Computations on a Tree of Processors

Sally A. Browning
Computer Science Department
California Institute of Technology
Pasadena, California 91125

Because processors and memories are both implemented in silicon, it is worthwhile to consider architectures that mingle both functions on a single chip. With the VLSI promise of a million or so devices on a chip, several hundred processors can communicate with each other at on-chip speeds.

But in order to manage the complexity of such a chip, both when designing it and when testing it, the interprocessor communication paths should be regular and simple. This paper examines the utility of a particular interconnect scheme, a binary tree.

The processors are arranged as a binary tree: each processor except the root has a single parent, and each processor except those at the leaves of the tree has two descendents. This arrangement models the hierarchical communication found in large organizations.

The binary tree architecture has some interesting aspects that make it a good choice for a general purpose structure. Any particular processor in a tree of n processors can be accessed in at most $\log_2 n$ time. This compares favorably with the $O(n)$ access time for a linear arrangement of processors, or the $O(\sqrt{n})$ access time if the processors are arranged in a rectangular array.

The number of processors available increases exponentially at each level in the tree machine. If the problem to be solved has this growth pattern, then the tree geometry will match the problem. By

---

The research described in this paper was sponsored by the Defense Advanced Research Projects Agency under contract number N00123-78-C-0806.

contrast, processors arranged as a list have a constant number of processors (namely 1) available at each level. And rectangular arrays make a polynomial number of processors available at each level, according to the allocation scheme chosen. Figure 1 demonstrates this property of the three structures.

These three schemes are the simplest ways to connect processors together. They provide each processor with two (the list), three (the tree), or four (the square array) neighbors. Each has advantages over the others, and each has a fan club.

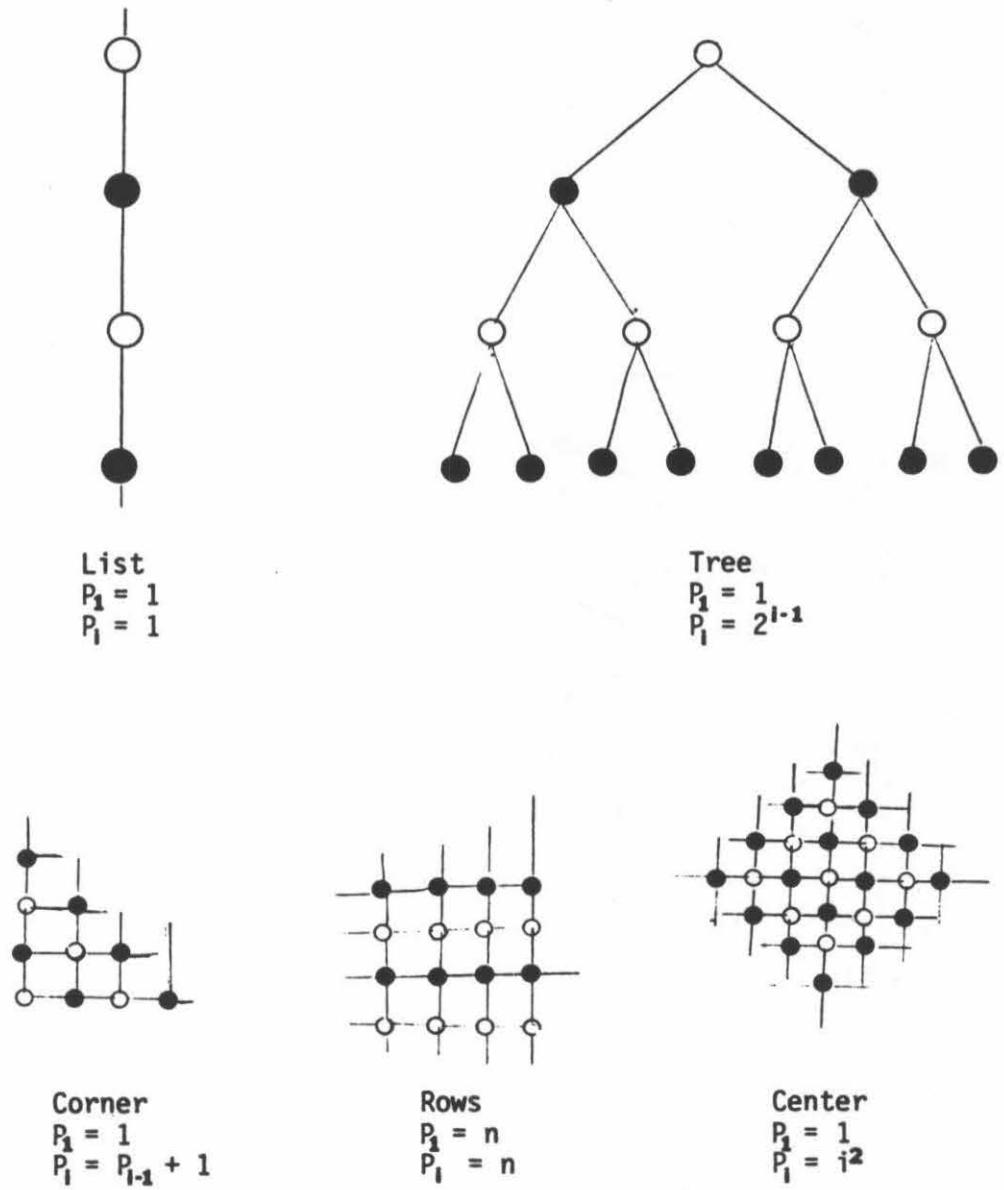
The point of my research is to determine whether or not there is a predominate geometry to the problems that might be solved on a highly concurrent machine. If such a geometry exists, and a hardware implementation is realizable in silicon, then that machine should be built.

Since the tree architecture appears to have more flexibility than the other two structures, I have concerned myself mostly with it. This paper will describe several algorithms that have been successfully mapped onto the tree. In later sections, the Ringmachine, a linear array of processors proposed by Mike Ullner[7], will be introduced in order to show that problems that are dominated by loading and unloading do not require the additional communication paths available in the tree. The final section of the paper describes a problem from numerical analysis that makes effective use of the tree machine. The paper ends with some comments about the direction future investigations will take.

A Digression into Programming Notation.

The processors in the tree have some characteristics that must be emphasized by the notation used to describe them.

First, each processor is a general computing machine with some amount of local store. A template that describes both the program



P_i = number of processors at level i.

Figure 1. Processors Available at each Level.

and the data that will characterize each processor. This template will be instantiated as many nodes of the tree.

Communication between each processor, its parent, and its children should be limited to explicitly defined entry points. That is, there is no omnipotent entity that is able to oversee and influence the actions of other processors except as explicitly described. Each processor can expect to have local sovereignty, and can only be affected by communication it expects.

And perhaps most importantly, locality should be encouraged in the problem solutions. Communication between processors requires synchronizing their actions, limiting the amount of concurrency that can be achieved.

The notation that embodies these criteria is the class construct described by Dahl and Hoare [4]. The class allows the programmer to define as a single entity both a data structure and the procedures that operate on it. Thus the implementation details are known only to the class itself. Each object is an instance of a class, and can be thought of as a machine, capable of local computation but responding to well-defined requests from the outside world.

The most widely known programming language that incorporates the class construct is SIMULA 67 [2]. SIMULA extends the syntax of Algol 60 with class definitions. I will use a modified version of the SIMULA syntax to describe the nodes in the processor tree. The syntax for a class declaration can be described in BNF as follows:

```
<class declaration> ::= class <class identifier>
                            <formal parameter part>;
                            <attribute part>;
                            <class body>

<class body> ::= <statement>
```

In order to describe highly concurrent algorithms despite the sequential nature of SIMULA, the meaning of the semicolon symbol is changed. In vanilla SIMULA, semicolon is used to terminate a statement. Instead, read semicolon as "At this point, all statements in progress must be terminated before advancing to the next statement". Linefeed will be used to indicate syntactic end of the statement. In other words, linefeeds are used to separate statements; semicolons are used to separate groups of statements that can execute concurrently. E. W. Dijkstra introduces this semicolon convention in [6].

Making Arbitrary Branching Ratios.

While the physical structure of the tree restricts each processor to two descendents, a logical structure can be imposed on the tree to accomodate an arbitrary branching ratio. Each logical processor consists of several physical processors, enough to provide the desired number of offspring. A logical node with n children is built from n-1 physical nodes and is $\log_2 n$ levels deep. Figure 2 shows some sample logical processors. Figure 3 gives a SIMULA representation of the algorithm used to simulate arbitrary branching ratios.

All of the algorithms described in this paper will describe logical processors and the logical structure of the tree. The SIMULA code assumes the existence of the logical processor defined in Figure 3, and builds definitions based on it. The complexity of each algorithm will be calculated for the physical structure, however.

Sorting.

A binary tree with depth $\log_2 n$ can be used to sort n numbers. The sort is accomplished as a byproduct of loading the numbers into memory and then reading them out again. The numbers themselves are

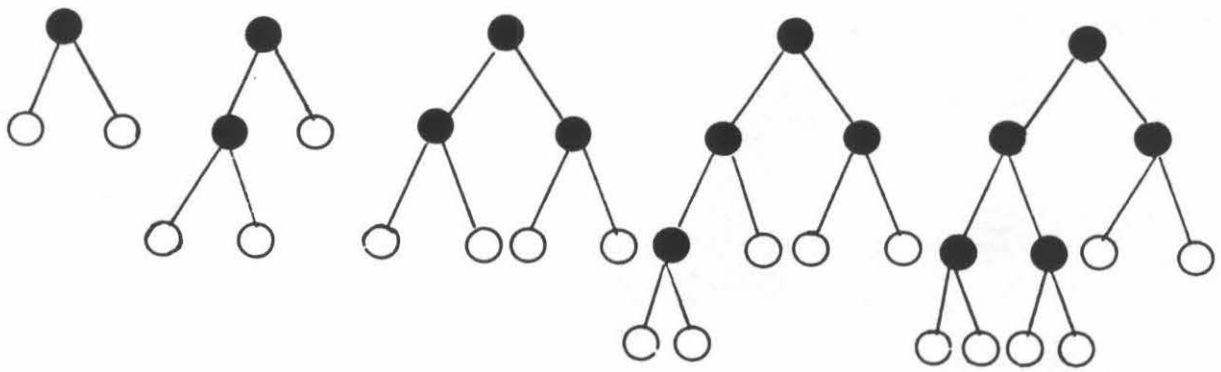


Figure 2. Logical Processors with 2 to 6 Descendents (solid color boxes comprise the logical processor)

```
CLASS Node(n); INTEGER n;
BEGIN
   REF(Node)left,right;

   !init code to build logical node;
   IF n>2 THEN left:-NEW Node((n+1)//2);
   IF n>3 THEN right:-NEW Node(n//2);

END of CLASS Node;


Node CLASS Processor;
BEGIN

   REF(Processor) PROCEDURE Son(s); INTEGER s;
   BEGIN REF(node)p;
      p:-IF s<=(n+1)//2 THEN left ELSE right;
      WHILE NOT (p IN Processor) DO
         p:-IF s<=(p.n+1)//2 THEN p.left ELSE p.right;
      Son:-p;
   END of PROCEDURE Son;

END of CLASS Processor;
```

Figure 3. Making Arbitrary Branching Ratios

never in sorted order internally, but come out of the tree in the desired order.

Sorting is a particularily interesting example because it illustrates a fundamental issue in concurrency. It is well known that sorting on a sequential machine can be done with $O(n\log_2 n)$ comparisons. However, it has been shown on very fundamental grounds that if communication is restricted to nearest neighbors, at least n^2 comparisons are required[5]. The apparent advantage of the $O(n\log_2 n)$ algorithms comes as a direct result of longer communication paths. It is also clear that no scheme will be able to produce an ordered set of numbers until all numbers are loaded into the machine. This means that the best achievable time complexity is $O(n)$.

The algorithm I use is an implementation of heap sorting. The algorithm that runs in each processor, given in Figure 4, has a procedure for loading the tree called **Fillup** and a procedure invoked during the output cycle called **Passup**.

Fillup keeps the largest number seen to date, and passes the smaller one to the right or left child, keeping the tree balanced by alternating sides.

Passup returns this processor's current number and refills itself with the larger of the numbers stored in its descendents. This action is pipelined so that the largest number is always available in the root.

This sort algorithm is bounded by the time it takes to load and remove the numbers. Thus it has time complexity $O(n)$. It requires n processors, one for each number to be sorted.

```
Processor CLASS HeapSort;
BEGIN

   INTEGER number;
   BOOLEAN balanced,empty;
   REF(processor)left,right;

   PROCEDURE fillup(candidate); INTEGER candidate;
   BEGIN
      IF empty THEN
      BEGIN
         number:=candidate
         empty:=FALSE;
      END
      ELSE
      BEGIN
         IF candidate>number THEN !swap;
         BEGIN INTEGER t;
            t:=candidate;
            candidate:=number;
            number:=t;
         END;
         IF balanced
         THEN left.fillup(candidate)
         ELSE right.fillup(candidate);
         balanced:=NOT balanced;
      END;
   END of procedure fillup;

   INTEGER PROCEDURE passup;
   BEGIN
      passup:=number;
      IF left==NONE AND right==NONE THEN empty:=TRUE !its a leaf;
      ELSE
      IF left.empty THEN
      BEGIN
         IF right.empty THEN empty:=TRUE  !both subtrees empty;
         ELSE number:=right.passup; !fill from right son;
      END
      ELSE
      IF right.empty THEN number:=left.passup !fill from left son;
      ELSE number:=IF left.number>right.number
         THEN left.passup ELSE right.passup;
         !take the larger of the two;
   END of procedure passupnumber;


   !init code;
   empty:=TRUE;
   balanced:=TRUE;
   !left and right set;

END of class HeapSort;
```

Figure 4. Heap Sort

Matrix Multiplication.

Consider the problem of multiplying two nxn matrices together. The tree machine algorithm that provides the answer in the least amount of time divides the multiplicand into rows and the multiplier into columns, pipelines the loading and multiplication of pairs of single elements. This process requires $O(n^2)$ processors and takes $O(n^2)$ time, a processor and time product of $O(n^4)$. If each processor has enough memory to store a row of the matrix instead of a single element, the algorithm would require $O(n)$ processors, resulting in the more familiar $O(n^3)$ product.

The algorithm makes use of a tree that has a branching ratio of n at each node, and is two levels deep. The root node has n descendents, each controlling n leaves of the tree. Then there are n^2 leaves and a total of $2n^2-1$ processors.

Each child node of the root, hereafter called a **row supervisor**, will represent a row of the multiplicand matrix, and produce a row of the product matrix. Each of the n descendents of a row supervisor will hold one element of the row.

The algorithm is given in Figure 5. The multiplier matrix is loaded into the tree one element at a time, by column. The root hands each element to all row supervisors, which send it to their appropriate leaf: the first element in any column goes to the first child of each row supervisor, the nth element to the nth child. That child multiplies the multiplier element by the multiplicand element it holds, and returns the product to the row supervisor. When an entire column of the multiplier has been loaded into the tree, each row supervisor takes the n products generated in its children, adds them, and returns one element in the corresponding column of the product matrix. That is, when the first column of the multiplier has been loaded into the tree, the first column of the product matrix is available, and so on.

This process can be pipelined to take $O(n^2)$ time. Thus the time it

```
Processor CLASS Rowsupervisor;
BEGIN
!the matrix size, N, is an attribute of CLASS Processor, and is available
to us;

   REAL product;
   INTEGER count;

   PROCEDURE Load(element); REAL element;
   BEGIN
      count:=count+1;
      son[count].load(element);
      IF count=N THEN count:=0;
   END of procedure Load;

   REAL PROCEDURE Multiply(element); REAL element;
   BEGIN
      count:=count + 1;
      product:=product + son[count].multiply(element);
      IF count=N THEN
      BEGIN
         multiply:=product;
         count:=0;
         product:=0.0;
      END;
   END of procedure Multiply;

   !initialization;
   count:=0;
   product:=0.0;
END of class Rowsupervisor;


Processor CLASS Leaf;
BEGIN

   REAL rowelement;

   PROCEDURE Load(element); REAL element;
   BEGIN
      rowelement:=element;
   END of procedure Load;

   PROCEDURE Multiply(element); REAL element;
   BEGIN
      multiply:=rowelement * element;
   END of procedure Multiply;

END  of class Leaf;
```

Figure 5. Matrix Multiplication

takes to load the n^2 elements of the matrices dominates the time complexity of the problem. Remember, however, that matrix operations are meaningless except in the context of the driving problem. The entries in the matrix are not so much loaded as generated, and the generation time may be less than $O(n^2)$. Care must be taken, however, to generate the matrix entries in the arrangement used by the multiplication algorithm; moving elements around in the tree is costly.

The Color Cost Problem.

This NP-complete problem is an adaptation of the K-colorability problem. Given an undirected graph G of n nodes and a set of n colors, each with an associated cost, find a minimum cost coloring of the graph such that no nodes sharing an edge are the same color.

There are n^n possible colorings of the graph. Evaluating them sequentially produces a solution in time $O(n^n)$. I present a parallel algorithm of order n^2.

Each level in the processor tree represents the consideration of another node. That is, level one shows possible colors for the first node, level two colors the second node based on the choices made for at level one, and so on. I will describe the generation of the potential colorings.

Each processor, described in Figure 6, has an edge list called edge and a list of costs indexed by color number called colorcosts. There is an array called coloring that reflects the color choices for preceding nodes, and a boolean array called colors that is used to generate the possible colorings for this node.

The algorithm, given in procedure color, begins by assuming that all colors yield valid colorings. The array coloring is used to eliminate those colors that have been used to color nodes that share an edge with this node. This reduced set of colors, all of

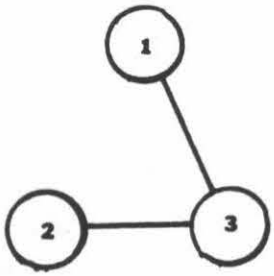
```
Processor CLASS ColorCost;
BEGIN

   BOOLEAN ARRAY edge[1:n,1:n],colors[1:n];
   INTEGER ARRAY coloring[1:n],colorcosts[1:n];
   INTEGER cost;

   PROCEDURE color(node); INTEGER node;
   BEGIN INTEGER i;
      IF node>n THEN
      BEGIN
         cost:=0;
         FOR i:=1 TO node-1 DO cost:=cost+colorcost[coloring[1]];
      END
      ELSE
      BEGIN
         FOR i:=1 TO node-1 DO IF edge[1,node] THEN
         colors[coloring[i]]:=FALSE;
         FOR i:=1 TO n DO
            IF colors[i] THEN
            BEGIN
               son(i).coloring[node]:=i
               son(i).color(node+1);
            END
            ELSE son(i):-NONE;
         cost:=maxcost;
         FOR i:=1 TO n DO
            IF (IF son(i) = NONE
               THEN FALSE ELSE cost>son(i).cost)
            THEN cost:=son(i).cost;
      END;
   END of procedure color;

END of class ColorCost;
```

Figure 6. Color-Cost Problem



Symbol	Color	Cost
B	Blue	2
G	Green	1
R	Red	0

Figure 7. Color-Cost Example: Graph and Color List

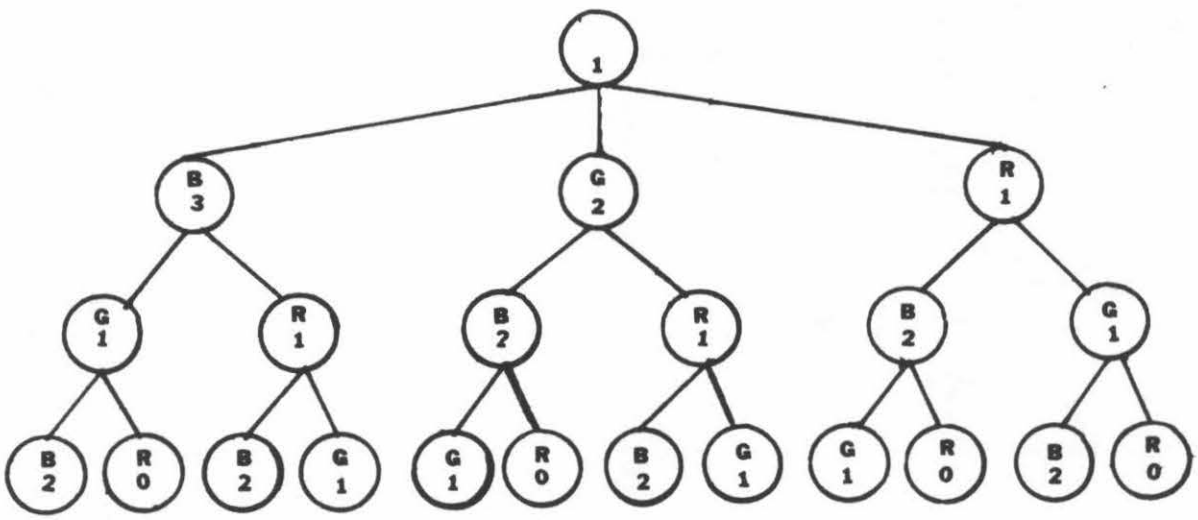


Figure 8. Color-Cost Example: Solution Tree

which are legal colorings, is used to spawn descendents, one for each coloring of this node.

When the tree is n levels deep all the legal colorings have been generated. The leaf nodes calculate a cost for the coloring they represent, and each parent node takes as its cost the least cost among its children. Thus the minimum cost coloring is stored at the root.

An example is in order. A sample graph and color set are given in Figure 7. Figure 8 shows the colorings and costs arrived at by the algorithm. Each level of the tree represents a node of the tree. That is, if the root is level 0, the first node is colored in level 1, and level 3 represents potential colorings for the third node. Besides representing a part of a coloring, each node also contains the minimum cost coloring found among its descendent colorings.

The minimum cost of coloring the sample graph is 1, and is achieved by coloring nodes (1,2,3) (red,green,red).

When the color cost problem is solved in a brute force manner on a sequential machine, it takes exponential time. The tree machine can solve the problem in $O(n^2)$ time using an exponential number of processors. So on either machine, this problem exhibits exponential growth.

Transitive Closure.

Given a directed graph G, the transitive closure of G, G*, can be generated. The arcs of G* are subject to the following condition: for every arc (v,w) in G* there is a path, $(v,e_1),(e_1,e_2)$, ... (e_m,w), in G.

The best sequential algorithm for generating the transitive closure of a graph is attributed to Warshall[1,8]. The algorithm uses three FOR loops that run through the incidence matrix adding arcs. After

k steps of the outer loop, there is a path from vertex i to vertex j through vertices in the set {1,2,...,k} if and only if B[i,j]=1. On a sequential machine, this algorithm takes $O(n^3)$ time. The code is given in Figure 9.

A direct mapping of Warshall's algorithm onto the tree machine yields a rather boring n^3 algorithm that merely spreads the three iterative steps among the processors in the tree.

There is a much more fruitful path to take. By understanding what actually happens during the execution of the algorithm, an effective mapping of Warshall's algorithm onto the tree machine is discovered.

There are two key points to be made about Warshall's algorithm. First, the algorithm is cascading. Newly created arcs can effect the creation of yet more arcs. Any realization of the algorithm must allow for this characteristic. It is not sufficient to consider only the arcs in the original graph.

Also important is the comparison between arcs. In Figure 9 this comparison is stated as

```
IF  b[i,j] AND b[j,k] THEN b[i,k]:=TRUE;
```

In English, this reads "if there is an arc from i to j, and an arc from j to k, then create an arc from i to k".

Suppose that instead of an incidence matrix, there is a list of arcs. This list will be used as input to the tree machine. The output is the list of arcs in the transitive closure.

The tree has a root node, n descendents of the root that are instances of the class vertex, and n^2 descendents of the vertex processors described by the class toVertex. The vertex processors represent the n nodes in the graph. The toVertex processors are the n possible arcs from each node. Jim Rowson deserves special thanks

for distilling my complicated structure into this very simple one.

The arcs in the original graph are used only as the starting place and are indistinguishable from generated arcs. As new arcs are created, they are considered by all the vertex processors just as the original arcs are.

Arcs are created using a variant of the Warshall comparison. An arc has a starting point, fromV, and an ending point, toV. Each arc is considered by all the vertex processors. Each vertex will create an arc by turning on its appropriate descendent if one of two conditions is true. Either this vertex must be the starting point of the arc, or there must be an existing arc from this vertex to the starting point.

The first condition takes care of the arcs in the original graph. The tree starts out with no arcs. As the original arcs are loaded into the tree, the first condition is true and arcs are created.

The second condition is the Warshall comparison. Suppose the arc (v,w) is being considered by vertex u. If arc (u,v) exists then arc (u,w) is created. This is how new arcs are created.

As each arc is created, by satisfying either criterion, it is broadcast throughout the tree; it might effect the creation of other arcs.

The code for this algorithm is given in Figures 10 and 11. Figure 10 shows the properties common to all three kinds of processor nodes, and defines some auxiliary classes used for queueing and passing data between processors. Figure 11 is the definition of the three processors, including the procedures that implement the revised Warshall algorithm.

The key routines are load and unload. Procedure load appears in the root and vertex processors and is used to pass arcs through the system. Unload is in the root. Each call on unload yields an arc in

```
BOOLEAN ARRAY B[1:n,1:n];
        INTEGER i,j,k;

        FOR k:=1 to n DO
            FOR i:=1 to n DO
                FOR j:=1 to n DO
                    IF B[i,k] AND B[k,j] THEN B[i,j]:=TRUE;
```

Figure 9. Warshall's Algorithm (Sequential Machine)

```
CLASS processor;
BEGIN

   REF(processor) array son[1:n];
   REF(processor) parent;

END of class processor;

processor CLASS Qprocessor;
BEGIN

   REF(head)Q;

   PROCEDURE insertinQ(qe); REF(queueElement)qe; qe.into(Q);

   REF(queueElement) PROCEDURE firstinQ;
   BEGIN REF(queueElement)qe;
      firstinQ:-qe:-Q.first;
      qe.out;
   END of procedure firstinQ;

   Q:-NEW head;

END of class Qprocessor;


link CLASS queueElement(myOwner); REF(Qprocessor)myOwner;
BEGIN
END of class queueElement;


CLASS edge(fromV,toV); INTEGER fromV,toV;
BEGIN
END of class edge;
```

Figure 10. General Processor Definition and Auxilary Classes

```
Qprocessor CLASS root;
BEGIN
   PROCEDURE load(e); REF(edge)e;
   BEGIN INTEGER i;
      FOR i:=1 STEP 1 UNTIL n DO  son[i].load(e);
   END of procedure load;

   REF(edge) PROCEDURE unload;
   BEGIN
      IF Q.empty THEN unload:-NONE
      ELSE BEGIN REF(queueElement)qe; REF(edge)e;
         qe:-firstinQ;
         unload:-e:-qe.myOwner.nextEdge;
         load(e);
      END;
   END of procedure unload;

   BEGIN integer i;
      FOR i:=1 STEP 1 UNTIL n DO son[i]:-new vertex(i);
   END of init code;
END of class root;


Qprocessor CLASS vertex(myNode); INTEGER myNode;
BEGIN
   REF(queueElement)qe;
   BOOLEAN queued;

   REF(edge) PROCEDURE nextEdge;
   BEGIN REF(queueElement)qt;
      qt:-firstinQ;
      nextEdge:-NEW edge(myNode,qt.myOwner.myNode;
      IF NOT Q.empty THEN parent.insertinQ(qe)
      ELSE queued:=FALSE;
   END of procedure nextEdge;

   PROCEDURE load(e); REF(edge)e;
   BEGIN
      IF e.fromV=myNode OR son[e.fromV].edgeexits
      THEN BEGIN
         son[e.toV].markedge;
         IF NOT queued
         THEN BEGIN
            parent.insertinQ(qe);
            queued:=TRUE;
         END;
      END;
   END of procedure load;

  queued:=false;
   qe:-NEW queueElement(THIS vertex);
  BEGIN INTEGER i;
     FOR i:=1 STEP 1 UNTIL n DO son[i]:-new toVertex(i);
  END of init code;
END of class vertex;
```

Figure 11. Revised Warshall Implementation ***(Continued on next page)***

```
Qprocessor CLASS toVertex(myNode); INTEGER myNode;
BEGIN
   REF(queueElement)qe;
   BOOLEAN edgeExists;

   PROCEDURE markEdge;
   BEGIN
      IF NOT edgeExists
      THEN BEGIN
         edgeExists:=TRUE;
         parent.insertInQ(qe);
      END;
   END of procedure markEdge;

   edgeExists:=FALSE;
   qe:-NEW queueElement(THIS toVertex);
END of class edge;
```

Figure 11. Revised Warshall Algorithm Implementation

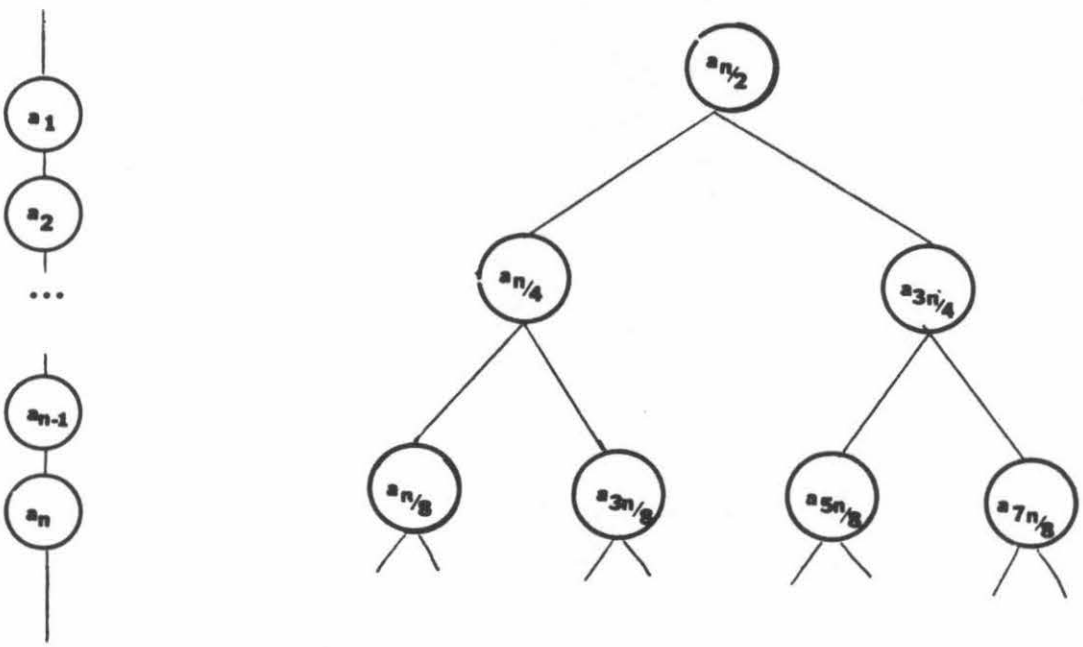


Figure 12. Arrangements of a in the Tree and Ringmachine

the transitive closure.

Each arc in the original graph is given to the root via a call on procedure load. The arc is passed to all vertex processors. There, on the second level, each vertex executes the test described above to see if the arc causes the creation of an arc from this vertex.

Once all the arcs of the original graph have been loaded, the arcs of the transitive closure are available for unloading. As an arc is handed to the outside world by a call on the root's unload procedure, it is passed back down the tree to the vertex processors, just as the original arcs were, by a call on procedure load.

A double system of queues is used to indicate the availability of arcs for the unloading and broadcasting operations. The queue in the root is used by the vertex processors to indicate willingness to provide an arc to the root. When an arc is unloaded, it is also broadcast through the tree via the load routine. The queue in the vertex processors is used by the toVertex processors to indicate that another arc has been created.

The queues are used to avoid polling the vertexs and toVertexs from available arcs. The polling introduces two iteration statements which are executed for each arc in the transitive closure. They cloud the issue by appearing to affect the complexity. The queues, on the other hand, simulate the hardware nicely. The two upper levels of the tree need to respond to a signal from any one of their children. The queues provide this effect.

The algorithm as described above and in Figure 11 has time complexity of the order of the number of arcs in the transitive closure. The maximum number of arcs in a directed graph of size n, is n^2; the transitive closure is itself such a graph, is n^2. Thus the time complexity of this algorithm is $O(n^2)$, limited by the time it takes to read out the arcs of the closure.

As described, $2n^2-1$ processors are used to generate the closure. A solution using only n+1 processors, yet essentially the same, can be devised. Suppose each vertex processor, now the leaves of the tree, contains a boolean array instead of using toVertex processors to represent existing arcs. The vertex processors have more local store, and a parameter of the problem, the size of the graph, has been introduced into the physical requirements for each processor. This is something I want to avoid. It is, however, a perfectly valid implementation, and indeed, retains the $O(n^3)$ total complexity.

Is the Tree Machine Magic?

It is time to address the question of whether these problems need the tree machine structure. The answer is simple. No. I will give an alternative architecture that yields an equivalent solution.

Mike Ullner has proposed the Ringmachine [7], a "tree of branching ratio one". The structure is a doubly-linked ring of processors, or more simply, a linear pipeline.

This structure is also capable of doing transitive closure in $O(n^2)$ time using $O(n^2)$ processors, and the code is as simple as that for the tree machine implementation. The Ringmachine algorithm is given in detail in [3].

The key to the $O(n^2)$ solution is the pipeline, not the communication path. In fact, sorting and matrix multiplication are also problems in this class. The size of the answer determines the size of the problem. Any pipelined structure that can spew out answers one at a time in a continuous stream is adequate.

So what is the tree machine better at? The difference between the tree and the ring is that any particular node in the tree can be total number of processors. Problems that have one answer that can be in any of a large number of processors can take advantage of the

tree structure. NP-complete problems, like the color cost problem treated earlier, are a graphic example of this. Those problems require an exponential number of processors, however, and thus are not practical.

An Algorithm that Uses the Tree Effectively.

I have found a problem that does make use of the extra communication paths in the tree. It is taken from numerical analysis, and is presented here out of context. The problem is to generate a vector x from a vector a according to the following rule:

$$x_i = \sum_{j=1}^{i} a_j$$

In other words, the ith element of the vector x is the sum of the first i elements of the vector a. This problem is solvable on a sequential machine in O(n) time.

If the tree machine and Ringmachine are treated as peripheral functional units that are given a and produce x, the performance of the two machine is identical. Loading and unloading the vectors again dominates the time complexity. In each case, n processors are used to solve the problem in O(n) time.

A more interesting formulation of the problem assumes that the tree and Ringmachine are already loaded with some convenient arrangement of a. How fast can x be generated, with x ending up in the same arrangement as a?

Given the arrangements shown in Figure 12, the Ringmachine uses n processors to generate x in place in O(n) steps. The tree machine, on the other hand, uses n processors, but arrives at the answer in $O(\log_2 n)$ steps. For large n, this is a significant difference.

```
CLASS sum(s,max); INTEGER s,max;
BEGIN END;

Processor CLASS vectorSum;
BEGIN
   INTEGER subscript;
   INTEGER x;

   REF(sum) PROCEDURE sumup;
   BEGIN
      IF left==NONE AND right==NONE
      THEN sumup:-NEW sum(x,subscript)
      ELSE BEGIN REF(sum)l,r;
         l:-IF left==NONE THEN NEW sum(x,subscript) ELSE left.sumpup;
         r:-IF right==NONE THEN NEW sum(x,subscript) ELSE right.sumpup;
         x:=x+l.s;
         sumup:-NEW sum(x+r.s,r.max);
         left.sumdown(l);
         right.sumdown(NEW sum(x,subscript));
      END;
   END of procedure sumup;

   PROCEDURE sumdown(p); REF(sum)p;
   BEGIN
      IF p.max<subscript THEN x:=x+p.s;
      IF left=/=NONE THEN left.sumdown(p);
      IF right=/=NONE THEN right.sumdown(NEW sum(x,subscript);
   END of procedure sumdown;
END of class vectorSum;
```

Figure 13. Algorithm for finding x_i.

	Sequential Machine		Tree Machine	
	space	time	processors	time
Heap Sort	n	$n\log_2 n$	n	n
Matrix Multiplication	n^2	$\sim n^3$	n^2	n^2
Color Cost	n	n^n	n^n	n^2
Transitive Closure	n^2	n^3	n^2	n^2
x_i	n	n	n	$\log_2 n$

Figure 14. Sequential and Tree Machine Performance.

The Ringmachine algorithm is straightforward. Starting with the vector a distributed as in Figure 12, each processor adds numbers that are passed in from the left to the ai it holds before passing them on. After the ith processor has seen i-1 numbers, it sends ai to the right and becomes dormant. The nth processor waits n-1 time steps for a_1. The other n-2 elements arrive in the next n-2 time steps, and are added to a_n to form x_n. Thus the process is complete after 2n-1 cycles.

The algorithm on the tree machine is not as simple. The arrangement of the ai's given in Figure 12 is not intuitive. And the algorithm requires data to flow up and down the tree simultaneously. The SIMULA code is given in Figure 13.

The summing starts in the lower left hand corner of the tree. Each node gets partial sums from its left and right children. The left hand sum is added to the a_i in the processors, stored as x_i, and passed to the right child. Then the sum from the right child is added in, and this result, the sum of all three numbers available, is sent to to the parent processor. It takes $\log_2 n$ cycles for the root to receive the sum of the a_i's in the left half of the tree, and another $\log_2 n$ steps for that sum to filter down to the lower right corner, forming x_n.

The algorithm described above uses the extra communication paths of the tree to advantage. It remains to be seen if the problem can be put back into the numerical analysis context from which it came, and still perform better on the tree than on the Ringmachine.

Conclusions.

The work described in this paper is aimed at deciding two questions. First, are multiprocessor systems useful? And if so, what kind of system should be built?

The answer to the first question is a resounding yes. Figure 14

compares the performance of the algorithms described here on sequential machines and the tree machine. In each case, the time complexity is substantially reduced.

The second question does not yet have a clear answer. I am just beginning to examine problems that can use the three-neighborness of the tree to advantage. Unless the additional complexity of building a tree rather than a Ringmachine can be justified, the simpler structure is heavily favored. I am hopeful, however, that numerical analysis problems will demonstrate the value of the tree machine.

References

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

[2] Birtwistle, G.M., O-J Dahl, B. Myhrhaug, K. Nygaard
SIMULA BEGIN
Petrocelli, New York, 1973

[3] Browning, Sally A.
"Transitive Closure and the Tree Machine"
Computer Science Department Display file #2402
California Institute of Technology, 1978

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

[5] Demuth, H.B.
"Electronic Data Sorting"
PhD. Thesis (Stanford University, October 1956)

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

[7] Ullner, Mike
"Ringmachine"
Computer Science Department Display file in progress
California Institute of Technology, 1978

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