central controller concurrently. Distributed synchronous control corresponds to algorithms where coordination is achieved by simple control mechanisms local to the task modules. Some algorithms use asynchronous control via shared data to achieve concurrency control. Module granularity is the maximal amount of computation performed by a task module before it has to communicate with other modules. Communication geometry refers to the topological layout of the network resulting from the wire connection of task modules to perform the algorithm. The communication structure can be regular or irregular. Trees represent one of the regular communication structures. Other examples are one-dimensional and two-dimensional arrays.

Based on these attributes, we will divide the tree algorithms we will discuss in this sections into three categories:

- SIMD algorithms, where the concurrency control is synchronous and task modules are executing the same instruction simultaneously. The SIMD notion refers to the corresponding computer architecture which executes these algorithms efficiently.

- MIMD algorithms, where the task modules interact asynchronously. These algorithms usually have large module granularity.

- Systolic algorithms, where the concurrency control is distributed and is achieved by local simple control mechanisms.

This section will be divided into three subsections, describing algorithms in the three categories mentioned above. It is worth noting here that the tree machines we discussed before are usually oriented towards efficiently executing algorithms in one of these categories, because of the kind of concurrency control used. For example the Caltech tree is oriented toward performance of MIMD algorithms because its processors interact through messages. The same is true for the X-tree. The NON-VON supercomputer executes SIMD and systolic algorithms efficiently while MIMD algorithms must be adapted (where possible) to its central concurrency control mechanism. The Stony Brook machine and the DADO machine can execute a mixture of MIMD and SIMD algorithms because a processor in the tree can disconnect itself and assume the the role of the central controller for its descendants

5.1 SIMD Algorithms

In SIMD algorithms all PE's in the tree machine execute the same instruction, broadcast by a central controller, concurrently on their own local data. Associative algorithms constitute a subset of SIMD algorithms. Many associative algorithms for solving different kinds of problems can be found in the literature. Examples include [Bent 68], [Walk 62], [Fill 63], [Well 69], [Chen 79], [Film 71], and [Shaw 80]. As an example, this section presents a SIMD algorithm to find the transitive closure of a directed graph [Shaw 83].

5.1.1 The Transitive Closure Algorithm (SIMD Version)

The transitive closure of a directed graph (digraph) G=(V,E), where V is the set of nodes in the graph and E is the set of arcs connecting the nodes, is defined as the directed graph $G^*=(V,E^*)$ where there is an arc from node u to node v in G^* iff there is a directed path from u to v in G. If the number of nodes in G is n, then the digraph G can be presented as an n x n matrix known as the adjacency matrix. If A is the adjacency matrix of the digraph G then $a[i,j]=1$ only if there is an arc going from node i to node j in G. Otherwise $a[i,j]=0$. The number of edges

in G is greater than zero and is less or equal to n^2. The same is true of the number of edges in G^*. There are several algorithms to find the transitive closure. The sequential versions of some of these algorithms are mentioned in [Brow 79]. The widely known Floyd-Warshall algorithm takes $O(n^3)$ steps on a sequential machine to find the transitive closure of a digraph presented to the algorithm in the form of its adjacency matrix. The best time sequential algorithm known at present is $O(n^{\log 7} (\log n)^2)$, where $O(n^{\log 7})$ is the best time complexity known for binary matrix multiplication.

The SIMD algorithm presented in this section is a parallelization of Floyd-Warshall algorithm. Warshall's algorithm is as follows:

```
For k:= 1 to n do
 For i:= 1 to n do
  For j:= 1 to n do
   a[i,j] := a[i,j] or (a[i,k] and a[k,j]);
```

In Warshall's algorithm, newly created arcs affect the creation of new arcs in the closure. The creation of a new arc results from comparing two existing arcs in the closure found so far. The SIMD algorithm for computing the transitive closure of a digraph has a time complexity ($O(n^2)$).

In the SIMD algorithm, each PE in the tree is used to hold a boolean value of the adjacency matrix. Each PE also holds a pair of integers representing the initial and terminal endpoints of the edge held by the PE. Thus n^2 PE's are used in the tree, each containing a boolean value (exists[*,*]). In each of n iterations, for some fixed k, all of the boolean values given by exists[*,k] and exists[k,*] are broadcast into the tree by the central controller in $O(n)$ time. Note that exists[*,*] is initially equal to the value of a[*,*] in the adjacency matrix of G. The boolean values exists[*,k] and exists[k,*] are reported to the central controller before being broadcast throughout the tree. This ensures the use of their most recently updated values. During each of the n steps, PE(i,j) listens for exactly two booleans, namely exists[i,k] and exists[k,j]. If. these two values are true then exists[i,j] is set to true; otherwise it keeps its previous value. The algorithm is formally described as follows [Shaw 83]:

```
For k:=1 to n Do
 Begin
  For i:=1 to n Do
   Begin read exists(i,k);
         broadcast exists(i,k)
   End;

  For j:=1 to n Do
   Begin read exists(k,j);
         broadcast exists(k,j)
   End;
  exists(i,j):= exists(i,j) OR ( exists(i,k) AND exists(k,j) )
 End
```

Note that this parallel version of the Floyd-Warshall algorithm has time complexity $O(n)$ for each of the n iterations, and that it is limited by the I/O time. The interested reader is referred to [Shaw 83] for similar SIMD algorithms to compute the product of two matrices, and all pair shortest path .

5.2 MIMD Algorithms

In this subsection we present an algorithm to find the transitive closure of a digraph on a MIMD machine, for purpose of comparison with the SIMD algorithm for transitive closure. Other MIMD algorithms on tree machines can be found in [Brow 79], [Brow 80], and [Harr 79].

5.2.1 The Transitive Closure Algorithm (MIMD Version)

We will describe the MIMD algorithm for computing the transitive closure of a directed graph as implemented on the CalTech MIMD tree machine [Brow 79]. As mentioned in the section describing the Caltech tree, it is possible to write algorithms that assume arbitrary fanout of the tree node processors. A MAP program will translate these algorithms to the physical binary tree machine. The tree used to find the transitive closure consists of three levels. The first level contains the root processor (*the closureRoot* processor) with a fanout of n where n is the number of nodes in the graph. The second level has n processors (*node processors*), each with a fanout of n. Each node in this level represents a node in the graph. The third level is the leaf nodes level and contain n^2 node processors (*the endNodes processors*). The connection between a *node* and an *endNode* represents a potential arc in the closure. Each endNode processor represents an arc from its parent to the node it represents. The endNode processor will contain a flag that indicates if the arc connecting it to its parent is part of the closure or not

With each arc added to the closure more arcs are broadcast around the tree and new arcs are formed. The algorithm terminates when no new arcs are formed in the tree. Each of the three tree node types executes a different code and they communicate with each other using messages. In the following we will describe the code executed by each node type.

The *closureRoot* processor begins by initializing the tree. Each *node* is assigned a node number. The closureRoot is connected to an external system bus that provides the arcs of the directed graph. The newly formed arcs in the closure are also sent out on this bus. The closureRoot processor is responsible for rebroadcasting newly formed arcs in the closure to all its descendants. The *node* processor, upon receiving the message from the root assigning a node number to it *myV*, starts the arc-processing phase. Upon receiving an arc starting and terminal points (i,j) from the closureRoot processor, the node processor compares the starting and terminal points of this arc to its node number. The node processor creates a new arc by turning on the appropriate descendant if one of two conditions is true. The first condition is that the starting point is equal to the node number. This will take care of the arcs in the initial graph. The second condition is that there is an existing arc from the node to the starting point. In this case, the arc *myV,j* is created. This second condition takes care of Warshall's comparison. The endNode processors are used to store a boolean value indicating whether there is an arc from their parents to them. If the boolean values were to be stored in the node processors, then the algorithm would be dependent on the problem's size. Newly created arcs are broadcast throughout the tree through the closureRoot processor. There are other operations performed by these types of processors to ensure that all arcs are being processed.

The algorithm as described above requires time proportional to the number of arcs in the closure. Thus, the time complexity of the algorithm is $O(n^2)$, limited by the time needed to read out the arcs of the closure. The number of processors used in this algorithm is $2n^2$-1.

5.3 Systolic Algorithms

In this section, we present a systolic algorithm proposed by Leiserson [Leis 79b] to implement a priority queue on a binary tree. Other systolic algorithms on tree machines can be found in [Song 79], [Song 81], and [Kung 79b].

5.3.1 The Systolic Priority Queue Algorithm

A priority queue is a data structure that allow us to insert records into a set and at any time to retrieve from the set the record having the smallest key according to some ordering [Leis 79b]. The systolic tree that implements the priority queue is a binary tree where PE's rhythmically compute and pass data among themselves. The two operations of insertion and extraction can be pipelined. Thus, the systolic tree captures the concept of pipelining in addition to parallelism. Each PE in the binary tree holds two records as shown in figure SYSTREE. A record with key equal to ∞, where ∞ is larger than any possible value for the key, means an empty record. The two records stored in each PE (left record, and right record) are such that the key of the left record is smaller than any key found in the subtree rooted with this PE, and the value of the right record key is greater than any key in the same subtree. Thus, the record with the smallest key is the left record found in the root of the systolic tree.

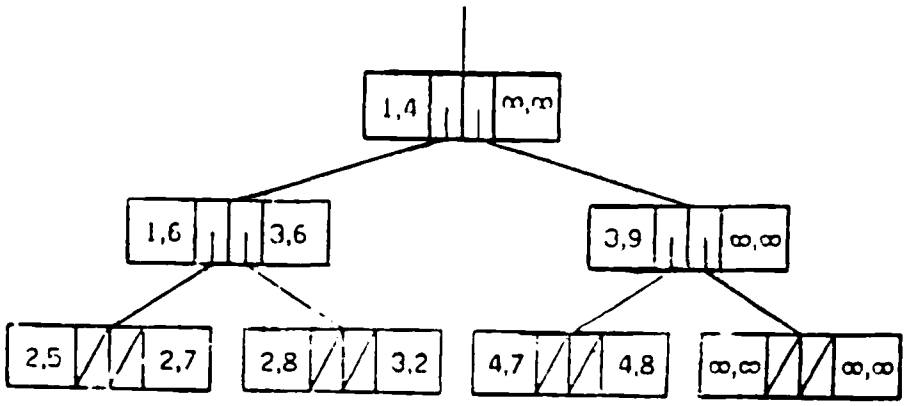

Figure 5-1: The Systolic Tree

In insertion, each PE looks at the incoming key and the values of the keys in its two children to decide what to do. The action taken by the PE can be one of the following:

- If the input key is smaller than the key of the left record, it will send the left record to its left child and replace it by the incoming record.

- If the input key is larger than the left key and is smaller than the right key, we have three subcases: if the incoming key is smaller than the right key of the left child, the input record will be sent to the left child. If the incoming key is larger then the right key of the right child, the right record will be sent to the parent and the incoming record will replace it. Otherwise, the incoming record will be directed to the right child.

- In the case of leaf PE's the incoming key is compared with the two keys, and the largest of the three will be sent to the parent. The other two are placed in such a way that the smaller will be in the left record.

Extracting an element from the queue is analogous to reading the left record of the the root and inserting an empty record.

The time complexity of the insertion algorithm is $O(\log n)$, which is the depth of the tree. Multiple priority queues can be implemented by having every key consists of two parts, the queue number Q and the key of the record a. In this case, a key $<Q,a>$ is treated as less than $<Q',a'>$ if $Q < Q'$, or $Q = Q'$ and $a < a'$. Figure 5-1 illustrates this case.

Acknowledgments

I wish to acknowledge the excellent criticism of my advisor, Professor D. E. Shaw, the valuable suggestions of Professors S. Stolfo, T. Bashkow, G. Maguire, and my fellow student B. Hillyer. I wish also to thank Ella Sanders for proof-reading this paper.

References

[Aho 74] Aho, A. V., Hopcroft, J. E. and Ullman, J. D.
The Design and Analysis of Computer Algorithms.
Addison-Wesley, 1974.

[Back 78] Backus, J.
Can Programming be Liberated from von Neuman Style? A Functional Style and its Algebra of Programs.
Communications of ACM 25, August, 1978.

[Bane 79] Banerjee, J., Hsiao, D. K. and Krishnamurthi.
DBC- A Database Computer for Very Large Databases.
IEEE Transaction on Computers 28(6), June, 1979.

[Bent 68] Bentley, C.
Path Finding with Associative Memory.
IEEE Transactions on Computers 17(7), July, 1968.

[Bent 79a] Bently, J. and Kung, H. T.
A Tree Machine for Searching Problems.
Technical Report, Carnegie Mellon University, August, 1979.

[Bent 79b] Bentley, J. L.
A Parallel Algorithm for Constructing Minimum Spanning Trees.
Technical Report, Computer Science Department, Carnegie Mellon University, August, 1979.

[Berk 71] Berkling, K. J.
A Computing Machine Based on Tree Structures.
IEEE Transactions on Computers 20(4), April, 1971.

[Bhat 79] Bhatt, D. and Smith, D. R.
Communication in a Hierarchical Multicomputer.
In *The Proceedings of IEEE 1st International Conference on Distributed Computing Systems.* 1979.

[Bhat 80] Bhatt, D., et. al.
An Operating System Kernel for a Hierarchical Multicomputer
In *The Proceedings of IEEE Fall Computer Conference* , pages 665:672. 1980.

[Bren 79] Brent, R. P. and Kung, H. T.
On the Area of Binary Tree Layout.
Technical Report, Computer Science Department, Carnegie Mellon University, July, 1979.

[Brow 79] Browning, S.
Computations on a Tree of Processors.
In *The Proceedings of the First Caltech Conference on VLSI*. January, 1979.

[Brow 80] Browning, S.
The Tree Machine: A Highly Concurrent Computing Environment.
PhD thesis, California Institute of Technology, January, 1980.

[Brow 81] Browning, S. and Seitz, C.
Communications in a Tree Machine.
In *The Proceeding of The Second Caltech Conference on VLSI*.
January, 1981.

[Chen 79] Chen, I-N, Chen, P. Y. and Feng, T.
Associative Processing of Network Flow Problems.
IEEE Transactions on Computers 28(3), March, 1979.

[Desp 78] Despain, A. M. and Patterson, D. A.
X-Tree: A Tree Structured Multi-Processor Computer Architecture.
In *Proceedings of the 5th Symposium on Computer Architecture*. April, 1978.

[Esti 63] Estin, G. and Fuller, R. H.
Some Applications for Content Addressable Memories.
In *The Proceedings of the Fall Joint Computer Conference*, pages 495.508. 1963.

[Fahl 80] Fahlman, S. E.
The Hashnet Interconnection Scheme.
Technical Report, Computer Science Department, Carnegie Mellon University, June, 1980.

[Falk 62] Falkoff, A. D.
Algorithms for Parallel Search Memories.
Journal of ACM 9(10), October, 1962.

[Finn 77] Finnila, C. A. and Love, H. Jr.
The Associative Linear Array Processor.
IEEE Transactions on Computers 26(2), February, 1977.

[Fish 80] Fisher, M. J. and Paterson, M. S.
Optimal Tree Layout.
In *The Proceedings of ACM Symposium on Theory of Computing*.
1980.

[Flyn 72] Flynn, M. J.
Some Computer Organizations and Their Effectiveness.
IEEE Transactions on Computers 21(9), September, 1972.

[Fost 76] Foster, C. C.
Content Addressable Parallel Processors.
Van Nostrand Reinhold, 1976.

[Full 67] Fuller, R. H.
Associative Parallel Processing.
In *The Proceedings of the Spring Joint Computer Conference*, pages 471:475. 1967.

[Gilm 71] Gilmore, P. A.
Numerical Solution of PDE by Associative Processing.
In *Proceedings of AFIPS, Fall Joint Conference* , pages 411:418. 1971.

[Good 81] Goodman, J. R. and Sequin, C. H.
Hypertree, a Multiprocessor Interconnection Topology.
IEEE Transactions on Computers 30(12), December, 1981.

[Harr 79] Harris, J. A. and Smith, D. R.
Simulation Experiments of a Tree Organized Machine.
In *The Proceedings of IEEE Parallel Processing.* 1979.

[Hill 81] Hillis, W. D.
The Connection Machine.
Technical Memo, M.I.T. Artificial Intelligence Lab., September, 1981.

[Hoar 78] Hoare, C. A. R.
Communicating Sequential Processes.
Communications of ACM 25, August, 1978.

[Horo 79] Horowitz, E. and Zorat, A.
A Divide and Conquer Computer.
Technical Report, University of South California, July, 1979.

[Horo 81] Horowitz, E. and Zorat, A.
The Binary Tree as an Interconnection Network:Applications to Multiprocessor Systems and VLSI.
IEEE Transactions on Computers 30(4), April, 1981.

[Kieb 79] Kieburtz, R. B.
A Hierarchical Multicomputer for Problem Solving by Decomposition.
In *The Proceedings of IEEE 1st International Conference on Distributed Computing Systems*. 1979.

[Kieb 81] Kieburtz, R. B.
A Distributed Operating System for the Stony Brook Multicomputer
In *The Proceedings of IEEE 2nd International Conference on Distributed Computing Systems*. April, 1981.

[Knut 73] Knuth, D. E.
The Art of Computer Programming.
Addison Wesley, 1973.

[Kuck 77] Kuck, D. J.
A Survey of Parallel Machine Organization and Programming.
Computing Surveys 9(1), March, 1977.

[Kung 79a] Kung, H. T.
The Structure of Parallel Algorithms.
Technical Report, Computer Science Department, Carnegie Mellon University, August, 1979.

[Kung 79b] Kung, H. T.
Let's Design Algorithms for VLSI Systems.
In *The PRoceedings of The First Caltech Conference on VLSI.* January, 1979.

[Lea 77] Lea, R. M.
Associative Processing of Non-Numeric Information.
Reidel Publishing Company, Dordrecht, Holland, 1977,

[Lee 63] Lee, C. Y. and Paull, M. C.
A Content Addressable Distributed Logic Memory with Applications to Information Retrieval.
In *The Proceedings of IEEE*, pages 924-932. June, 63.

[Leis 79a] Leiserson, C. E.
Area Efficient Graph Layout for VLSI.
Technical Report, Computer Science Department, Carnegie Mellon University, August, 1979.

[Leis 79b] Leiserson, C. E.
Systolic Priority Queues.
In *The Proceedings of Caltech First Conference on VLSI.* January, 1979.

[Leis 81] Leiserson, C. E.
Area Efficient VLSI Computation.
PhD thesis, Carnegie Mellon University, October, 1981.

[Lewi 77] Lewin, D.
Introduction to Associative Processors.
Reidel Publishing Company, Dordrecht, Holland, 1977, .

[Lin 77] Lin, C. S.
Sorting with Associative Secondary Storage Devices.
In *The Proceedings of the National Computer Conference.* 1977.

[Lipo 70] Lipovski, G. J.
The Architecture of a Large Associative Processor.
In *The Proceedings of the Spring Joint Computer Conference.* 1970.

[Mead 79a] Mead, C and Conway, L.
Introduction to VLSI Systems.
Addison Wesley, 1979.

[Mead 79b] Mead, C. A. and Rem, M.
Cost and Performance of VLSI Computing Structures.
IEEE Transaction on Solid State Circuits 14(2), April, 1979.

[Patt 79] Patterson, D. A., Fehr, E. S. and Sequin, C. H.
Design Considerations for the VLSI Processor of X-Tree.
In *The proceedings of the 6th Annual Symposium on Computer Architecture.* April, 1979.

[Pott 83] Potter, J. L.
Image Processing on the Massively Parallel Processor.
IEEE Computer Magazine 16(1), January, 1983.

[Rem 79] Rem, M.
Mathematical Aspects of VLSI Design.
In *The Proceedings of Caltech First Conference on VLSI* January, 1979.

[Schw 80] Schwartz, J. T.
Ultracomputers.
ACM Transactions on Programming Languages 2(4"), October, 1980.

[Seit 79] Seitz, C. L.
Self Timed VLSI Systems.
In *The Proceedings of the First Caltech Conference on VLSI.* January, 1979.

[Sequ 78] Sequin, C. H., Despain, A. M. and Patterson, D. A.
Communication in X-Tree, a Modular Multiprocesser System.
In *The Proceedings of The Annual Conference of ACM, Washington D.C.*. December, 1978.

[Sequ 79] Sequin, C. H.
Single Chip Computers, The New VLSI Building Blocks.
In *The Proceedings of the First Caltech Conference on VLSI.* January, 1979.

[Sequ 82] Sequin, C. H, and Fujimoto, R. M.
X-Tree and Y-Components.
Technical Report, University of California at Berkeley, October, 1982.

[Shaw 79] Shaw, D. E.
A Relational Database Machine Architecture.
Technical Report, Computer Science Department, Stanford University, October, 1979.

[Shaw 80] Shaw, D. E.
Knowledge-Based Retrieval on a Relational Database Machine.
PhD thesis, Stanford University, Computer Science Department, August, 1980.

[Shaw 81] Shaw, D E., Stolfo, S. J., Ibrahim, H., Wiederhold, G., Hillyer, B. and Andrews, J. A.
The Non-Von Database Machine. A brief Overview.
IEEE Computer Society Technical Committee, Quartely Report 4(2), December, 1981.

[Shaw 82] Shaw, D. E.
The NON-VON Supercomputer.
Technical Report, Computer Science Department, Columbia University, August, 1982.

[Shaw 83] Shaw, D. E., and Hillyer, B K.
Allocation and Manipulation of Records In NON-VON Supercomputer.
Technical Report, Computer Science Department, Columbia University, January, 1983.

[Shin 82] Shin, K. G., Lee, Y. H., and Sasidher, J.
Design of HM2p - A Hierarchical Multiprocessor for General Purpose Applications.
IEEE Transactions on Computers 31(11), November, 1982.

[Snyd 81] Snyder, L.
Programming Processor Interconnection Structures.
Technical Report, Department of Computer Science, Purdue University, October, 1981.

[Song 79] Song, S. W.
A Highly Concurrent Tree Machine for Database Applications.
Technical Report, Computer Science Department, Carnegie Mellon University, August, 1979.

[Song 81] Song, S. W.
I/O Complexity and Design of Special Hardware for Sorting.
Technical Report, Computer Science Department, Carnegie Mellon University, February, 1981.

[Ster 83] Sternberg, S. R.
Biomedical Image Processing.
IEEE Computer Magazine 16(1), January, 1983.

[Stil 71] Stillman, N. J. , Defiore, C. R. and Berra, P. B.
Associative Processing of Line Drawings.
In *The Proceedings of the Spring Joint Computer Conference.* 1971.

[Stol 82a] Stolfo, S. J. , Shaw, D. E.
DADO: A Tree-structured Machine Architecture for Production Systems.
In *The Proceedings of the 2nd National Conference on Artificial Intelligence.* August, 1982.

[Stol 82b] Stolfo, S. J., Miranker, D , and Shaw, D. E.
*Programming The DADO Machine: An Introduction to PPL/M**
Technical Report, Columbia University, November, 1982.

[Ston 71] Stone, H.
Parallel Processing with the Perfect Shuffle.
IEEE Transactions on Computers 20(2), February, 1971

[Suth 77] Sutherland, I. E., and Mead, C. A.
Microelectronics and Computer Science.
Scientific American (237(3)):210-228, September, 1977.

[Thom 80] Thompson, C. D.
A Complexity Theory for VLSI .
PhD thesis, Carnegie Mellon University, August, 1980.

[Thur 75] Thurber, K. J. and Wald, L. D.
Associative and Parallel Processors.
Computing Surveys 7(4), December, 1975.

[Toll 81a] Tolle, D. M. and Siddall, W. E.
On the Complexity of Vector Computation in Binary Tree Machines.
Information Processing Letters 13(3), December, 1981.

[Toll 81b] Tolle, D. M.
Coordination of Computation in a Binary Tree of Processors.
PhD thesis, University of North Carolina at Chapel Hill, 1981.

[Toll 81c] Tolle, D. M.
Implanting FFP Trees in Binary Trees: An Architectural Proposal.
In *The Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 115:122. October, 1981.

[Wesl 69] Wesley, M. A.
Associative Parallel Processing for the FFT.
IEEE Transactions on Audio and Electro Acoustics 17(2), June, 1969.

[Yau 77] Yau, S. S. and Fung, H. S.
Associative Processor Architecture.
Computing Surveys 9(1), March, 1977.