1985

Solving Tree Problems on a Mesh-Connected Processor Array

Mikhail J. Atallah
Purdue University, mja@cs.purdue.edu

Susanne E. Hambrusch
Purdue University, seh@cs.purdue.edu

Report Number:

85-518

SOLVING TREE PROBLEMS ON A MESH-CONNECTED PROCESSOR ARRAY

Mikhail J. Atallah
Susanne E. Hambrusch

CSD-TR-518
May 1985

SOLVING TREE PROBLEMS ON A MESH-CONNECTED PROCESSOR ARRAY[†]

Mikhail J. Atallah and *Susanne E. Hambrusch*

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907.

Abstract

In this paper we present techniques that result in $O(\sqrt{n})$ time algorithms for computing many properties and functions of an n-node forest stored in an $\sqrt{n} \times \sqrt{n}$ mesh of processors. Our algorithms include computing simple properties like the depth, the height, the number of descendents, the preorder (resp. postorder, inorder) number of every node, and a solution to the more complex problem of computing the Minimax value of a game tree. Our algorithms are asymptotically optimal since any nontrivial computation will require $\Omega(\sqrt{n})$ time on the mesh. All of our algorithms generalize to higher dimensional meshes.

Key Words

Analysis of algorithms, graph theory, mesh of processors, parallel computation.

† This work was supported by the Office of Naval Research under Contract N00014-84-K-0502 and by the National Science Foundation under Grants DCR-84-51393 and DMC-84-13496.

1. Introduction

Suppose we have a $\sqrt{n} \times \sqrt{n}$ mesh of processors as shown in Figure 1, where each processor has a fixed (i.e., $O(1)$) number of storage registers, and can communicate only with its four neighbours. The description of an n-node undirected forest is stored in the mesh; i.e., each processor contains an edge $\{i, j\}$ of the forest. Typical problems to be solved on a forest, which are not only interesting in their own right but also arise as subproblems in other graph problems [AK, H, TC, TV], are rooting every tree in the forest (the result is called a directed forest), computing the depth, the height, and the number of descendents of every node in a directed forest, and computing the preorder (resp. postorder, inorder) number [AHU] of every node in a directed forest.

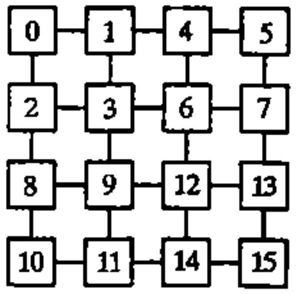


Figure 1. A 4×4 mesh with shuffled row-major indexing

While an algorithm designed for the Shared Main Memory model can always be simulated on a mesh (or any other fixed interconnection network), such a simulation usually does not result in the most efficient algorithm, since special characteristics and properties of the mesh are not taken into consideration. We present techniques that result in $O(\sqrt{n})$ time algorithms for the above mentioned basic problems, for computing the Minimax value of a game tree, and for a number of other problems. These techniques will be useful to anyone designing algorithms for the mesh, a popular model for parallel computation. The algorithms reported in [GRK] for solving basic tree problems on the mesh take $O(\sqrt{n} \log n)$ time, and are obtained by implementing, on the mesh, ideas developed for parallel algorithms on the Shared Main Memory model [HCS, TC, TV]. Stout [S] has independently solved some of the problems considered in Section 3 in $O(\sqrt{n})$ time by using an approach different from ours.

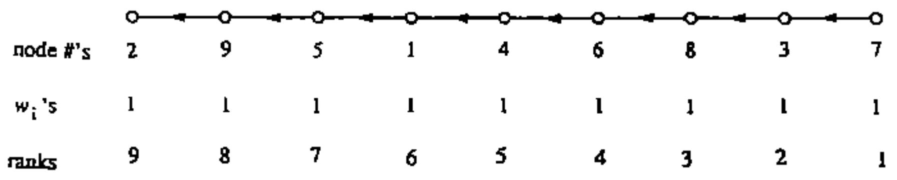
This paper is organized as follows. Section 2 gives an $O(\sqrt{n})$ time algorithm for a problem whose solution is a subroutine of all the algorithms described in the subsequent sections. In

Section 3 we show how to solve a number of basic tree problems in $O(\sqrt{n})$ time; i.e., finding the depth, the height, the number of descendents, and preorder (resp. postorder, inorder) number of every node of a directed tree, and turning an undirected tree into a directed one. Section 4 gives an $O(\sqrt{n})$ time algorithm for the problem of computing the Minimax value. This latter algorithm uses the results of the previous sections. In Section 5 we explain how to extend our results to forests, and point out how to use our techniques for optimally evaluating an arbitrary arithmetic expression tree and for solving other graph problems on the mesh. The paper assumes that the reader is familiar with the standard data movements that can be done in time $O(\sqrt{n})$ on the mesh (see [NS1, NS2, U] for details).

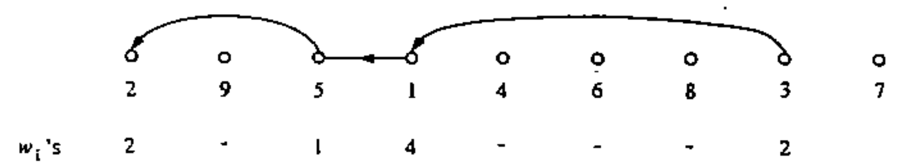
2. Weighted Ranking of a Linear Chain

In order to compute the height, the depth, and many other tree functions in time $O(\sqrt{n})$, it is necessary to be able to solve the following problem in $O(\sqrt{n})$ time. Assume an n-edge directed linear chain is stored in a $\sqrt{n} \times \sqrt{n}$ mesh of processors. Every processor contains one arc of the form $(i, succ(i))$, where node $succ(i)$ is the immediate successor of node i, $1 \le i \le n$, in the linear chain defined by the function $succ$. If i is the last node on the chain, then no processor contains an arc of the form $(i, succ(i))$. The processor containing an arc $(i, succ(i))$ also contains a weight w_i associated with node i (if $succ(i)$ is the last node in the chain, then that processor also contains $w_{succ(i)}$). The *rank* $R(i)$ of a node i is the sum of the weights of its predecessors (including itself) in the chain defined by the $succ$ function. See Figure 2.1(a) for an example. If the mesh contains a collection of node-disjoint chains rather than a unique chain, then obviously the rank of a node is with respect to the chain to which the node belongs. In this section we show how to compute the rank $R(i)$ of every node i in $O(\sqrt{n})$ time.

The obvious divide-and-conquer approach in which the mesh is divided into four submeshes, which are solved independently and subsequently merged, does not result in an efficient algorithm. The problem is in the merging step: Chains may 'jump' a large number of times between two submeshes, making it seemingly impossible to combine the four partial solutions in $O(\sqrt{n})$ time. Our algorithm too uses a divide-and-conquer strategy. It uses the above-mentioned technique to solve, in time $O(\sqrt{n})$, a special case of the problem which has a property that allows



(a) a linear chain; 2 is the last node and 7 is the first node in the chain.



(b) after step 4 of algorithm CHAIN_RANK.

Figure 2.1

the merging of subsolutions in time $O(\sqrt{n})$. The algorithm for the general case uses the one for the special case in order to reduce, in $O(\sqrt{n})$ time, the initial problem of size n to one of size no more than $n/2$. The recurrence for the time $T(n)$ taken by the algorithm computing the ranks is shown to be of the form $T(n) \leq c\sqrt{n} + T(n/2)$, which implies that $T(n)$ is $O(\sqrt{n})$. Before giving a precise description of the general ranking algorithm, we describe how to efficiently compute the ranks in chains of a special type.

Assume that k processors of the mesh contain one arc each, $k \leq n$, which together define a collection H of node-disjoint chains, and that for every arc $(i, succ(i))$ the property $i > succ(i)$ holds. Recall that a processor containing an arc $(i, succ(i))$ of H also contains the weight associated with node i. If i is a node on H, then the rank of i *with respect to* H is denoted by $R_H(i)$, and it is the sum of the weights of the predecessors of i in the chain of H containing i. Algorithm SIMPLE_RANK, which computes the $R_H(i)$'s in $O(\sqrt{n})$ time, uses both the row major and the shuffled row-major indexing scheme for the processors of the mesh. Recall that in the shuffled row-major indexing scheme the processors with indices $1, \cdots, n/4$ are the ones in submesh I, where submesh I is as shown in Figure 2.2. Within submesh I, the processors are indexed using the shuffled row-major indexing scheme. Submeshes II, III, and IV are filled analogously. See Figure 1 for an example and [TK] for precise definitions.

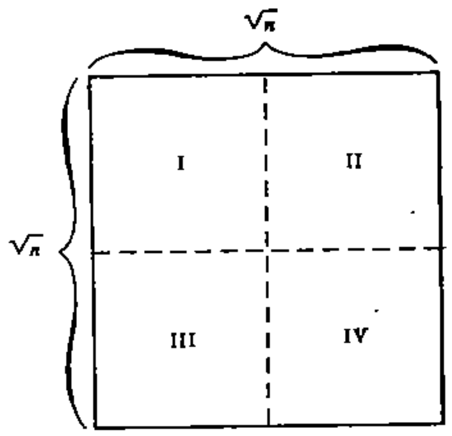


Figure 2.2

Algorithm SIMPLE_RANK

Input: Collection H of chains such that every arc $(i, succ(i))$ in H has the property $i > succ(i)$.

Output: The $R_H(i)$'s; i.e., at the end of the computation every processor containing an arc $(i, succ(i))$ of H also contains $R_H(i)$.

Step 1: Sort the entries $(i, succ(i), w_i)$ according to i and store them in the mesh according to the shuffled row-major indexing scheme.

Comment: Let H_α be that portion of H obtained by considering only the arcs of H that are stored in submesh α, $\alpha \in \{I, II, III, IV\}$. Then after Step 1, for every arc $(i, succ(i))$ in H_α, the arc $(succ(i), succ(succ(i)))$ is stored in $H_{\hat{\alpha}}$ where $\hat{\alpha} \leq \alpha$. This holds because of the property $i > succ(i)$.

Step 2: Recursively compute for each one of the four submeshes $H_I, \cdots, H_{IV}$ the ranks with respect to the portion of H stored in it; i.e., submesh α computes the $R_{H_\alpha}(i)$'s. This does not yet give the final values of the $R_H(i)$'s, since a chain in H may extend over more than one submesh. But a chain in H cannot cross submesh boundaries more than 3 times, because of the comment following Step 1. This last property is crucial for Step 3 to run in $O(\sqrt{n})$ time. Note that for every node i in H_{IV}, we have $R_{H_{IV}}(i) = R_H(i)$.

Step 3: In order to combine the results of Step 2 to obtain the ranks with respect to H, first combine the results in submesh I with those in II to get the $R_{H_I \cup H_{II}}(i)$'s, and simultaneously (in parallel) combine the results in III with those in IV to get the $R_{H_{III} \cup H_{IV}}(i)$'s. Then combine the two so obtained results in order to get the final ranks $R_H(i)$ of the nodes i in the upper half (regions I and II). (For every node i in the lower half the final rank is then already known, since $R_H(i) = R_{H_{III} \cup H_{IV}}(i)$.)

Implementation of Step 3: We describe the "merging" step only for the case of combining regions I and II to get the $R_{H_I \cup H_{II}}(i)$'s (the other computations of Step 3 are analogous). First determine in submesh II all arcs $(i, succ(i))$ for which $succ(i)$ is the last node of a chain in H_{II}. Then, for every such i, do the following: (i) send $R_{H_{II}}(i)$ to the processor of submesh I which contains the arc $(succ(i), succ(succ(i)))$, and (ii) add the value of $R_{H_{II}}(i)$ to the current rank of every node in H_I that is in the same chain as node $succ(i)$ (including $succ(i)$).

Determining the nodes i and performing step (i) can easily be done in $O(\sqrt{n})$ time by using standard data movement techniques [U]. Step (ii) is done in $O(\sqrt{n})$ time by first determining the connected components induced by the arcs in H_I so that arcs in the same connected component can be arranged to occupy adjacent processors. This takes $O(\sqrt{n})$ time, since the connected components of any n-node forest can be found in $O(\sqrt{n})$ time by an easy application of the techniques of [NS1]. (Actually it has recently been shown [RS,K] that this holds for arbitrary graphs, not just forests.) After this connected components computation, all $R_{H_{II}}(i)$'s can be propagated to the appropriate entries in $O(\sqrt{n})$ time.

End of Algorithm SIMPLE_RANK.

If we let $F(n)$ be the time required for determining the ranks of all the nodes in H, then we have $F(n) \le F(n/2) + c\sqrt{n}$, which implies that $F(n)$ is $O(\sqrt{n})$. It is clear that an analogous algorithm exists for an H with $i < succ(i)$ for every arc $(i, succ(i))$ in H. We now describe the algorithm that computes the rank of every node with respect to arbitrary chains (i.e., chains in which $i < succ(i)$, respectively $i > succ(i)$, does not hold for all i).

Algorithm CHAIN_RANK

Input: Every processor contains an arc $(i, succ(i))$ and a weight w_i. The function $succ$ defines an n-edge linear chain.

Output: Every processor containing $(i, succ(i))$ also contains $R(i)$, the sum of the weights of the predecessors of node i in the n-edge linear chain defined by the function $succ$.

Step 1: Let n_1 (resp. n_2) be the number of processors containing an entry with $succ(i) < i$ (resp. $succ(i) > i$). Determine which of n_1 and n_2 is the larger, and broadcast the outcome to every processor. Without loss of generality the algorithm assumes throughout that $n_1 \geq n_2$. (Note that in this case $n_1 \geq n/2 \geq n_2$.)

Step 2: Let H be the collection of chains obtained by considering only those arcs $(i, succ(i))$ with $succ(i) < i$. From Step 1 it follows that the total number of arcs of the chains in H is at least $n/2$. The $R_H(i)$'s are computed in $O(\sqrt{n})$ time as described in algorithm SIMPLE_RANK.

Step 3: For every chain in H, determine the node that is the immediate predecessor of the first node of that chain in the original input chain. For a given chain in H, let l be this node. For example, in Figure 2.1(a) node 3 is the immediate predecessor of node 8, and node 8 is the first node of the chain (8,6), (6,4), (4,1). Broadcast l to all the other nodes in the same chain. This is done, in parallel for all chains of H, in time $O(\sqrt{n})$ by using known techniques.

Comment: The purpose of this step is to reduce the problem of computing the ranks of nodes on H to that of computing the rank of the immediate predecessor of the first node of every chain in H. If $(l, succ(l))$ is an arc with $succ(l)$ being the first node of a chain in H, and $R(l)$ is known, then the final rank of every node v in the same chain in H as $succ(l)$ is $R_H(v) + R(l)$.

Step 4: Modify the original input chain by "bypassing" the chains in H as follows: Let $i_1, \cdots, i_{k-1}, i_k$ be a chain in H and $succ(l) = i_1$ for some l. $R_H(i_1), \cdots, R_H(i_k)$ have already been computed by the previous call to SIMPLE_RANK. Set $succ(l)$ equal to i_k (i.e., the last node of the chain) and set the weight of node i_k to $R_H(i_k)$. ($R_H(i_k)$ is stored in the processor containing the arc (i_{k-1}, i_k).) See Figure 2.1(b) where $succ(3)$ is set to 1 and the weight of node 1 is set to 4. This new weight now reflects the weight of the "bypassed" nodes. Note that the surviving chain has length $n_2 \leq n/2$, and that the (yet to be computed) ranks of nodes on that chain are the same as their ranks in the original full chain.

Comment: Recall that in the chain $i_1, \cdots, i_k$, every i_j knows $R_H(i_j)$ as well as node l. Therefore when, at a later stage, we know $R(l)$, then $R(i_j)$ is obtained by simply adding

$R_H(i_j)$ and $R(l)$.

Step 5: Compress the n_2 arcs of the surviving chain so that they are stored in the $\sqrt{n_2}\times\sqrt{n_2}$ top-left submesh. See Figure 2.3. Use the $\sqrt{n_2}\times\sqrt{n_2}$ submesh to recursively solve the remaining problem: that of computing ranks of the nodes in the surviving chain.

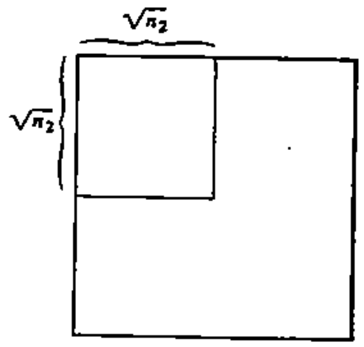


Figure 2.3

Step 6: Update the ranks of the nodes in H, as explained in the comment following Step 3. Note that the nodes in the chain used by the recursive call of Step 5 do not need to update their ranks (since these ranks are computed correctly by the recursive call).

End of Algorithm CHAIN_RANK.

The correctness proof of the above algorithm is easy and is omitted. That it runs in time $O(\sqrt{n})$ follows from the fact that the data movements required in every step can be done in $O(\sqrt{n})$ time and that Step 5 makes a recursive call on a square mesh of size $\sqrt{n_2}\times\sqrt{n_2}$, where $n_2 \le n/2$. Note that the algorithm can easily be modified to compute the rank of a linear chain consisting of cn arcs stored in the mesh such that every processor contains no more than c' arcs (c and c' being constants). This concludes the description of the weighted ranking algorithm, which will be used as subroutine by the algorithms in the following sections.

3. Some Applications of Chain-Ranking

This section shows how algorithm CHAIN_RANK (given in Section 2) can be used to optimally compute various tree functions on the mesh. The idea is to create, from the input tree T, a linear chain $chain(T)$ and to use algorithm CHAIN_RANK on $chain(T)$. The weights assigned to the nodes of $chain(T)$ depend on which particular tree function is being computed.

We also show how to root an undirected tree in $O(\sqrt{n})$ time. The results of this section follow from Section 2 without too much effort, and the idea of creating a linear chain from a tree is a well known technique [TV]. A more complex tree computation, which makes use of the results of this section in an interesting way, will be described in Section 4. We start by describing how to create a linear chain from a given tree.

Let T be an n-node tree rooted at node r. T is represented by the arcs $(i,p(i))$, where $p(i)$ is the parent of node i, $1 \le i \le n$. Each processor of the mesh contains exactly one arc, with a "dummy" arc $(r,0)$ present for the root r. Imagine "wrapping" a chain around T in the manner depicted in Figure 3.1(a), where the dashed line represents the chain. Note that the chain goes through a node v $\delta(v)+1$ times, where $\delta(v)$ is the number of children of node v. Furthermore, from node v the chain visits the children of v in increasing order of their index.

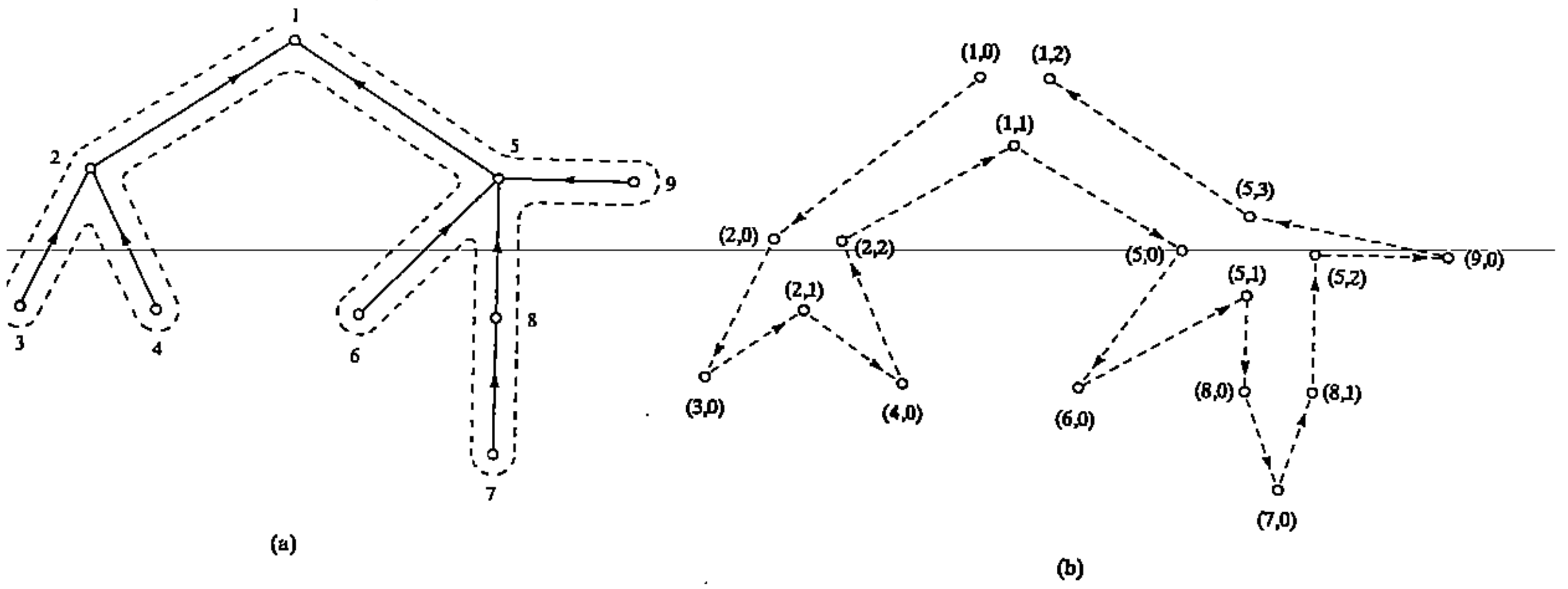


Figure 3.1

For the purpose of making the $2n-1$ nodes on the chain distinct from each other, we distinguish between the various occurrences of node v on the chain by referring to them as $v_0, \cdots, v_{\delta(v)}$. If node w is the i-th child of node v in T, $1 \le i \le \delta(v)$, then, in the chain, node v_i is the successor of node $w_{\delta(w)}$, and node v_{i-1} is the predecessor of node w_0. We refer to the chain obtained in this way from a tree T as *chain*(T). For the tree T shown on Figure 3.1(a), *chain*(T) is shown in Figure 3.2(b). Throughout this paper, if $w=v_k$ is a node on *chain*(T), then we assume that a processor containing that node can tell that it is the k^{th} occurrence of node v of T (this can easily be achieved by storing node $w = v_k$ as a pair (v, k), as done in Figure 3.1(b)).

Given a tree T, the following algorithm creates $chain(T)$ in $O(\sqrt{n})$ time.

Algorithm CREATE_CHAIN

Input: n arcs $(i, p(i))$ that define an in-tree T rooted at node r, with a dummy arc $(r, 0)$ out of the root. Node $p(i)$ is the parent of i in T.

Output: $2n-2$ arcs $(w, succ(w))$ that describe $chain(T)$.

Step 1: For every node i in T determine $\delta(i)$, the number of children of i; i.e., the number of arcs $(j, p(j))$ with $p(j)=i$.

Step 2: For every arc $(i, p(i))$ determine $s(i)$, the number of children of $p(i)$ that are no greater than i; i.e., the number of arcs $(j, p(j))$ with $p(j)=p(i)$ and $j<i$.

Step 3: For every arc (i, j) of T (except $(r, 0)$) generate two directed arcs of $chain(T)$, namely $(j_{s(i)}, i_0)$ and $(i_{\delta(i)}, j_{s(i)+1})$.

End of Algorithm CREATE_CHAIN.

Both the $\delta(i)$'s of Step 1 and the $s(i)$'s of Step 2 can easily be computed in $O(\sqrt{n})$ time, Step 3 is done in constant time, and thus $chain(T)$ can be created in $O(\sqrt{n})$ time.

We now turn our attention to the problem of computing the *depth* of every node v in an n-node rooted tree T. In the first step of the algorithm we create $chain(T)$ using algorithm CREATE_CHAIN. In the second step of the algorithm we set a weight for every node in the chain as follows. If (v_i, w_j) is an arc in the chain, then node w_j has a weight of +1 if and only if the arc (w, v) is in T (i.e., $p(w)=v$), and node w_j gets a weight of -1 otherwise (i.e., if $p(v)=w$). The weight of the first node in $chain(T)$ is set to 0. We then call algorithm CHAIN_RANK to determine the rank of every node in the chain. The depth of every node v in T is then the rank of v_0 in $chain(T)$. (Actually, the rank of any v_k in $chain(T)$ will do, since it too equals the rank of v_0.) Correctness follows from the definition of depth and the way weights were assigned to the nodes of $chain(T)$.

Computing the *number of descendents* of every node in time $O(\sqrt{n})$ is similar to the depth computation, and we only describe the differences. We assign to every node v_i in $chain(T)$ a weight of +1 if $i=\delta(v)$, and a weight of 0 if $0 \le i < \delta(v)$. In other words, the last occurrence of node v of T on *chain* (T) is given a weight of unity, while other occurrences of v are given a weight of zero. The number of descendents of v in T is equal to $R(v_{\delta(v)})-R(v_0)$, the rank of the last occurrence of v minus the rank of its first occurrence on *chain* (T).

We now describe how to compute the *height* of every node in a tree T. The algorithm begins by computing the depth of every node in T, as explained above. A byproduct of this depth computation is *chain* (T). Observe that the height of v equals the depth of the deepest node in the subtree of v minus the depth of v. If we let $z(v)$ denote this deepest leaf under node v, then *depth* $(z(v))$ is simply the maximum rank of any node which occurs between v_0 and $v_{\delta(v)}$ in *chain* (T). We briefly outline how to compute *depth* $(z(v))$ in $O(\sqrt{n})$ time, in parallel for every node v.

Assume that the arcs of *chain* (T) are stored in the mesh in row-major order according to the depth of the nodes in *chain* (T). First determine for every row i the maximum rank associated with an arc stored in row i, and broadcast this value to all the processors in column i, $1 \le i \le \sqrt{n}$. The computation of these max-row values can easily be done in $O(\sqrt{n})$ time. The maximum rank of any node which occurs between v_0 and $v_{\delta(v)}$ (and which is the depth of leaf $z(v)$) could be one of these max-row values, or it could be the maximum of two partial rows. Let i_1 (resp. j_1) be the row (resp. column) of the processor containing the arc $(v_0, succ(v_0))$, and let i_2 (resp. j_2) be the row (resp. column) of the processor containing the arc $(w, v_{\delta(v)})$. In parallel, determine for every node v_0 the maximum rank in row i_1 (resp. i_2) beginning at column j_1 (resp. ending at column j_2). The depth of node $z(v)$ is the maximum of these two values and the max-row values of rows $i_1+1, \cdots, i_2-1$. By taking the later values from row i_1, we avoid any 'congestion' problems. This concludes the description of the algorithm for computing the height.

The following theorem summarizes the above results:

Theorem 3.1 Given that an n-node directed tree is stored in the mesh, the depth, the height, and the number of descendents of every node can be computed in time $O(\sqrt{n})$.

Other consequences of the result of Section 2 are stated below without proof, since the proofs are very similar to the ones of Theorem 3.1.

Theorem 3.2 Given that an n-node ordered and directed tree is stored in the mesh, the preorder, postorder, and inorder numbers of every node can be computed in time $O(\sqrt{n})$.

In many graph algorithms there is a need to create a directed version of an undirected tree. We next describe an algorithm that generates, in $O(\sqrt{n})$ time, a rooted (directed) tree T from an undirected tree Q. The undirected tree Q is initially stored in the mesh in the obvious way: Each processor contains an (undirected) edge $\{x,y\}$, $1 \le x, y \le n$. Let r be the node to be made the root of T. The main idea of algorithm MAKE_ROOTED is to first use Q to create a chain of T (this is what the first four steps of the algorithm do), and then to use algorithm CHAIN_RANK on this chain to obtain T.

Algorithm MAKE_ROOTED

Input: $n-1$ edges $\{x,y\}$ that form an undirected tree Q, and a designated node r.

Output: $(n-1)$ arcs forming an in-tree T rooted at r and which is a directed version of the input tree Q.

Step 1: In parallel for every node x determine $d(x)$, the degree of node x in Q.

Step 2: In parallel for every node x, create $d(x)$ copies of that node, and call them $x_0, \cdots, x_{d(x)-1}$. Then create a directed cycle C_x of length $2d(x)$ that alternates between the $d(x)$ copies of x and the $d(x)$ neighbours of x in Q; i.e.,

$$C_x = x_0\, u\, x_1\, v\, \cdots\, x_{d(x)-1}\, w\, x_0,$$

where $u, v, \cdots, w$ are the neighbours of x in Q (the original neighbours, *not* the copies of these neighbours).

Comment: The union of the C_x's consists of $\sum_{x=1}^{n} 2d(x) = O(n)$ arcs, each of which is between a "real" node and a "copy" of another node.

Implementation Note: The arcs which make up the C_x's are created as follows. Create two arcs (x,y) and (y,x) for every edge $\{x,y\}$ of Q; for every so created arc (x,y) determine $s(x,y)$, the number of arcs (x,z) with $z<y$. Then replace every arc (x,y) by the arcs $(x_{s(x,y)}, y)$ and $(y, x_{(s(x,y)+1) \bmod d(x)})$.

Step 3: Replace every pair of arcs of the form (x_k, y) and (x, y_l) by the single arc (x_k, y_l). This is done by sorting the arcs of the C_x's so that every arc (x_k, y) is in the same processor as the arc (x, y_l), and then having that processor remove both of these two arcs and create the new arc (x_k, y_l).

Comment: The effect of this step is to "stitch" the C_x's together into one giant cycle that goes through every copy of every node exactly once. The next step "opens" this cycle at node r_0, thus creating *chain* (T).

Step 4: Create node $r_{d(r)}$, which is an additional copy of the root node r, and change the arc (z_k, r_0) to $(z_k, r_{d(r)})$. Note that at this time the arcs we created form a chain of T. The following step extracts T from this chain.

Comment: The reader may have noticed that the copies of a node x do not necessarily appear in the order $x_0, x_1, \cdots$ on the chain. This is of no consequence.

Step 5: Use algorithm CHAIN_RANK on the chain obtained in the previous steps, with every weight set to unity (i.e. every $w_i=1$). Then for every edge $\{x,y\}$ of Q, determine which of the two arcs, (x,y) or (y,x), is in T. This is done as follows. Let x_k (resp. y_t) be the smallest-ranked occurrence of x (resp. y) on the chain: If the rank of x_k is smaller than that of y_t then x is the parent of y in T and therefore arc (y,x) is in T, otherwise it is the arc (x,y) which is in T (recall that T is an in-tree).

End of Algorithm MAKE_ROOTED.

Correctness of the above algorithm follows from the comments included in its description. That it runs in $O(\sqrt{n})$ time is also easy to see, once we know that CHAIN_RANK runs in time $O(\sqrt{n})$. We therefore have the following:

Theorem 3.3 Given that an undirected n-node tree is stored in the mesh, rooting that tree can be done in time $O(\sqrt{n})$.

The next section gives an $O(\sqrt{n})$ time algorithm for a more difficult tree computation: The Minimax value problem for an n-node game tree.

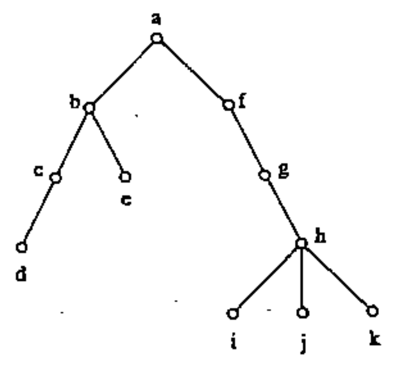
4. Computing the Minimax Value.

This section contains a complex, but rather elegant result: An $O(\sqrt{n})$ time algorithm for computing the Minimax value of an arbitrary n-node game tree. In this problem we are given an n-node directed tree T rooted at node r in which every leaf has a real number, called the *value of* of the leaf, attached to it, and every internal node is of type Min or Max. If the value of every leaf is either 0 or 1, then the tree is called a *0/1 game tree*. The problem is to compute $VAL(T)$, the value of the root r of T. If j is an internal node of type Min (resp. Max), then the value of j is the minimum (resp. maximum) of the values of the children of j. The main ingredients of our Minimax algorithm are a relationship between game trees with real values at the leaves and 0/1 game trees, an algorithm for efficiently computing the value of a 0/1 game tree, and the results of Section 3.

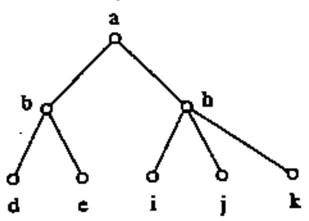
We start by establishing the relationship between arbitrary game trees and 0/1 game trees. We show how to reduce the problem of computing the value of an n-node game tree with real values associated with the leaves to that of computing the values of $\log n$ successive instances of 0/1 game trees, where the i-th instance is of size at most $c^i n$ and $c<1$ is a constant. Let T be an n-node game tree with λ leaves, and let $a_1, \cdots, a_\lambda$ be the numbers attached to its leaves (not in any particular order). Without loss of generality assume $a_1 \leq \cdots \leq a_\lambda$. Let T_i be the 0/1 game tree obtained from T by replacing every a_j by 0 if $a_j < a_i$, and by 1 if $a_j \geq a_i$. Let $VAL(T_i)$ be the value of the root of T_i. Observe that $VAL(T_i)=1$ implies $VAL(T_j)=1$ for every $j<i$, while $VAL(T_i)=0$ implies $VAL(T_j)=0$ for every $j>i$. If we let $\alpha = \max\{i \mid VAL(T_i)=1\}$, then we have $VAL(T)=a_\alpha$. To see this, simply note that $VAL(T_i)=1$ iff $VAL(T) \geq a_i$, and that $VAL(T_i)=0$ iff $VAL(T)<a_i$. (The notation just introduced will be used throughout this section.)

The above observations imply that, if $VAL(T_i)$ can be computed in $O(\sqrt{n})$ time, then $VAL(T)$ can be determined in $O(\sqrt{n}\log n)$ time by using binary search to compute the largest i such that $VAL(T_i)=1$. In this section we not only show how to compute $VAL(T_i)$ in $O(\sqrt{n})$ time, but we also remove the $\log n$ factor, and thus obtain an $O(\sqrt{n})$ time algorithm for computing $VAL(T)$. We continue the discussion assuming that $VAL(T_i)$ can indeed be computed in time $O(\sqrt{n})$.

We henceforth assume that every internal node of a game tree T has at least two children. If this is not so, then we can replace T by an equivalent tree $\hat{T}$ in which nodes with one child have been eliminated by "bypassing" them (see Figure 4.1). This "bypassing" operation can be done in $O(\sqrt{n})$ time by using essentially the same techniques as in Section 2, and therefore we omit the details of how this is done. The fact that every internal node of T has at least two children implies that $\lambda > n/2$ (recall that λ is the number of leaves and n is the total number of nodes of T).



tree T



tree $\hat{T}$

Figure 4.1

We achieve $O(\sqrt{n})$ time performance for $VAL(T)$ by using binary search where after each "probe" of the binary search, we reduce the problem size to 3/4 of the original size. That is, if before a probe of the binary search the problem size was m, then

(i) the probe takes $O(\sqrt{m})$ time, and

(ii) the problem can be reduced to an equivalent problem of size no more than $3m/4$, also in time $O(\sqrt{m})$.

This implies that the i^{th} probe of the binary search will take time $O(\sqrt{(3/4)^{i-1}n})$, and therefore the entire algorithm for computing $VAL(T)$ takes $O(\sqrt{n})$ time. We leave the description of step (i) (i.e., how an m-node 0/1 game tree is evaluated in $O(\sqrt{m})$ time) for later, and continue with the discussion of the size reduction step. We give the details only for the reduction which follows the first probe of the binary search; i.e., after computing $VAL(T_{\lambda/2})$. Without loss of generality, assume that the result of this first probe is $VAL(T_{\lambda/2})=0$; i.e., the next probe will compute $VAL(T_{\lambda/4})$. The idea is not to use $T_{\lambda/4}$ itself in the next probe, but rather a *smaller* tree which has the same value as $T_{\lambda/4}$. This is made possible by the following observation: Since all the subsequent probes will be on T_i's with $i<\lambda/2$, the values of the leaves in T which were $a_{\lambda/2}, \cdots, a_{\lambda}$ will remain 1 in every such T_i. Therefore, we can "remove" the leaves containing the values $a_{\lambda/2}, \cdots, a_{\lambda}$ from T, and, as far as subsequent probes are concerned, replace T by a new version of T as follows:

(i) If i is a leaf of T which contains an a_j with $j \geq \lambda/2$, then remove i from T.

(ii) Let k be an interior node of T that has at least one child removed in step (i) and that is of type Max. Make k a leaf of T (by deleting its remaining children and their subtrees), and give k the value a_λ. The justification for this is obvious: In all subsequent 0/1 probes a removed leaf (or leaves) will have value 1, forcing the value of k to be 1 (because k is of type Max). Making k a leaf with a value of a_λ achieves the same effect.

(iii) Let k be an interior node of T that has all its children removed in step (i) and that is of type Min. In this case make k a leaf of T and give it the value a_λ. The justification for doing so is similar to the one for (ii).

(iv) If the new version of T resulting from steps (i)-(iii) has any internal nodes with only one child, then modify T so that these nodes are eliminated (this is done by "bypassing" those nodes, as previously explained). The tree T resulting from this step will then have all its internal nodes with at least two children each.

Note that the new tree created in steps (i)-(iv) has the same value as the original tree T, and has no more than $3n/4$ nodes (this last observation follows from the fact that $\lambda > n/2$ and that the new tree has at least $\lambda/2$ fewer nodes than the original one, since step (i) removes $\lambda/2$ leaves). Before proceeding with the next probe of the binary search, we compress the arcs describing the new tree T within the top-left $\sqrt{3n/4} \times \sqrt{3n/4}$ submesh, and it is within this smaller submesh that the rest of the computation will take place. The above discussion was for the case when the first probe resulted in $VAL(T_{\lambda/2})=0$. The case when $VAL(T_{\lambda/2})=1$ can be handled analogously.

In general, the number of steps needed for the size reduction of the i-th probe of the binary search is $O(\sqrt{(3/4)^{i-1}n})$ and therefore the total time taken by the algorithm is $O(\sqrt{n})$, if a given n-node 0/1 game tree Q can be evaluated in $O(\sqrt{n})$. Now we give an $O(\sqrt{n})$ time algorithm for computing $VAL(Q)$. This algorithm makes use of the following lemma, which generalizes the results of sections 2 and 3 to rectangular meshes.

Lemma 4.1 Suppose that an n-node directed tree H is stored in an $l \times w$ rectangular mesh, where $n = l.w$. Then the depth, the height, the number of descendents, and the preorder (resp. postorder, inorder) number of every node can be computed in time $O(l+w)$.

Proof: The results of [A1, KA] imply that any problem that can be solved in time $O(\sqrt{n})$ on a $\sqrt{n} \times \sqrt{n}$ mesh can also be solved in time $O(l+w)$ on an $l \times w$ mesh where $l.w = n$. This, together with theorems 3.1 and 3.2, implies the lemma. □

We need to state the algorithm for computing the value of a 0/1 game tree in terms of a *rectangular* mesh rather than a square mesh, because even though we may start with a square mesh, the recursive calls (which are made on subtrees obtained from a centroid computation) will be for rectangular meshes rather than square ones. (Insisting that recursive calls be on square submeshes runs into trouble, since there may not be enough room in the original mesh for the squares.)

Algorithm 0/1-VALUE

Input: An n-node 0/1 game tree Q, rooted at r. Every arc $(i, p(i))$ of Q is stored in one of the processors of an $l \times w$ rectangular mesh, where $n = l.w$.

Output: $VAL(Q)$ stored in the top-left processor.

Step 0: If $l < 10$ and $w < 10$, then solve the problem in constant time (e.g. using any brute force algorithm). Otherwise proceed to Step 1.

Step 1: Find a *centroid* c of the tree Q. Recall that a centroid of an n-node tree is a node whose removal from the tree disconnects it into connected components none of which has more than $n/2$ nodes. (See [Kn] for a proof of the existence of a centroid.)

Implementation Note: Since the number of descendents of every node can be found in time $O(l+w)$, a centroid can be found in time $O(l+w)$.

Step 2: Mark every node on the path from the the centroid c to the root r (including c and r) as being "special".

Implementation Note: Step 2 is done in time $O(l+w)$ as follows. First, compute the preorder number and the number of descendents of every node. Next, let every processor know the preorder number of c and the number of descendents of c. Finally, the special nodes can be marked in constant time by comparing, for every node i, its preorder number and number of descendents with those of c (such a comparison will reveal whether that node is ancestor of c, i.e. whether it is special).

Step 3: Let $Q_1, \cdots, Q_\xi$ be the collection of rooted trees resulting from the removal of the special nodes from Q. Let r_i denote the root of Q_i (see Figure 4.2). Note that, in tree T, the parent of every r_i is a special node. Assuming (without loss of generality) that $l \geq w$, store the descriptions of $Q_1, \cdots, Q_\xi$ in ξ rectangular submeshes, as shown in Figure 4.3. If n_i is the number of nodes in Q_i then the submesh containing the arcs of Q_i is of size $l_i \times w$, where $l_i = n_i / w$. Of course, no n_i is larger than $n/2$ (since c is a centroid) and therefore $l_i \leq l/2$ for every i. Store the arcs of T' that are not in any Q_i (i.e., the arcs that are incident to a special node) in that part of the mesh not containing the description of any Q_i, as shown in Figure 4.3.

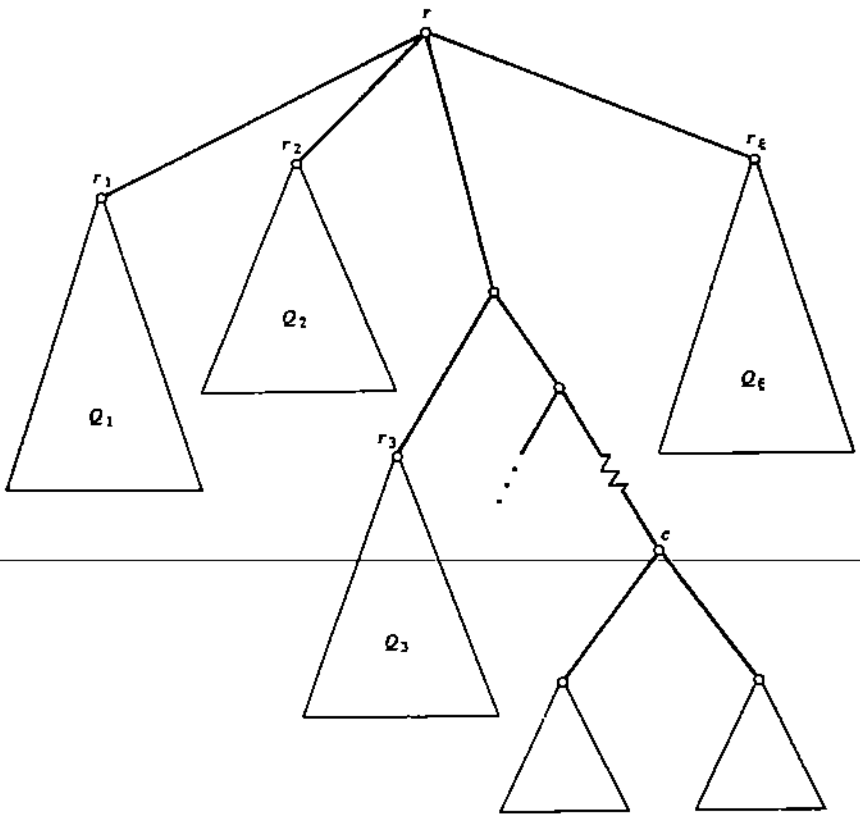


Figure 4.2

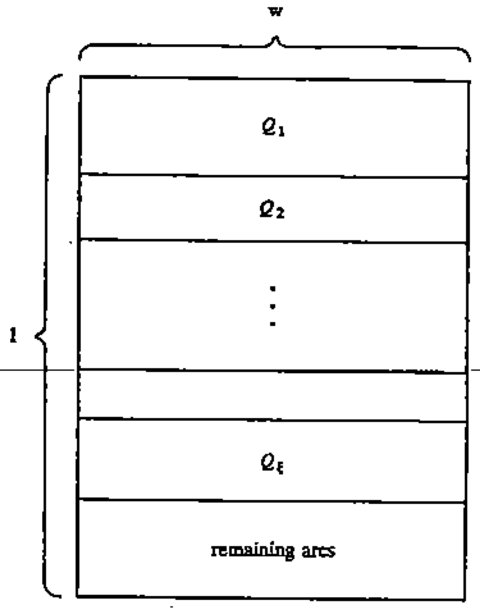


Figure 4.3

Implementation Note: Finding the various Q_i's is essentially a connected components computation which, as already stated, takes $O(l+w)$ time. Compressing the Q_i's into the appropriate submeshes is straightforward and we omit its details.

Step 4: Recursively compute $VAL(Q_i)$ in parallel for every Q_i. If $T(l,w)$ is the total time taken by algorithm 0/1-VALUE, then the cost of this step is no more than $T(l/2,w)$, since every l_i is no larger than $l/2$. (Of course, if we had $l<w$ then the cost of this step would be no more than $T(l,w/2)$.)

Comment: After this step we have the value of every r_i, and therefore we now are left with the problem of computing the values of the special nodes; i.e., the nodes on the path from c to r in T (every r_i is a child of one of these nodes). Actually, we are only interested in the value of one of those special nodes: The root r. The next step computes the value of r, and hence that of Q.

Step 5: Let H be the subtree of Q which consists of the special nodes and the r_i's. Note that the r_i's are the leaves of H, with a value of 0 or 1 attached to each of them. For every r_i whose value is 0 do the following. Let p_i be its parent node. Remove r_i from H. If p_i is of type Min, then make p_i a leaf with value 0. If p_i is of type Max and all of p_i's children have value 0, then make p_i a leaf with value 0. The case when r_i has value 1 is symmetric. Implementing this in $O(l+w)$ time is trivial. After this step, H is a collection of (one or more) chains. The first node in every chain in H has a value of 0 or 1 attached to it. One such chain has r as the last node, and the final result we seek is the value of the first node in the chain containing r. Computing that value can be be done in $O(l+w)$ time by using the techniques of Section 2.

End of Algorithm 0/1-VALUE.

Correctness of the above algorithm is easily proven by induction. That it runs in $O(l+w)$ time is a consequence of the fact that its running time $T(l,w)$ satisfies the following recurrence:

$T(l,w) \le T(l/2,w) + O(l+w)$ if $Max(l,w) \ge 10$, and $l \ge w$

$T(l,w) \le T(l,w/2) + O(l+w)$ if $Max(l,w) \ge 10$, and $l < w$,

$T(l,w) = O(1)$ if $Max(l,w) < 10$.

This implies that $T(l,w) = O(l+w)$. We can therefore state the main result of this section.

Theorem 4.2 Given that an n-node game tree is stored in a $\sqrt{n} \times \sqrt{n}$ mesh, with a real number associated with every leaf and every interior node being of type Min or Max, the Minimax value of the tree can be computed in time $O(\sqrt{n})$.

5. Concluding Remarks

We have presented techniques that lead to $O(\sqrt{n})$ time algorithms for computing many tree functions on a $\sqrt{n} \times \sqrt{n}$ mesh of processors. We now describe how to modify our algorithms to handle the case when the input is a forest, rather than a tree. If the initial input in the $\sqrt{n} \times \sqrt{n}$

mesh is a forest, then we first find its connected components in $O(\sqrt{n})$ time. Let these components be the trees $Q_1, \cdots, Q_\xi$. Store $Q_1, \cdots, Q_\xi$ in rectangular meshes as shown in Figure 4.3 (of course, in this case there are no remaining arcs). Since we have already shown that for a tree stored in an $l \times w$ rectangular mesh our algorithms run in time $O(l+w)$, the results for the forest follow.

The techniques presented in this paper are not limited to the problems mentioned. They can, for example, be used to obtain an $O(\sqrt{n})$ time algorithm for the problem of computing the value of an arithmetic expression of length n. The algorithm for this problem is based on the ideas developed in references [B] and [MR], and the techniques of this paper merely make an $O(\sqrt{n})$ time implementation possible on the mesh. S.R. Kosaraju has pointed out that an approach similar to the one used in the algorithm for evaluating arithmetic expressions is an alternative way of establishing Theorem 4.2 without using binary search.

Another problem for which our techniques result in an $O(\sqrt{n})$ time solution is the problem of optimally placing the minimum number of centers on the nodes or edges of a tree so that every node of the tree is at most distance d away from a center, where d is given. The recursive algorithm for doing so uses a centroid decomposition to generate the subproblems to be solved independently, and it uses a height computation to determine the bottom of the recursion. We omit the details since they are of a similar flavour as the ones for the Minimax algorithm.

Since trees play a fundamental role in so many graph algorithms, it should come as no surprise that the techniques of this paper also enable $O(\sqrt{n})$ solutions to many graph problems, where n now denotes the number of edges of the input graph. For example, the parallel algorithm for finding Euler Tours described in [AV] can be implemented in $O(\sqrt{n})$ time on the mesh (when the computation terminates, the processor containing edge e also contains its predecessor and successor on the resulting Euler Tour). The parallel biconnectivity algorithm of [TV] can also be implemented in time $O(\sqrt{n})$, and so can the known parallel strong orientation algorithm [A2, V]. Implementing these known algorithms in $O(\sqrt{n})$ time on the mesh makes crucial use of our techniques, as the reader can easily verify.

All the algorithms presented for the 2-dimensional mesh generalize to higher dimensional meshes; i.e., they can easily be modified to run in time $O(n^{1/d})$ on an d-dimensional mesh of n

processors.

References

[A1] M. J. Atallah, 'Simulations Between Mesh-Connected Processor Arrays,' *Proc. 23nd Annual Allerton Conference on Communication, Control, and Computing*, Monticello, Illinois, October 1985.

[A2] M.J. Atallah, 'Parallel Strong Orientation of an Undirected Graph,' *Info. Proc. Letters*, Vol. 18, No. 1, January 1984, pp. 37–39.

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

[AK] M.J. Atallah, S.R. Kosaraju, 'Graph Problems on a Mesh-Connected Processor Array', *JACM*, Vol. 31, No. 3, pp. 649-667, July 1984.

[AV] M.J. Atallah and U. Vishkin, 'Finding Euler Tours in Parallel,' *CSS* Vol. 29, No. 3, December 1984, pp. 330–337.

[B] R.P. Brent, 'The Parallel Evaluation of General Arithmetic Expressions,' *JACM*, Vol. 21, No. 2, April 1974, pp. 201–208.

[GRK] P.S. Gopalakrishnan, I.V. Ramakrishnan, L.N. Kanal, 'Computing Tree Functions on SIMD Computers', *Proceedings of 1985 Internatiuonal Conf. on Parallel Processing*, pp. 703-710, 1985.

[H] S.E. Hambrusch, 'Parallel Algorithms for Bridge- and Bi-Connectivity on Minimum Area Meshes', Techn. Report, Purdue University, 1984.

[HCS] D.S. Hirschberg, A.K. Chandra, D.V. Sarwate, 'Computing Connected Components on Parallel Computers', *CACM*, pp. 461-464, Aug. 1979.

[Kn] D.E. Knuth, *The Art of Computer Programming*, Addison Wesley, Reading, MA.

[K] S.R. Kosaraju, personal communication.

[KA] S.R. Kosaraju and M.J. Atallah, 'Optimal Simulations Between Arrays of Processors,' Purdue Univ. Comp. Sci. Tech. Rept. 561, 1985.

[MR] G.L. Miller, J.H. Reif, 'Parallel Tree Contraction and its Application', *Proceedings of 26-th FOCS*, pp 478-489, 1985.

[NS1] D. Nassimi, S. Sahni, 'Finding Connected Components and Connected ones on a Mesh-connected Parallel Computer', *SIAM J. on Comp.*, pp. 744-757, 1980.

[NS2] D. Nassimi, S. Sahni, 'Data Broadcasting in SIMD Computers', *IEEE Transactions on Computers*, pp. 101-106, 1981.

[RS] J. Reif, Q. Stout, personal communication.

[S] Q.F. Stout, 'Tree-Based Graph Algorithms for Some Parallel Computers,' *Proc. of 1985 Int. Conf. on Parallel Processing*, pp 727-730, 1985.

[TC] Y.H. Tsin, F.Y. Chin, 'Efficient Parallel Algorithms for a Class of Graph Theoretic Problems', *SIAM J. on Computing*, pp. 580-599, 1984.

[TK] C. Thompson, H. Kung, 'Sorting on a Mesh-Connected Parallel Computer', *CACM*, pp. 263-271, 1977.

[TV] R.E. Tarjan, U. Vishkin, 'Finding Biconnected Components and Computing Tree Functions in Logarithmic Parallel Time', *Proc. of 25-th FOCS*, pp. 12-20, 1984.

[U] J.D. Ullman, *Computational Aspects of VLSI*, CSP, 1984.

[V] U. Vishkin, 'An Efficient Parallel Strong Orientation,' CS Dept. Tech. Rept., Courant Inst., NYU.