FINDING BICONNECTED COMPONEMTS AND COMPUTING TREE FUNCTIONS IN LOGARITHMIC PARALLEL TIME
Extended Summary

Robert E. Tarjan* - Uzi Vishkin**

* AT&T Bell Laboratories, Murray Hill, NJ 07974.
** Courant Institute, New York University and (present address) Department of Computer Science, Tel Aviv University, Tel Aviv 69 978, Israel.

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 $p \leq n^2/\log^2 n$ of processors, on a concurrent-read, exclusive-write parallel RAM. The latter algorithm has optimal speedup, assuming an adjacency matrix representation of the input.

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

Keywords: 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 [SV 82], is a member of a family of models for parallel computation. (See [BH 82], [SV 81], [V 83c].)

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 $p \leq n^2/\log^2 n$ 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.

We achieve our results 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 [Ta 72]. 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 so efficiently both sequentially and in both parallel implementations that the efficiencies (time and number of processors) of parallel connectivity algorithms become the only obstacle to a further improvement in implementation 2. This is interesting since intuitively the connectivity problem seems easier than the biconnectivity problem.

2. A novel algorithmic technique for parallel algorithms on trees. Given a tree, the technique uses an Euler tour of a graph obtained from a tree by adding a parallel edge for each edge of the tree. Therefore, we call it the Euler tour technique on trees. This technique is very powerful. It allows the computation 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 paper we show how to use this Euler tour technique in order to compute preorder and postorder numbering of the vertices of a tree,

The research of the second author was supported by DOE grant DE-AC02-76ER03077 and by NSF grant NSF-MCS79-21258.

number of descendents for all vertices and a number of other tree functions. An elegant feature of our paper is that these computations are all minor variations of the same technique. The previously best known general algorithmic technique for trees implies $O(\log^2 n)$ time algorithms and is known by the name centroid decomposition. See [M 83] for an example where the later technique is applied and discussed. It is an interesting exercise to observe that the centroid decomposition is the backbone in an earlier paper by Winograd [Wi 75].

Implementation 2 is faster than any of the previously known parallel algorithms [SJ 81], [Ec 79b], [TC 84]. Eckstein's algorithm [Ec 79b] 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' [SJ 81] 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. Tsin and Chin's algorithm [TC 84] matches the bounds of our implementation 3. These algorithms use the CREW PRAM model, which is somewehat weaker than the CRCW PRAM model. However, Eckstein [Ec 79a] and Vishkin [V 83a] 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.

Each of our implementations readily implies an algorithm for computing bridges in the same time and number of processors. This improves on the bridge-finding algorithm of Savage and Ja'Ja' [SJ 81], which runs in $O(\log^2 n)$ time using $O(n^2 \log n)$ processors. Tsin and Chin's algorithm for bridges matches the bounds of our implementation 3.

The idea of reducing the biconnectivity problem to a connectivity problem on an auxiliary graph was discovered independently by Tsin [Ts 82]. It is used for Tsin and Chin's algorithm that matches the bounds of our implementation 3. This, again, was discovered indepently. However, there are two substantial differences between Tsin and Chin's solution and ours.

(1) The auxiliary graph in which connectivity has to be computed has many more edges than the auxiliary graph we use. This causes the following problems. It complicates the computation of the auxiliary graph and, more important, does not permit a fast parallel algorithm using only a linear number of processors. An elegant feature of our algorithm is that the same reduction is used in all three implementations.

(2) Their computation of preorder, postorder and number of descendents on trees takes $O(\log n)$ time using $n^2/\log n$ processors - almost the square of the number of processors used here.

The remainder of the paper consists of four sections. In Section 2 we develop the block-finding algorithm and give a linear-time sequential implementation. In Section 3 we describe our $O(\log n)$-time parallel implementation and present the Euler tour technique. Section 4 sketches our alternative parallel implementation. Section 5 concludes by reviewing applications of the Euler tour technique and suggesting some 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. Our stated complexity bounds take advantage of this observation.

Historical Remark. A variant of the block-finding algorithm presented here was first discovered by R. Tarjan in 1974 [Ta 82]. U. Vishkin independently rediscovered a similar algorithm in 1983 and proposed parallel implementations and the Euler tour technique [V 83b]. Subsequent simplification by the two authors working together resulted in [TV 83]. Recently, [V 84] proposed further amplifications to this technique. Parts from these two papers are represented in the present summary.

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* of G. It is known that R is an equivalence relation [Ha 69]. 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 (someties 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 Figure 1.)

[Figure 1]

We can compute the equivalence classes of R, and thus the blocks of G, in $O(n+m)$ serial time using depth-first search [Ta 72], where $n = |V|$ and $m = |E|$. Unfortunately, this algorithm seems to have no fast parallel implementation. In this section we develop an $O(n+m)$-time serial algorithm that is suited for both our parallel implementations. The algorithm can use any spanning tree, rather than 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

*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.

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 Figure 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.

A formal justification for the definition of G′ is given is Theorem 1 below. Let us give first some intuition for this definition. Consider the problem of classifying only the edges of T into biconnected components of G. Let G" be the subgraph of G′ which is induced by vertices that correspond to edges of T only. The main step of the algorithm computes the connected components of G". Here, we explain only this step. Consider an edge $\{u,v\}$ of G-T. $\{u,v\}$ implies that all edges in T on the path from u to v are in the same biconnected component of G. If u is an ancestor of G then (iii) (in the definition of G′) yields connectivity of these edges. If u and v are unrelated in T then (iii) yields connectivity within two sets of edges in T: (1) edges of the path from u to the lowest common ancestor of u and v and (2) edges of the path from v to the lowest common ancestor of u and v. Finally, (ii) yields connectivity between these two sets.

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 [Be 73]. 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 $C_1, C_2, \ldots, C_k$. Without loss of generality we can order $C_1, C_2, \ldots, C_k$ 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 $C_1, C_2, \ldots, C_k$ would induce a disconnected subgraph.) It follows by induction on k that all edges in $C_1, C_2, \ldots, C_k$ 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 $nd(v) = 1 + \Sigma \{nd(w) \mid v \to w \text{ in } T\}$. (We regard every vertex as a descendant of itself.) A vertex w is a descendant of another vertex v if and only if $v \leq w \leq v + nd(v) - 1$ [Ta 74].

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:

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

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

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 $v + nd(v) \leq w$, 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 $high(w) \geq v + nd(v)$ (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 [Ta 72].). If only a serial implementation is desired, the algorithm can be simplified somewhat. (See [Ta 82].) 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 Section 3 or 4). This does not change the resource bounds of the algorithm.

3. Fast Parallel Implementation

In this section we describe how to implement the block-finding algorithm of Section 2 to run in O(log n) time with 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,...,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 [AKS 83] 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) it should be noted that there is a large constant in front of the log m. Instead of this algorithm we can use the randomized sorting algorithm of Reif and Valiant [RV 83]. 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 needs $O(n^2)$ space. For more information on such sorting algorithms see Thompson [Th 83].

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 [SV 82]. 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 non-tree 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 α(D(i)) by the assignment α(D(i)) ← 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' [SJ 81] used it to derive a minimum spanning forest algorithm from the connectivity algorithm of Hirschberg, Chandra and Sarwate [HCS 79].

Having determined the edges of an unrooted spanning tree, we must determine a root and number the vertices in preorder. First, we 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 "doubling" technique [Wy 79]. For each edge (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))). This takes O(log m) iterations over the edges. 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(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 Euterian 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 quite a number of functions on the tree.

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 preorder 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 traversal 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} {k|(j,k) is an unmarked (nontree) edge}) for each vertex j in O(log n) time using O(m) processors. Below, we assume, w.l.g., that n is a power of 2. We define an auxiliary value globallow[i,j] = min({ locallow(k)| i ≤ k ≤ j}, i.e., globallow[i,j] is the minimum of locallow over the interval [i,i+1,...,j]. For each $0 \leq \alpha \leq$ log n we compute globallow of the intervals $[(k-1)2^{\alpha}+1,\ldots,k2^{\alpha}]$ for $1 \leq k \leq n/2^{\alpha}$. (The total number of such intervals is O(n). They have the property that any interval [i,...,j] , $1 \leq i \leq j \leq n$, can be represented as a union of at most 2log n of them.)

Initialization. Assign globallow[i,i] ← locallow(i) for all $1 \leq i \leq n$.

```
for α ← 1 to log n pardo
  for each 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])
```

This computation takes O(log n) time using n processors. (Actually, n/log n processors suffice but we shall not discuss it here).

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

low(j) = min{ locallow(k)| j ≤k≤ 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,...,j+nd(j)-1] is a union of at most 2log n intervals on which globallow has already been computed. The variables little(j) and big(j) intially mark the endpoints of the interval. During the course of the computation the interval [little(j),...,big(j)] contains the subinterval of [j,...,j+nd(j)-1] that has not yet been taken into account in the computation of low(j).

```
for all 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)+$2^{\alpha-1}$-1])
little(j) ← little(j) + $2^{\alpha-1}$
if big(j) is not divisible by 2^{α}
then low(j) ← min(low(j),globallow[big(j) - $2^{\alpha-1}$ + 1,big(j)])
big(j) ← big(j) - $2^{\alpha-1}$
if little(j) > big(j)
then Halt and output low(j)

It is easy to verify the following. (1) All our requests for values of globallow were for intervals that have been computed before. (2) The intervals that are taken into account in the computation of low(j) really "cover" the interval [j,...,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 apropriate 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 non-tree edge (i,j) such that i < j, we assign D((i,j)) ← 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 $p \leq n^2/\log^2 n$. 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 [V 81], which runs on a CRCW PRAM, and the algorithm of Chin, Lam, and Chen [CLC 81], 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 [V 81]). 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 Section 3. The remainder of the Step 1 computation proceeds as in Section 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 [Wy 79], a total of $O(n^2/\log^2 n)$ processors. The remainder of the low computation proceeds as in Section 3 taking O(log n) time using O(n) processes. The computation of high is similar.

Step 3. Construction 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. Conclusion

5.1 The Euler tour technique for trees revisited

We presented the Euler tour technique for trees earlier in the paper. A non-trivial contribution of this paper is the wide applicability of this technique. Let T be an

undirected tree having n vertices. In this concluding section we mention a few functions on T that the Euler tour technique can be applied for their computation. Thereby, we support our claim that this technique is powerful. All algorithms mentioned in this section run in O(log n) time using O(n) space and n processors on the EREW PRAM model of computation.

Function 1. Compute H, a directed version of T which is rooted at some vertex r.

The algorithm was given in Section 3.

Whenever we refer in this paper to the Euler tour technique on trees we refer to utilizations of the traversal list given in Section 3. For better understanding of the amount of information hidden in this Euler path it is helpful to think about each directed edge f of H as a left parenthesis and its anti-parallel edge as a right parenthesis. The Euler path will then correspond to a legal sequence of parentheses, where matching pairs of parentheses will represent the two copies of an edge of T.

Function 2. Compute preorder numbering of the vertices of H.

Algorithm: See Section 3.

Function 3. Compute postorder numbering of the vertices of H.

Algorithm: See Section 3.

Function 4. Compute levels of the vertices of H. That is, for each vertex in H find the length of the path from r to it.

Finding this algorithm is simple. It is left to the reader.

Function 4. Number of descendents of the vertices in H.

Algorithm: See Section 3.

Function 5. Lowest common ancestor (LCA) of two vertices in H. We used the Euler path to form a data-structure which enables retrieval of the LCA of any pair of vertices in O(log n) time by a single processor. See [V 84].

Function 6. low(v) for every vertex v in G. Where low(v) is defined as the the minimum preorder number over: v, descendents of v and vertices adjacent to a descendent of v by an edge of G-T.

Algorithm: See Section 3.

Function 7. high(v) for every vertex v in G. Where high(v) is defined as the the maximum preorder number over: v, descendents of v and vertices adjacent to a descendent of v by an edge of G-T.

Algorithm: See Section 3. (Recall that the computation of the last two functions plays an important role in our biconnectivity algorithm).

[AIS 84] and [AV 83] gave (independently from each other) algorithms for finding Euler tours in general Euler graphs. The present paper preceded both [AIS 84] and [AV 83] and is more fundamental than them. While there is no apparent way in which the present paper can benefit from these papers, [AV 83] indicates how to apply ideas of the present paper for substantial simplification of the algorithm for finding Euler tours (with respect to the algorithm of [AIS 84])

5.2 Future work

We close this section and the paper with a few remarks about future work. The parallel tree computations used in Section 3 may have applications in other graph algorithms. This deserves study. Also, there are still open problems concerning parallel biconnectivity algorithms. The algorithm of Section 4, as does the algorithm of Tsin and Chin [TC 84], has optimal speed-up for dense graphs but not for sparse ones, whereas the algorithm of Section 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 $p \leq (n+m)/\log^2 n$ or $p \leq (n+m)/\log n$.) 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 Section 3 implies a block-finding algorithm of time $O((n\log n + m)/p)$ using $p \leq n\log n + m$ 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 $V = \{1,2,\ldots,n\}$. Each edge $\{i,j\}$ is represented by two directed edges (i,j) and (j,i). The 2m directed edges of the graph appear in an 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 Section 3 still requires the following modifiction. 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 which contains all directed edges of the tree in 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 the previous section. A similar remark applies to the computation of locallow(j) (just before the construction of the tree).

REFERENCES

[AIS 84] B. Awerbuch, A. Israeli and Y. Shiloach, "Finding Euler circuits in logarithmic parallel time", Proc. Sixteenth ACM Symp. on Theory of Computing, 249-257.

[AKS 83] M. Ajtai, J. Komlós, and E. Szemerédi, "An O(n log n) sorting network," Proc. Fifteenth ACM Symp. on Theory of Computing (1983), 1-9.

[AV 84] M. Atallah and U. Vishkin, "Finding Euler tours in parallel", preprint. To appear in JCSS.

[Be 73] C. Berge, Graphs and Hypergraphs, North-Holland, Amsterdam, 1973.

[BH 82] A. Borodin and J.E. Hopcroft, "Routing, merging and sorting on parallel models of computation," Proc. Fourteenth ACM Symp. on Theory of Computing (1982), 338-334.

[CLC 81] F.Y. Chin, J. Lam, and I. Chen, "Optimal parallel algorithms for the connected component problems," Proc. 1981 International Conf. on Parallel Processing (1981), 170-175.

[Ec 79a] D.M. Eckstein, "Simultaneous memory access," Technical Report TR-79-6, Computer Science Department, Iowa State University, Ames, Iowa, 1979.

[Ec 79b] D.M. Eckstein, "BFS and biconnectivity," Technical Report TR-79-11, Computer Science Department, Iowa State University, Ames, Iowa, 1979.

[Ec 79b] F. Harary, Graph Theory, Addison Wesley, Reading, Mass., 1969.

[HCS 79] D.S. Hirschberg, A.K. Chandra, and D.V. Sarwate, "Computing connected components on parallel computers," Comm. ACM 22 (1979).

[M 83] N. Megiddo, "Applying parallel computation algorithms in the design of serial algorithms", JACM 30,4(1983), 852-865.

[RV 83] 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.

[SJ 81] C. Savage and J. Ja'Ja', "Fast, efficient parallel algorithms for some graph problems," SIAM J. Comput. 10 (1981), 682-691.

[SV 81] Y. Shiloach and U. Vishkin, "Finding the maximum, merging and sorting in a parallel computation model," J. Algorithms 2 (1981), 88-102.

[SV 82] Y. Shiloach and U. Vishkin, "An O(log n) parallel connectivity algorithm," J. Algorithms 3 (1982), 57-63.

[Ta 72] R.E. Tarjan, "Depth-first search and linear graph algorithms," SIAM J. Comput. 1 (1972), 146-160.

[Ta 74] R.E. Tarjan, "Finding dominators in directed graphs," SIAM J. Comput. 3 (1974), 62-89.

[Ta 82] R.E. Tarjan, "Graph partitions defined by simple cycles," Technical Memorandum, Bell Laboratories, Murray Hill, New Jersey, 1982.

[TC 84] Y.H. Tsin and F.Y. Chin, "Efficient parallel algorithms for a class of graph theoretic problems," SIAM J. Comput. 13(1984), 580-599.

[Th 83] C.D. Thompson, "The VLSI complexity of sorting", IEEE Trans. Comput., (December 1983).

[Ts 82] Y.H. Tsin, "A generalization of Tarjan's depth first search algorithm for the biconnectivity problem," Dept. of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1982.

[TV 83] R.E. Tarjan and U. Vishkin, "An efficient parallel biconnectivity algorithm," Technical Report #69(revised), Computer Science Department, New York University, New York, New York, 1983. To appear in SIAM J. Comp.

[V 81] U. Vishkin, "An optimal parallel connectivity algorithm," Technical Report RC 9149, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, 1981. To appear in Discrete Applied Mathematics.

[V 83a] U. Vishkin, "Implementation of simultaneous memory address access in models that forbit it," J. Algorithms 4 (1983), 45-50.

[V 83b] U. Vishkin, "O(log n) and optimal parallel biconnectivity algorithms," Technical Report #69, Computer Science Department, New York University, New York, New York, 1983.

[V 83c] U. Vishkin, "Synchronous parallel computation - a survey", Technical Report #71, Computer Science Department, New York University, New York, New York, 1983.

[V 84] U. Vishkin, "An efficient parallel strong orientation", Technical Report #109, Computer Science Department, New York University, New York, New York, 1984.

[Wi 75] Winograd, S., "On the evaluation of certain arithmetic expressions", JACM 22, 4(1975), pp. 477-492.

[Wy 79] J.C. Wyllie, "The complexity of parallel computation", Technical Report TR 79-387, Department of Computer Science, Cornell University, Ithaca, New York, 1979.

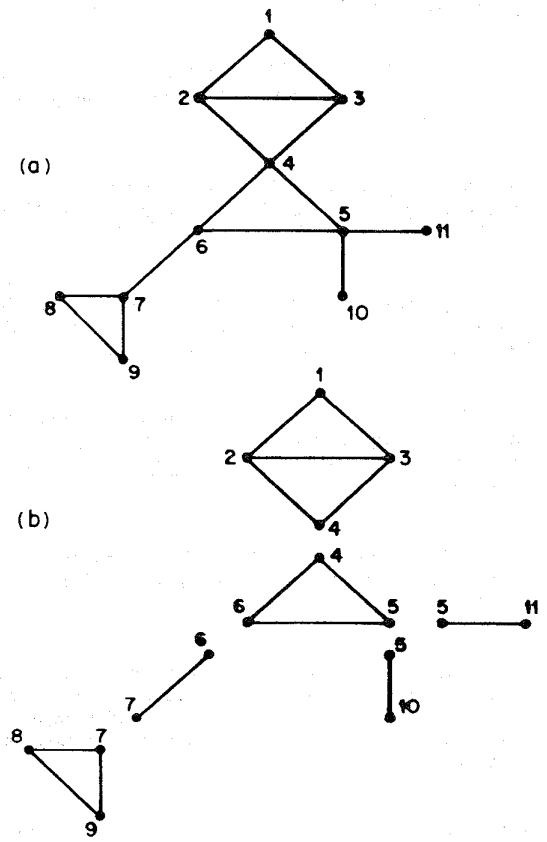

Figure 1. (a) An undirected graph.
(b) Its blocks. Vertices 4,5,6 and 7 are cut vertices. Edges {6,7}, {5,10}, and {5,11} are bridges.

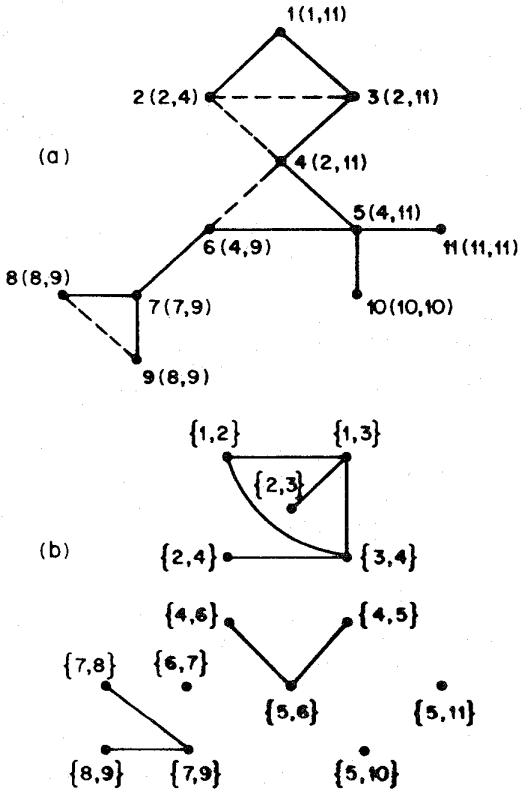

Figure 2.
(a) A spanning tree of the graph in Figure 1. Dashed edges are non-tree edges. Vertices are numbered in preorder. Numbers in parentheses are the low and high number of each vertex.
(b) The auxiliary graph G'.