An Efficient Parallel Biconnectivity Algorithm

Article *in* SIAM Journal on Computing · November 1985

DOI: 10.1137/0214061 · Source: DBLP

2 authors:

Robert Endre Tarjan
Princeton University
432 PUBLICATIONS **60,581** CITATIONS

SEE PROFILE

Uzi Vishkin
University of Maryland, College Park
224 PUBLICATIONS **10,205** CITATIONS

SEE PROFILE

SIAM J. COMPUT.
Vol. 14, No. 4, November 1985

AN EFFICIENT PARALLEL BICONNECTIVITY ALGORITHM*

ROBERT E. TARJAN† AND UZI VISHKIN‡

Abstract. In this paper we propose a new algorithm for finding the blocks (biconnected components) of an undirected graph. A serial implementation runs in $O(n+m)$ time and space on a graph of n vertices and m edges. A parallel implementation runs in $O(\log n)$ time and $O(n+m)$ space using $O(n+m)$ processors on a concurrent-read, concurrent-write parallel RAM. An alternative implementation runs in $O(n^2/p)$ time and $O(n^2)$ space using any number $p \leqq n^2/\log^2 n$ of processors, on a concurrent-read, exclusive-write parallel RAM. The last algorithm has optimal speedup, assuming an adjacency matrix representation of the input.

A general algorithmic technique that simplifies and improves computation of various functions on trees is introduced. This technique typically requires $O(\log n)$ time using processors and $O(n)$ space on an exclusive-read exclusive-write parallel RAM.

Key words. parallel graph algorithm, biconnected components, blocks, spanning tree

1. Introduction. In this paper we consider the problem of computing the blocks (biconnected components) of a given undirected graph $G=(V, E)$. As a model of parallel computation, we use a concurrent-read, concurrent-write parallel RAM (CRCW PRAM). All the processors have access to a common memory and run synchronously. Simultaneous reading by several processors from the same memory location is allowed as well as simultaneous writing. In the latter case one processor succeeds but we do not know in advance which. This model, used for instance in [SV82], is a member of a family of models for parallel computation. (See [BH82], [SV81], [V83c].)

We propose a new algorithm for finding blocks. We discuss three implementations of the algorithm:

1. A linear-time sequential implementation.

2. A parallel implementation using $O(\log n)$ time, $O(n+m)$ space, and $O(n+m)$ processors, where $n=|V|$ and $m=|E|$.

3. An alternative parallel implementation using $O(n^2/p)$ time, $O(n^2)$ space, and any number $p \leqq n^2/\log^2 n$ of processors. This implementation uses a concurrent-read, exclusive-write parallel RAM (CREW PRAM). This model differs from the CRCW PRAM in not allowing simultaneous writing by more than one processor into the same memory location. The speed-up of this implementation is optimal in the sense that the time-processor product is $O(n^2)$, which is the time required by an optimal sequential algorithm if the input representation is an adjacency matrix.

Implementation 2 is faster than any of the previously known parallel algorithms [SJ81], [Ec79b], [TC84]. Eckstein's algorithm [Ec79b] uses $O(d \log^2 n)$ time and $O((n+m)/d)$ processors, where d is the diameter of the graph. The first (resp. second) algorithm of Savage and Ja'Ja' [SJ81] uses $O(\log^2 n)$ (resp. $O((\log^2 n)\log k)$) time, where k is the number of blocks, and $O(n^3/\log n)$ (resp. $O(mn+n^2 \log n)$) processors.

* Received by the editors August 11, 1983, and in final revised form August 22, 1984. This is a revised and expanded version of the paper *Finding biconnected components and computing tree functions in logarithmic parallel time*, appearing in the 25th Annual Symposium on Foundations of Computer Science, Singer Island, FL, October 24-26, 1984, © 1984 IEEE.

† AT & T Bell Laboratories, Murray Hill, New Jersey 07974.

‡ Courant Institute, New York University, New York, New York 10012 and (present address) Department of Computer Science, School of Mathematical Sciences, Tel Aviv University, Tel Aviv 69978, Israel. The research of this author was supported by the U.S. Department of Energy under grant DE-AC02-76ER03077, by the National Science Foundation under grants NSF-MCS79-21258 and NSF-DCR-8318874, and by the U.S. Office of Naval Research under grant N0014-85-K-0046.

Tsin and Chin's algorithm [TC84] matches the bounds of our implementation 3. These algorithms use the CREW PRAM model, which is somewhat weaker than the CRCW PRAM model. However, Eckstein [Ec79a] and Vishkin [V83a] present general simulation methods that enable us to run implementation 2 on a CREW PRAM in $O(\log^2 n)$ time, without increasing the number of processors. On sparse graphs, the resulting algorithm uses fewer processors than either our implementation 3 or the algorithm of Tsin and Chin.

We achieve our improvements through two new ideas:

1. A block-finding algorithm that uses any spanning tree. The previously known linear-time algorithm for finding blocks uses a depth-first spanning tree [Ta72]. Depth-first search seems to be inherently serial; i.e. there is no apparent way to implement it in poly-log parallel time. The algorithm uses a reduction from the problem of computing biconnected components of the input graph to the problem of computing connected components of an auxiliary graph. This reduction can be computed efficiently enough both sequentially and in parallel that the running time of the fastest parallel connectivity algorithm becomes the only obstacle to a further improvement in implementation 2. (See the discussion in § 5.)

2. A novel algorithmic technique for parallel computations on trees is introduced. Given a tree, the technique uses an Euler tour of a graph obtained from the tree by adding a parallel edge for each edge of the tree. Therefore, we call is the *Euler tour technique on trees.* This technique allows the computations of various kinds of information about the tree structure in $O(\log n)$ time using $O(n)$ processors and $O(n)$ space on an exclusive-read exclusive-write parallel RAM. This model differs from the CREW PRAM in not allowing simultaneous reading from the same memory location. In the present paper we show how to use this Euler tour technique in order to compute preorder and postorder numbering of the vertices of a tree, number of descendants for all vertices, and other tree functions. Recently Vishkin [V84] proposed further extensions of this technique. (See § 5.) After the appearance of the first version of the present paper Awerbuch et al. [AIS84] and independently Atallah and Vishkin [AV84] essentially applied Euler tours on trees to finding Euler tours of general Eulerian graphs. See [AV84] for an explanation of this connection. The previously best known general technique for parallel computations on trees is the *centroid decomposition* method, which gives $O(\log^2 n)$-time algorithms. See [M83] for a discussion of this method and its use. The centroid decomposition method is the backbone of an earlier paper by Winograd [Wi75].

The idea of reducing the biconnectivity problem to a connectivity problem on an auxiliary graph was discovered independently by Tsin and Chin [TC84], who used the idea in their block-finding algorithm. However, their algorithm has two drawbacks:

(1) Their auxiliary graph contains many more edges than ours. This complicates the computation of the auxiliary graph and, more important, does not lead to a fast parallel algorithm using only a linear number of processors. One of the elegant features of our algorithm is that the same reduction is used in all three implementations.

(2) Their computation of preorder and postorder numbers and number of descendants in trees takes $O(\log n)$ time using $n^2/\log n$ processors—almost the square of the number of processors that we use.

The remainder of the paper consists of four sections. In § 2 we develop the block-finding algorithm and give a linear-time sequential implementation. In § 3 we describe our $O(\log n)$-time parallel implementation and present the Euler tour technique. Section 4 sketches our alternative parallel implementation. In § 5 we discuss variants of the algorithm for solving two additional problems: finding bridges and

directing the edges of a biconnected graph to make it strongly connected. We also discuss possible future work.

Note. If a parallel algorithm runs in $O(t)$ time using $O(p)$ processors then it also runs in $O(t)$ time using p processors. This is because we can always save a constant factor in the number of processors at the cost of the same constant factor in running time.

Historical remark. A variant of the block-finding algorithm presented here was first discovered by R. Tarjan in 1974 [T82]. U. Vishkin independently rediscovered a similar algorithm in 1983 and proposed a parallel implementation and the Euler tour technique [V83b]. Subsequent simplification by the two authors working together resulted in the present paper.

2. Finding blocks. Let $G=(V, E)$ be a connected undirected graph. Let R be the relation on the edges of G defined by $e_1 R e_2$ if and only if $e_1 = e_2$ or e_1 and e_2 are on a common simple cycle[1] of G. It is known that R is an equivalence relation [Ha69]. The subgraphs of G induced by the equivalence classes of R are the *blocks* (sometimes called *biconnected components*) of G. The vertices in two or more blocks are the *cut vertices* (sometimes called *articulation points*) of G; these are the vertices whose removal disconnects G. The edges in singleton equivalence classes are the *bridges* of G; these are the edges whose removal disconnects G. (See Fig. 1.)

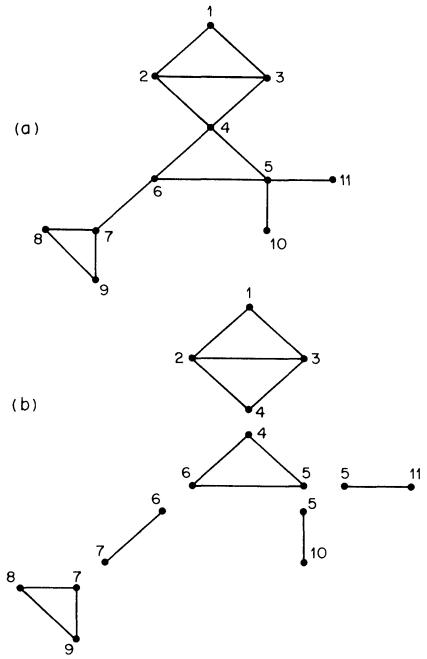

FIG. 1. (a) *An undirected graph.* (b) *Its blocks. Vertices* 4, 5, 6 *and* 7 *are cut vertices. Edges* {6, 7}, {5, 10}, *and* {5, 11} *are bridges.*

We can compute the equivalence classes of R, and thus the blocks of G, in $O(n+m)$ serial time using depth-first search [Ta72], where $n=|V|$ and $m=|E|$. Unfortunately, this algorithm seems to have no fast parallel implementation. In this

[1] In this paper a *cycle* is a path starting and ending at the same vertex and repeating no edge; a cycle is *simple* if it repeats no vertex except the first, which occurs exactly twice.

section we develop an $O(n+m)$-time serial algorithm that is well-suited for parallel implementation. The algorithm can use any spanning tree, not just a depth-first spanning tree.

We shall define an auxiliary graph G' of G whose connected components correspond to the blocks of G. The vertices of G' are the edges of G; if S is a set of edges in G, S induces a block of G if and only if S induces a connected component of G'. Let T be any rooted spanning tree of G. We shall denote the edges of T by $v \to w$, where v is the parent of w, denoted by $p(w)$. Let the vertices of T be numbered from 1 to n in preorder and identify each vertex by its number. G' contains each edge of G as a vertex and all edges of the following forms (see Fig. 2):

(i) $\{\{u, w\}, \{v, w\}\}$, where $u \to w$ is an edge of T and $\{v, w\}$ is an edge of $G - T$ such that $v < w$.

(ii) $\{\{u, v\}, \{x, w\}\}$, where $u \to v$ and $x \to w$ are edges of T and $\{v, w\}$ is an edge of $G - T$ such that v and w are unrelated in T.

(iii) $\{\{u, v\}, \{v, w\}\}$, where $u \to v$ and $v \to w$ are edges of T and some edge of G joins a descendant of w with a nondescendant of v.

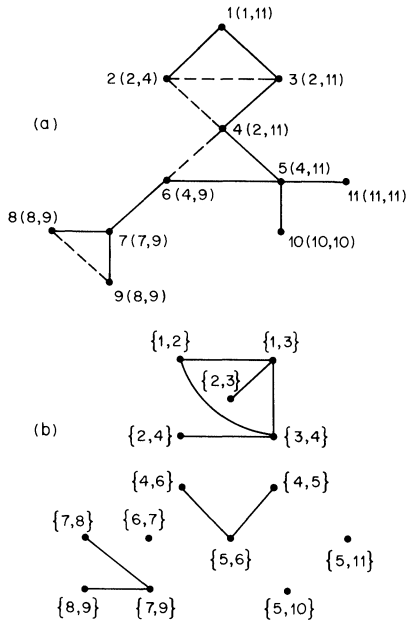

FIG. 2. (a) *A spanning tree of the graph in Fig.* 1. *Dashed edges are nontree edges. Vertices are numbered in preorder. Numbers in parentheses are the low and high number of each vertex.* (b) *The auxiliary graph* G'.

The intuition behind this construction is that every edge of $G - T$ defines a cycle consisting of this edge and the path in T joining its endpoints. All edges on this cycle are in the same biconnected component. We add enough edges to G' so that the vertices in G' corresponding to the edges on the cycle are in the same connected component.

THEOREM 1. *Two edges of* G *are in a common block of* G *if and only if as vertices of* G' *they are in a common connected component of* G'.

Proof. Any edge $\{x, y\}$ of $G - T$ defines a simple cycle of G, consisting of edge $\{x, y\}$ and the unique path in T joining x and y. These cycles are a cycle basis of G; the edge set of any cycle is the mod-two sum of the edge sets of appropriate basis cycles [Be73]. Define the relation R' by $e_1 R' e_2$ if and only if e_1 and e_2 are two edges of G on a common basis cycle, and let R'^* be the reflexive, transitive closure of R'.

We claim $R'^* = R$. Since R is an equivalence relation and $R' \subseteq R$, we have $R'^* \subseteq R$. To prove the converse, suppose $e_1 R e_2$. Then e_1 and e_2 are on a common simple cycle, which is a mod-two sum of basis cycles $C_1, C_2, \cdots, C_k$. Without loss of generality we can order $C_1, C_2, \cdots, C_k$ so that C_i for $i > 1$ has at least one edge in common with some C_j such that $j < i$. (Otherwise the mod-two sum of $C_1, C_2, \cdots, C_k$ would induce a disconnected subgraph.) It follows by induction on k that all edges in $C_1, C_2, \cdots, C_k$ are equivalent under R'^*, and in particular $e_1 R'^* e_2$. Thus $R \subseteq R'^*$.

Let $\{u, v\}$ and $\{x, w\}$ be adjacent in G'. If case (i) holds, $\{u, v\}$ is on the basis cycle defined by $\{x, w\}$. (In this case $x = v$.) If Case (ii) holds, $\{u, v\}$ and $\{x, w\}$ are on the basis cycle defined by $\{v, w\}$. If Case (iii) holds, say $\{y, z\}$ is an edge with y a descendant of w and z a nondescendant of $v = x$, then $\{u, v\}$ and $\{x, w\}$ are on the basis cycle defined by $\{y, z\}$. Thus in all cases $\{u, v\}$ and $\{x, w\}$ are in the same block of G.

Conversely, let $\{x, y\}$ be an edge of $G - T$ defining a basis cycle consisting of edge $\{x, y\}$, edges on the tree path from z to x, and edges on the tree path from z to y, where z is the nearest common ancestor of x and y. Without loss of generality suppose $x < y$. By Case (i), $\{x, y\}$ and $\{p(y), y\}$ are adjacent in G'. The existence of $\{x, y\}$ implies by Case (iii) that any two edges on the tree path from z to x are adjacent in G'. Similarly any two edges on the tree path from z to y are adjacent. If $z = x$, the tree path from z to x is empty. Otherwise (i.e. $z \neq x$), x and y are unrelated, and by Case (ii) $\{p(x), x\}$ and $\{p(y), y\}$ are adjacent in G'. Thus all edges on the basis cycle are in the same connected component of G'. The theorem follows.

Theorem 1 gives the following $O(n+m)$-time serial algorithm for finding blocks:

Step 1. Find a spanning tree T of G using any linear-time search method. Number the vertices of G from 1 to n in preorder and identify each vertex by its preorder number. Compute the number of descendants $nd(v)$ of each vertex v by processing the vertices in postorder using the recurrence $nd(v) = 1 + \sum \{nd(w) | v \to w \text{ in } T\}$. (We regard every vertex as a descendant of itself.) A vertex w is a descendant of another vertex v if and only if $v \leqq w \leqq v + nd(v) - 1$ [Ta74a].

Step 2. For each vertex v, compute $low(v)$, the lowest vertex that is either a descendant of v or adjacent to a descendant of v by an edge of $G - T$, and $high(v)$, the highest vertex that is either a descendant of v or adjacent to a descendant of v by an edge of $G - T$. The complete set of $2n$ *low* and *high* vertices can be computed in $O(n+m)$ time by processing the vertices of T in postorder using the following recurrences:

$low(v) = \min(\{v\} \cup \{low(w) | v \to w \text{ in } T\} \cup \{w | \{v, w\} \text{ in } G - T\})$;

$high(v) = \max(\{v\} \cup \{high(w) | v \to w \text{ in } T\} \cup \{w | \{v, w\} \text{ in } G - T\})$.

Step 3. Construct G'', the subgraph of G' induced by the edges of T, as follows. (The edges of G'' are those implied by cases (ii) and (iii).) For each edge $\{w, v\}$ in $G - T$ such that $v + nd(v) \leqq w$, add $\{\{p(v), v\}, \{p(w), w\}\}$ to G'' (Case (ii)). For each edge $v \to w$ of T such that $v \neq 1$ add $\{\{p(v), v\}, \{v, w\}\}$ to G'' if $low(w) < v$ or $high(w) \geqq v + nd(v)$ (Case (iii).)

Step 4. Find the connected components of G'' using any kind of linear-time search.

Step 5. Extend the equivalence relation on the edges of T (the vertices of G'') to the edges of $G - T$ by defining $\{v, w\}$ equivalent to $\{p(w), w\}$ for each edge $\{v, w\}$ of $G - T$ such that $v < w$ (Case (i).)

It is easy to implement this algorithm to run in $O(n+m)$ time using standard techniques. (See [Ta72]). If only a serial implementation is desired, the algorithm can be simplified somewhat (see [Ta82]); the algorithm as presented is designed for easy

parallel implementation. Note that each edge of $G-T$ is a vertex of degree one in G', and G'' contains $n-1$ vertices and at most $m-1$ edges.

Remark. Although we have assumed that G is connected, we can use the algorithm to find the blocks of a disconnected graph by applying it to each of the connected components (in series in the case of the implementation in this section, in parallel in the case of the implementations in §§ 3 or 4). This does not change the resource bounds of the algorithm.

3. A fast parallel implementation. In this section we describe how to implement the block-finding algorithm of § 2 to run in $O(\log n)$ time using $O(n+m)$ processors on a CRCW PRAM. We shall emphasize the ideas involved, only sketching the details. As the input representation, we assume that the vertex set is $V=\{1, 2, \cdots, n\}$ and that each undirected edge $\{i, j\}$ is represented by two directed edges (i, j) and (j, i). Each vertex i has a list of its outgoing edges: $adj(i)$ points to the first such edge and $next((i, j))$ points to the edge after (i, j) on i's list. (If there is no such edge, $next((i, j))=$ null.) Each edge (i, j) also has a pointer to its reversal (j, i). Each vertex i and each directed edge (i, j) has its own processor, denoted by $pr(i)$ and $pr(i, j)$, respectively.

Remark. This input representation is the most convenient one for our purposes, but it is not the only one that will work. For example, we can begin with an array of the $2m$ directed edges in arbitrary order and use the $O(\log m)$-time, $O(m)$-processor sorting algorithm of Ajtai, Komlós, and Szemerédi [AKS83] to sort the edges by first component. Once the edges are sorted, it is easy to construct incidence lists. Sorting the edges (i, j) lexicographically on $(\min\{i, j\}, \max\{i, j\})$ allows the construction of pointers between each edge and its reversal. Thus we obtain the desired input representation. While the asymptotic running time of this sorting algorithm is only $O(\log m)$, the constant factor is huge. Instead of this algorithm, we can use the randomized sorting algorithm of Reif and Valiant [RV83]. It will sort in time $O(\log m)$ almost surely using m processors. A third possibility is to perform this sorting in time $O(\log n)$ and m processors using an adaptation of the simple notion of "orthogonal trees". However, this takes $O(n^2)$ space. For more information on such sorting algorithms see Thompson [Th83].

Step 1. Construction of a spanning tree and computation of the preorder number and number of descendants of each vertex.

First we construct an unrooted spanning tree by using a modification of the Shiloach-Vishkin connected components algorithm [SV82]. We assume some familiarity with this algorithm. The algorithm maintains for each vertex v a pointer $D(v)$. Initially $D(v)=v$ for all vertices v. As the algorithm proceeds, the D-pointers are the parent pointers of a forest, each tree of which contains vertices known to be in a single connected component of the graph. (If v is the root of a tree in this *D-forest*, $D(v)=v$.) The D-pointers are changed by two kinds of steps:

Shortcutting. Replace $D(i)$ by $D(D(i))$ for some vertex i. Such a step changes the structure of the D-forest by moving v and its descendants closer to the root of its tree, but does not change the vertex partition defined by the D-trees.

Hooking. Replace $D(D(i))$ by $D(j)$, where $D(i)$ is the root of a D-tree, j is a vertex in another D-tree, and $\{i, j\}$ is an edge in the graph.

We modify the Shiloach-Vishkin algorithm so that all the edges are initially marked as nontree edges, and each time a hooking step is performed, the corresponding graph edge $\{i, j\}$ is marked as a tree edge. When the algorithm finishes, all the vertices are in a single D-tree, and the marked edges define a spanning tree. The original algorithm

runs in $O(\log n)$ time using $O(n+m)$ processors; these bounds are not affected by the modifications for computing a spanning tree.

One detail of this method deserves further discussion. Processors corresponding to several directed edges (i,j) may simultaneously try to write to the same location $D(D(i))$ to cause a hooking, but only one succeeds. In order to keep track of which one succeeds, we use an auxiliary array α. When a processor $pr((i,j))$ tries to cause a hooking step to take place, it first writes its name into $\alpha(D(i))$ by the assignment $\alpha(D(i)) \leftarrow pr((i,j))$. For a fixed value of $D(i)$, only one such processor succeeds. The successful processor $pr((i,j))$ then carries out the actual hooking step and marks both (i,j) and (j,i).

Remark. This idea for obtaining a spanning tree from a connected components computation has been used before. In particular Savage and Ja'Ja' [SJ81] used it to derive a minimum spanning forest algorithm from the connectivity algorithm of Hirschberg, Chandra and Sarwate [HCS79].

Having determined the edges of an unrooted spanning tree, we choose a root and number the vertices of the resulting rooted tree in preorder. To do this we first construct for each vertex i a list of the outgoing edges corresponding to tree edges. We can do this in $O(\log m) = O(\log n)$ time with $O(m)$ processors by using a standard "doubling" technique [Wy79]. For each (i,j), we initialize $treenext((i,j)) = next((i,j))$ and then repeat the following step, in parallel on all edges (i,j), $\lceil \log m \rceil$ times (until none of the *treenext* values change): if $treenext((i,j))$ is not null and not marked, replace $treenext((i,j))$ by $treenext(treenext((i,j)))$. Once all the *treenext* values are computed, we define $treeadj(i)$, for each vertex i, to be $adj(i)$ if $adj(i)$ is null or marked, $treenext(adj(i))$ otherwise. The *treeadj* and *treenext* maps define incidence lists for the spanning tree.

Next, we construct a circular list corresponding to an Eulerian tour of the directed version of the spanning tree. For each edge (i,j), the next edge $tournext((i,j))$ in the tour is $treenext((j,i))$ if $treenext((j,i))$ is not null, $treeadj(i,j)$ otherwise. This tour corresponds to the order of advancing and retreating along edges during a depth-first transversal of the tree, starting at an arbitrary vertex. To root the tree, we break the Eulerian tour at an arbitrary edge, causing some edge, say (i,j), to be the first edge on the list. Vertex i becomes the root of the tree. We call the broken list the *traversal list.* This traversal list is the backbone of the Euler tour technique that is introduced in this paper. In the sequel, we show that this list is the key to computing a number of tree functions.

We can number the edges of the traversal list from 1 to $2n-2$ in traversal order in $O(\log n)$ time with $O(n)$ processors by using the doubling technique to compute for each edge (i,j) the number of edges from (i,j) to the end of the list. We do this by initializing $numtoend((i,j)) = 1$ and $ptr((i,j)) =$ null for all $((i,j))$. Once this computation is complete, the number of edge (i,j) is $2n-1-numtoend((i,j))$.

Of two edges (i,j) and (j,i), the lower-numbered one corresponds to an advance from i to j along tree edge $\{i,j\}$ and the higher-numbered one to a retreat from j to i along $\{i,j\}$. Using the edge numbers, we can thus mark each directed edge as either an advance edge or a retreat edge. For each vertex j other than the root, there is exactly one advance edge (i,j); the parent $p(j)$ of j in the tree is i.

In the traversal list, the advance edges (i,j) occur in preorder on j. We can thus number the vertices in preorder using doubling, much as we computed the edge numbers. The only differences are that we initialize $numtoend(i,j)$ to be 1 if (i,j) is an advance edge, 0 otherwise, and when the computation is complete, if (i,j) is an

advance edge, we define $n+1-numtoend(i,j)$ to be the reorder number of vertex j. Once preorder numbers are computed, we replace each occurrence of a vertex by its preorder number, retaining an inverse map to restore the original vertex names when the computation is complete. (For each number i, we remember $vertex(i)$, the vertex with number i.)

Remark. Although not needed in this paper, a similar computation will number the vertices in postorder; for each vertex j other than the tree root, there is exactly one retreat edge (j, i), and the retreat edges appear in the transversal list in postorder on j.

The last part of Step 1 is the computation of the number of descendants $nd(j)$ of each vertex j. If j is not the tree root, $nd(j)$ is just the number of advance edges from $(p(j), j)$ to the end of the list (including $(p(j), j)$) minus the number of advance edges from $(j, p(j))$ to the end of the list. Two doubling computations, one of which we have already done to compute preorder numbers, and a parallel subtraction give the number of descendants of all the vertices.

Step 2. Computation of $low(j)$ and $high(j)$ for each vertex j.

We shall describe how to compute *low*; the computation of *high* is similar. Using doubling on the adjacency lists, we can compute $locallow(j) = \min(\{j\} \cup \{k | (j,k)$ is an unmarked (nontree) edge$\})$ for each vertex j in $O(\log n)$ time using $O(m)$ processors. Below we assume without loss of generality that n is a power of 2. We define an auxiliary value $globallow[i,j] = \min(\{locallow(k) | i \leqq k \leqq j\})$, i.e., $globallow[i,j]$ is the minimum of *locallow* over the interval $[i, i+1, \cdots, j]$. For each $0 \leqq \alpha \leqq \log n$ we compute *globallow* of the intervals $[(k-1)2^\alpha + 1, \cdots, k2^\alpha]$ for $1 \leqq k \leqq n/2^\alpha$. (The total number of such intervals is $O(n)$. They have the property that any interval $[i, \cdots, j]$, $1 \leqq i \leqq j \leqq n$, can be represented as a union of at most $2 \log n$ of them.)

```
Initialization. Assign globallow[i, i] ← locallow(i) for all 1 ≦ i ≦ n.
for α ← 1 to log n pardo
  for 0 ≦ k ≦ (n/2^α) − 1 do
  globallow[k2^α + 1, (k+1)2^α] ←
       min (globallow[k2^α + 1, (2k−1)2^(α−1)],
            globallow[(2k−1)2^(α−1) + 1, (k+1)2^α])
  end for
end for
```

This computation takes $O(\log n)$ time using n processors. (Actually, $n/\log n$ processors suffice but this is not important here.)

We compute $low(j)$ for each vertex j using the formula

$$low(j) = \min\{locallow(k) | j \leqq k \leqq j + nd(j) - 1\}.$$

That is, we compute $globallow[j, j+nd(j)-1]$, for each vertex j. The computation below uses the property that the interval $[j, \cdots, j+nd(j)-1]$ is a union of at most $2 \log n$ intervals on which *globallow* has already been computed. The variables $little(j)$ and $big(j)$ initially mark the endpoints of the interval. During the course of the computation the interval $[little(j), \cdots, big(j)]$ contains the subinterval of $[j, \cdots, j+nd(j)-1]$ that has not yet been taken into account in the computation of $low(j)$.

```
for 2≦j≦n pardo
  Initialize: little(j)←j; big(j)←j+nd(j)−1;
    low(j)←n+1 (Comment: This is a default value)
  for α←1 to log n do
    if little(j)−1 is not divisible by 2^α
    then low(j)←min (low(j), globallow[little(j), little(j)+2^(α−1)−1])
      little(j)←little(j)+2^(α−1)
    end if
    if big(j) is not divisible by 2^α
    then low(j)←min (low(j), globallow[big(j)−2^(α−1)+1, big(j)])
      big(j)←big(j)−2^(α−1)
    end if
    if little(j)>big(j)
    then Halt and output low(j)
    end if
  end for
end for
```

It is easy to verify the following. (1) All our requests for values of *globallow* are for intervals that have been previously computed. (2) The intervals that are taken into account in the computation of $low(j)$ actually cover the interval $[j, \cdots, j+nd-1]$. (3) The whole computation of Step 2 takes $O(\log n)$ time using $O(n)$ processors.

Step 3. Construction of the auxiliary graph G''.

This computation requires only $O(1)$ time using $O(m)$ processors, since testing the appropriate condition for each possible edge of G'' takes $O(1)$ time. After this test, which takes place in parallel, we have a set of at most $m-1$ processors, each of which knows an edge of G''.

Step 4. Finding the connected components of G''.

We apply the connected components algorithm of Shiloach and Vishkin. The information computed in Step 3 is sufficient as input to this algorithm, which takes $O(\log n)$ time and $O(n+m)$ processors. Once the algorithm finishes, each vertex (i, j) of G'' (advance edge of the spanning tree) has a D-pointer to a canonical "vertex" (x, y) representing the connected component containing (i, j).

Step 5. Extension of the equivalence relation found in Step 4 to the edges of $G-T$.

For each nontree edge (i, j) such that $i<j$, we assign $D((i, j))\leftarrow D((p(j), j))$. This takes $O(1)$ time and $O(m)$ processors.

This completes the computation except for restoring the original vertex names. An inspection of the various steps shows that none uses more than $O(\log m)=O(\log n)$ time, more than $O(n+m)$ space, or more than $O(n+m)$ processors. The only place concurrent writing is used is in the connected components algorithm, used in Steps 1 and 4.

4. An alternative parallel implementation. In this section we develop an implementation of the block-finding algorithm that runs in $O(\log^2 n)$ time using $O(n^2/\log^2 n)$ processors on a CREW PRAM, assuming that the input graph is represented by an adjacency matrix. Since we can always trade time for processors, this method gives an $O(n^2/p)$ time algorithm using p processors, for any $p\leqq n^2/\log^2 n$. This algorithm has optimal speed-up, assuming an adjacency matrix representation of the input. We shall not go through the details of the implementation but merely mention where it differs from the $O(\log n)$-time implementation of the previous section.

There are two known connected components algorithms that run in $O(\log^2 n)$ time using $O(n^2/\log^2 n)$ processors: the algorithm of Vishkin [V81], which runs on a

CRCW PRAM, and the algorithm of Chin, Lam, and Chen [CLC81], which runs on a CREW PRAM. Although the latter is more complicated, we shall use it instead of the former in Steps 1 and 4, since it uses a less powerful computation model. Chin, Lam, and Chen describe how to adapt their algorithm to compute a (minimum) spanning forest.

Step 1. Construction of a spanning tree and computation of the preorder number and number of descendants of each vertex.

We apply the algorithm of Chin, Lam, and Chen to mark the entries in the adjacency matrix corresponding to tree edges. We can convert each row of the adjacency matrix to an incidence list for the corresponding vertex (of edges incident in the spanning tree) by using a balanced binary tree with n leaves to guide the computation. (For each marked entry, we need to compute the next marked entry in the row.) The computation is similar to a standard partial-sum computation and takes $O(\log^2 n)$ time with $O(n/\log^2 n)$ processors (see for instance [V81]). Since we can carry out the computation for all rows in parallel, the total time is $O(\log^2 n)$ with $O(n^2/\log^2 n)$ processors. Establishing pointers between each directed edge (i, j) and its reverse is easy. Now we have the representation of the unrooted spanning tree used in § 3. The remainder of the Step 1 computation proceeds as in § 3, taking $O(\log n)$ time on $O(n)$ processors.

Step 2. Computation of *low* and *high*.

Computing *locallow*(j) requires n parallel minimum computations. Each takes $O(\log^2 n)$ time using $O(n/\log^2 n)$ processors [Wy79], a total of $O(n^2/\log^2 n)$ processors. The remainder of the *low* computation proceeds as in § 3 taking $O(\log n)$ time using $O(n)$ processors. The computation of *high* is similar.

Step 3. Constuction of the auxiliary graph G''.

This is easy in $O(\log^2 n)$ time with $O(n^2/\log^2 n)$ processors.

Step 4. Finding the connected components of G''.

Step 5. Extension of the equivalence relation found in Step 4 to the edges of $G - T$.

This is easy in $O(\log^2)$ time with $O(n^2/\log^2 n)$ processors.

5. Extensions and future work. There are two related problems that can be solved using variants of our algorithm, in the same resource bounds. Neither of these requires the second connected-component-finding step (Step 4). The first is the problem of finding all bridges of a graph. The bridges are just the one-edge biconnected components. Thus we can use the biconnected components algorithm directly. However, there is a simpler algorithm. Suppose we number the vertices in preorder with respect to any spanning tree; identify vertices by number; compute $nd(v)$, the number of descendants of the vertex, for each vertex v; and compute the *low* and *high* functions defined in § 2. A tree edge $v \to w$ with v the parent of w is a bridge if and only if $w \leqq low(w)$ and $high(w) \leqq w + nd(w) - 1$, i.e. if and only if both $low(w)$ and $high(w)$ are descendants of w [Ta74b], [TC83]. No nontree edge is a bridge. By applying this test, we can find all bridges in $O(\log n)$ time and $O(n+m)$ processors on a CRCW PRAM using the algorithm of § 2, or in $O(n^2/p)$ time using any number $p \leqq n^2/\log^2 n$ of processors on a CREW PRAM using the algorithm of § 3. The latter bounds for bridge-finding were first obtained by Tsin and Chin [TC83] using this approach; the former bounds are new.

The second problem is that of directing the edges of a bridgeless graph so that the resulting directed graph is strongly connected. Atallah [At84] proposed an algorithm for this problem that runs in $O(\log n)$ time using $O(n^3)$ processors on a CRCW PRAM. Vishkin [V84] gave an algorithm with the same resource bounds as our method for finding bridges and biconnected components. We shall propose an alternative, simpler

algorithm. As above, assume that we have found a spanning tree, numbered the vertices in preorder, identified each vertex by its number, and computed $low(v)$ for each vertex v. Let $\{v, w\}$ be a nontree edge. We call $\{v, w\}$ a *back edge* if v and w are related in the tree and a *cross edge* otherwise. We can determine in $O(1)$ time and $O(m)$ processors the cross edges and back edges, since a nontree edge $\{v, w\}$ with $v < w$ is a back edge if and only if w is a descendant of v, i.e. $w \leqq v + nd(v) - 1$. We define $lowback(v)$ analogously to $low(v)$ but using only back edges, as follows:

$$lowback(v) = \min(\{v\} \cup \{lowback(v) | v \to w \text{ in } T\} \cup \{w | \{v, w\} \text{ is a backedge}\}).$$

We can compute *lowback* just as we computed *low*, in the same resource bounds.

Now suppose G has no bridges. To convert G to a strongly connected directed graph, we direct the edges as follows:

(i) If $\{v, w\}$ is a back edge with $v < w$, direct $\{v, w\}$ from w to v.

(ii) If $\{v, w\}$ is a cross edge with $v < w$, direct $\{v, w\}$ from v to w.

(iii) If $\{v, w\}$ is a tree edge with $v < w$, direct $\{v, w\}$ from v to w if $lowback(w) < w$ or if $low(w) \geqq w$, and from w to v otherwise.

The intuition behind this construction is to direct back edges from descendant to ancestor and tree edges from parent to child. This suffices if there are no cross edges (i.e. the tree is a depth-first spanning tree). To handle the cross edges we direct them from lower to higher endpoint and reverse the natural directions of some of the tree edges as described in (iii).

THEOREM 2. *The directed graph formed by applying rules* (i), (ii), *and* (iii) *is strongly connected.*

Proof. We must show that every vertex is reachable from vertex 1 (the tree root) and vertex 1 is reachable from every vertex. We show that every vertex v is reachable from vertex 1 by induction on the preorder number of v. Obviously vertex 1 is reachable from itself. Suppose vertices $1, 2, \cdots, v-1$ are reachable from vertex 1 and consider vertex v. There is some tree edge $\{u, v\}$ with $u < v$. If this edge is directed from u to v, then v is reachable from vertex 1. Otherwise, by rule (iii), $low(v) < v$ and $lowback(v) \geqq v$. This means that there is a cross edge directed from $low(v)$ to some descendant of v, say x.

Vertex x is reachable from $low(v)$ and hence from 1 by the induction hypothesis. Furthermore, v is reachable from x by a directed path consisting of tree edges and back edges. Otherwise, let $y \neq v$ be the lowest ancester of x and descendant of v reachable from v by such path. We know $low(x) \leqq low(v) < v \leqq x$. By rule (iii), it must be the case that $lowback(y) < y$; otherwise the tree edge $\{p(y), y\}$ is directed from y to $p(y)$, contradicting the choice of y. But this implies, also by rule (iii), that there is a directed cycle containing y and $p(y)$ consisting of a back edge from a descendant of y to $lowback(y)$ and all tree edges on the tree path between these vertices. This also contradicts the choice of y. We conclude that v is indeed reachable from x, and hence from 1. By induction all vertices are reachable from 1.

It remains for us to show that vertex 1 is reachable from every vertex. This will follow if we can prove that from any vertex $v \neq 1$ we can reach a vertex larger in postorder. If $lowback(v) < v$, there is a directed path from v to $lowback(v)$ by rule (iii). If $lowback(v) \geqq v$ but $low(v) < v$, there is a directed edge from v to its parent. The only remaining possibility is $low(v) \geqq v$. In this case $high(v) > v + nd(v) - 1$, i.e. $high(v)$ is not a descendant of v, for otherwise the tree edge $\{p(v), v\}$ would be a bridge. Let $\{x, high(v)\}$ be an edge connecting a descendant x of v to $high(v)$. This edge must be a cross edge, directed from x to $high(v)$. We claim that every descendant of v, including x, is reachable from v (by a directed path containing only descendants of v). This

follows from the fact that $low(v) \leqq v$ using an argument like that used to prove that every vertex is reachable from vertex 1. Hence $high(v)$ is reachable from v. In all cases a vertex larger than v in postorder is reachable from v, and it follows that vertex 1 is reachable from every vertex. □

Directing the edges according to rules (i)-(iii) takes $O(1)$ time using $O(m)$ processors once the vertices are numbered in preorder and *low* and *lowback* are computed.

We close this section and the paper with a few remarks about future work. The parallel tree computations we have used may have applications to other graph problems. This deserves study. Also, there are still open problems concerning parallel biconnectivity algorithms. The algorithm of this section, as does the algorithm of Tsin and Chin [TC84], has optimal speed-up for dense graphs but not for sparse ones, whereas the algorithm of § 3 is off by a factor of $\log n$ from optimal speed-up. A question worth exploring is whether there is an $O((n+m)/p)$-time algorithm using p processors, for p sufficiently small (say $p \leqq (n+m)/\log^2 n$ or $p \leqq (n+m)/\log n$.) Such an algorithm is unknown even for the problem of computing connected components.

Suppose that an algorithm of time $O((n+m)/p)$ could be found for the problem of computing connected components. Then the implementation of § 3 implies a block-finding algorithm of time $O((n \log n + m)/p)$ using $p \leqq n \log n + m$ processors, provided we are given a proper input representation. In order to see this, consider the following representation of the input graph for the block-finding problem. The vertex set is $V = \{1, 2, \cdots, n\}$. Each edge $\{i, j\}$ is represented by two directed edges (i, j) and (j, i). The $2m$ directed edges of the graph appear in ascending lexicographic order in a vector of length $2m$. (That is, $(i_1, j_1) < (i_2, j_2)$ if $i_1 < i_2$ or $i_1 = i_2$ and $j_1 < j_2$.) Each vertex i has a pointer to its first outgoing edge. The implementation of § 3 still requires the following modification. Recall the construction of the list of outgoing edges in the tree for every vertex. This was done using doubling, which required $O(\log n)$ time using only $O(m/\log m)$ processors. Instead, we construct a sorted vector (similar to the input vector) of length $2n-2$ that contains all directed edges of the tree. This takes time $O(\log n)$ using $O(m)$ processors: For each directed edge in the tree we need to find its serial number relative to the other directed edge of the tree. We use a balanced binary tree with $2m$ leaves, one for each input directed edge, to guide the computation, which is a standard partial sum computation where each active leaf enters one and gets in return its serial number relative to other active leaves. This is similar to the computation following Step 1 of this section. A similar remark applies to the computation of *locallow*(j) (just before the construction of the tree).

REFERENCES

[AIS84] B. AWERBUCH, A. ISRAELI AND Y. SHILOACH, *Finding Euler circuits in logarithmic parallel time*, Proc. Sixteenth ACM Symposium on Theory of Computing, 1984, pp. 249-257.

[AKS83] M. AJTAI, K. KOMLÓS, AND E. SZEMERÉDI, *An $O(n \log n)$ sorting network*, Proc. Fifteenth ACM Symposium on Theory of Computing, 1983, pp. 1-9.

[At84] M. J. ATALLAH, *Parallel strong orientation of an undirected graph*, Inform. Proc. Letters, 18 (1984), pp. 37-39.

[AV84] M. ATALLAH AND U. VISHKIN, *Finding Euler tours in parallel*, J. Comp. Sys. Sci., to appear.

[Be73] C. BERGE, *Graphs and Hypergraphs*, North-Holland, Amsterdam, 1973.

[BH82] A. BORODIN AND J. E. HOPCROFT, *Routing, merging and sorting on parallel models of computation*, Proc. Fourteenth ACM Symposium on Theory of Computing, 1982, pp. 334-338.

[CLC81] F. Y. CHIN, J. LAM, AND I. CHEN, *Optimal parallel algorithms for the connected component problem*, Proc. 1981 International Conference on Parallel Processing, 1981, pp. 170-175.

[Ec79a] D. M. ECKSTEIN, *Simultaneous memory access*, Technical Report TR-79-6, Computer Science Department, Iowa State Univ., Ames, Iowa, 1979.

[Ec79b] ———, BFS *and biconnectivity*, Technical Report TR-79-11, Computer Science Department, Iowa State Univ., Ames, Iowa, 1979.

[Ha69] F. HARARY, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.

[HCS79] D. S. HIRSCHBERG, A. J. CHANDRA, AND D. V. SARWATE, *Computing connected components on parallel computers*, Comm. ACM, 22 (1979), pp. 461–464.

[M83] N. MEGIDDO, *Applying parallel computation algorithms in the design of serial algorithms*, J. Assoc. Comput. Mach., 1983, pp. 852–865.

[RV83] J. REIF AND L. J. VALIANT, *A logarithmic time sort for linear size networks*, Proc. Fifteenth ACM Symposium on Theory of Computing, 1983, pp. 10–16.

[SJ81] C. SAVAGE AND J. JA'JA', *Fast, efficient parallel algorithms for some graph problems*, this Journal, 10 (1981), pp. 682–691.

[SV81] Y. SHILOACH AND U. VISHKIN, *Finding the maximum, merging, and sorting in a parallel computation model*, J. Algorithms, 2 (1981), pp. 88–102.

[SV82] ———, *An* $O(\log n)$ *parallel connectivity algorithm*, J. Algorithms, 3 (1982), pp. 57–63.

[TA72] T. E. TARJAN, *Depth-first search and linear graph algorithms*, this Journal, 1 (1972), pp. 146–160.

[Ta74a] ———, *Finding dominators in directed graphs*, this Journal, 3 (1974), pp. 62–89.

[Ta74b] ———, *A note on finding the bridges of a graph*, Inform. Proc. Letters, 2 (1974), pp. 160–161.

[Ta82] ———, *Graph partitions defined by simple cycles*, Technical Memorandum, AT&T Bell Laboratories, Murray Hill, NJ, 1982.

[TC83] Y. H. TSIN AND F. Y. CHIN, *A general program scheme for finding bridges*, Inform. Proc. Letters, 17 (1983), pp. 269–272.

[TC84] ———, *Efficient parallel algorithms for a class of graph theoretic problems*, this Journal, 13 (1984), pp. 580–599.

[Th83] C. D. THOMPSON, *The* VLSI *complexity of sorting*, IEEE Trans. Comput., C-32 (1983), pp. 1171–1184.

[Ts82] Y. H. TSIN, *A generalization of Tarjan's depth first search algorithm for the biconnectivity problem*, Dept. Computing Science, Univ. Alberta, Edmonton, Alberta, Canada, 1982.

[TV84] R. E. TARJAN AND U. VISHKIN, *Finding biconnected components and computing tree functions in logarithmic parallel time*, Proc. 25th Annual IEEE Symposium on Foundations of Computing, 1984, pp. 12–20.

[V81] U. VISHKIN, *An optimal parallel connectivity algorithm*, Technical Report RC 9149, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1981; Disc. Appl. Math., to appear.

[V83a] ———, *Implementation of simultaneous memory address access in models that forbid it*, J. Algorithms, 4 (1983), pp. 45–50.

[V83b] ———, $O(\log n)$ *and optimal parallel biconnectivity algorithms*, Technical Report # 69, Computer Science Dept., New York Univ., New York, 1983.

[V83c] ———, *Synchronous parallel computation—a survey*, Technical Report # 71, Computer Science Dept., New York Univ., New York, 1983.

[V84] ———, *An efficient parallel strong orientation*, Technical Report # 109, Computer Science Dept., New York Univ., New York, 1984.

[Wi75] S. WINOGRAD, *On the evaluation of certain arithmetic expressions*, J. Assoc. Comput. Mach., 22 (1975), pp. 477–492.

[Wy79] J. C. WYLLIE, *The complexity of parallel computation*, Technical Report TR 79-387, Dept. Computer Science, Cornell Univ., Ithaca, NY, 1979.