AN EFFICIENT PARALLEL BICONNECTIVITY ALGORITHM

by

Robert E. Tarjan
&
Uzi Vishkin

Technical Report - #69
Ultracomputer Note #31
May 1983
AN EFFICIENT PARALLEL BICONNECTIVITY ALGORITHM

by

Robert E. Tarjan
&
Uzi Vishkin

Technical Report - #69
Ultracomputer Note #51
May 1983

Robert E. Tarjan, Bell Laboratories, Murray Hill, N.J. 07974
Uzi Vishkin, Courant Institute, New York University, N.Y. 10012
AN EFFICIENT PARALLEL BICONECTIVITY ALGORITHM

Robert E. Tarjan
Bell Laboratories
Murray Hill, NJ 07974

Uzi Vishkin
Courant Institute, New York University
New York, NY 10012

May, 1983

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 speed-up, assuming an adjacency matrix representation of the input.

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 81a].)

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.

Implementation 2 is faster than any of the previously known parallel algorithms [SJ 81], [Ec 79b], [TC 82]. 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 $O(n^2/\log n)$ (resp. $O(mn+n^2\log n)$) processors. Tsin and Chin's algorithm [TC 82] matches the bounds of our implementation 3 but is more complicated. These algorithms use the CREW PRAM model, which is somewhat weaker than the CRCW PRAM model. However, Eckstein [Ec 79a] and Vishkin [V 81a] 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.

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 [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. A similar but more complicated block-finding algorithm was discovered independently by Tsin [Ts 82].

2. New implementation techniques for parallel algorithms on trees. These techniques allow the computation of various kinds of information about the tree structure in $O(\log n)$ time, and are likely to have other applications.

The remainder of the paper consists of three 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. Section 4 sketches our alternative parallel implementation.

Note. Whenever specifying the number of processors used by a parallel algorithm we ignore the constant factor. 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
[T 82]. U. Vishkin independently rediscovered a similar
algorithm in 1983 and proposed a parallel implementation
[V 83]. 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* 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 (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 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

---

*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.
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 parallel implementation. The algorithm can use any spanning tree, rather than just a depth-first spanning tree. A similar but more complicated algorithm was developed independently by Tsin \([Ts 82]\).

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

(ii) \(\{u,v\},\{x,w\}\), where \(u \rightarrow v\) and \(x \rightarrow w\) 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 \(u \rightarrow v\) and \(v \rightarrow w\) are edges of \(T\) and some edge of \(G\) joins a descendant of \(w\) with a non-descendant of \(v\).
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 $G\setminus T$ 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 $L'$. 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 non-descendant 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 \( L' \). The existence of \( \{x,y\} \) implies by Case (iii) that any two edges on the tree path from \( z \) to \( x \) are adjacent in \( L' \). 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 \( L' \). Thus all edges on the basis cycle are in the same connected component of \( L' \). 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 \( \text{nd}(v) \) of each vertex \( v \) by processing the vertices in postorder using the recurrence \( \text{nd}(v) = 1 + \sum \text{nd}(w) | v \rightarrow w \text{ in } T \) \). (We regard every vertex as a descendant of itself.) A vertex \( v \) is a descendant of another vertex \( w \) if and only if \( v \leq w \leq v + \text{nd}(v)-1 \) [Ta 74].

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

\[
\text{low}(v) = \min\{\{v\} \cup \{\text{low}(w) \mid v \rightarrow w \text{ in } T\} \cup \{w \mid \{v,w\} \text{ in } G-T\});
\]
\[
\text{high}(v) = \max\{\{v\} \cup \{\text{high}(v) \mid v \rightarrow w \text{ in } T\} \cup \{w \mid \{v,w\} \text{ in } G-T\}).
\]

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

**Step 4.** Find the connected components of \( L'' \) using any kind
of linear-time search.

**Step 5.** Extend the equivalence relation on the edges of \( T \)
(the vertices of \( L'' \)) 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 L', and L" 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 Sections 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 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: \texttt{adj}(i) points to the first such edge and \texttt{next}((i, j)) points to the edge after (i, j) on i's list. (If there is no such edge, \texttt{next}((i, j)) = \texttt{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. \[\Box\]

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 \alpha. When a processor \texttt{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)) = \texttt{pr}(i,j). For a fixed value of D(i), only one such processor succeeds. The successful processor \texttt{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 Hirshberg, Chandra and Sarwate [HCS 79].

Having constructed 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 [W 79]. For each edge \((i,j)\), we initialize \texttt{treenext}((i,j)) = \texttt{next}((i,j)) and then repeat the following step, in parallel on all edges \((i,j)\), until none of the \texttt{treenext} values change: if \texttt{treenext}((i',j)) is not \texttt{null} and not marked, replace \texttt{treenext}((i,j)) by \texttt{treenext} (\texttt{treenext}((i,j))). This takes \(O(\log m)\) iterations over the edges. Once all the \texttt{treenext} values are computed, we define \texttt{treeadj}(i), for each vertex \(i\), to be \texttt{adj}(i) if
adj(i) is null or marked, \( \text{treenext}(\text{adj}(i)) \) otherwise. The \( \text{treeadj} \) and \( \text{treenext} \) 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 \( \text{tournext}((i, j)) \) in the tour is \( \text{treenext}((j, i)) \) if \( \text{treenext}((i, j)) \) is not null, \( \text{treeadj}(j) \) otherwise. This tour corresponds to the order of advancing and retreating along edges during a depth-first traversal 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. We can number the edges 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 \( \text{numtoend}((i, j)) = 1 \) and \( \text{ptr}((i, j)) = \text{tournext}((i, j)) \) for all edges \((i, j)\) and repeating the following computation in parallel at the edges until \( \text{ptr}((i, j)) = \text{null} \) for all \((i, j)\): if \( \text{ptr}((i, j)) \neq \text{null} \), replace \( \text{numtoend}((i, j)) \) by \( \text{numtoend}((i, j)) + \text{numtoend}(\text{ptr}(i, j)) \) and \( \text{ptr}((i, j)) \) by \( \text{ptr}(\text{ptr}(i, j)) \). Once this computation is complete, the number of edge \((i, j)\) is \( 2n-1-\text{numtoend}((i, j)) \).
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 tree 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 \(\text{numtoend}(i,j)\) 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 \(n+1 - \text{numtoend}(i,j)\) 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 \(\text{vertex}(i)\), 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 \( \text{nd}(j) \) of each vertex \( j \). If \( j \) is not the tree root, \( \text{nd}(j) \) 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 \( \text{low}(j) \) and \( \text{high}(j) \) for each vertex \( j \).

We shall describe how to compute \( \text{low} \); the computation of \( \text{high} \) is similar. Using doubling on the adjacency lists, we can compute \( \text{locallow}(j) = \min\{j \cup \{k\} | (j,k) \text{ is an unmarked (nontree) edge}\} \) for each vertex \( j \) in \( O(\log n) \) time using \( O(m) \) processors. We then compute \( \text{low}(j) \) for each vertex \( j \) using the formula

\[
\text{low}(j) = \min\{\text{locallow}(k) | j \leq k \leq j + \text{nd}(j) - 1\}.
\]

The computation of the low values uses a modified doubling computation. For \( 1 \leq j \leq n \), we maintain four values: \( \text{low}(j) \), \( \text{globallow}(j) \), \( \text{little}(j) \), and \( \text{big}(j) \), initially equal to \( \text{locallow}(j+\text{nd}(j)-1) \), \( \text{locallow}(j) \), \( j \), and \( j + \text{nd}(j) - 1 \), respectively. We use \( \lceil \log_2 n \rceil + 1 \) iterations over the vertices, one for each value of \( i \) in the range \( 0 \leq i \leq \lceil \log_2 n \rceil \). At the beginning of iteration \( i \),
\[ \text{globallow}(j) = \min\{\text{localallow}(k) \mid j \leq k \leq j+2^i-1\} \] if \( j \) is divisible by \( 2^i \) and \( j + 2^i-1 \leq n \). Furthermore if \( \text{little}(j) = \text{big}(j) \) then \( \text{low}(j) \) has its correct value; otherwise \( j < \text{little}(j) \leq \text{big}(j) \leq j + \text{nd}(j)-1 \), both \( \text{little}(j) \) and \( \text{big}(j) \) are divisible by \( 2^i \), and
\[ \text{low}(j) = \min\{\text{localallow}(k) \mid j \leq k < \text{little}(j) \text{ or } \text{big}(j) \leq k \leq j + \text{nd}(j)-1\} \].

The \( i \)th iteration consists of carrying out the following computation in parallel for each value of \( j \) in the range \( 1 \leq j \leq n \):

**Step A.** If \( \text{little}(j) < \text{big}(j) \) and \( \text{little}(j) \neq 0 \text{ mod } 2^{i+1} \), replace \( \text{low}(j) \) by \( \min\{\text{low}(j), \text{globallow}(\text{little}(j))\} \) and \( \text{little}(j) \) by \( \text{little}(j) + 2^i \).

**Step B.** If \( \text{little}(j) < \text{big}(j) \) and \( \text{big}(j) \neq 0 \text{ mod } 2^{i+1} \), replace \( \text{low}(j) \) by \( \min\{\text{low}(j), \text{globallow}(\text{big}(j)-2^i)\} \) and \( \text{big}(j) \) by \( \text{big}(j) - 2^i \).

**Step C.** If \( j \equiv 0 \text{ mod } 2^{i+1} \) and \( j + 2^{i+1}-1 \leq n \), replace \( \text{globallow}(j) \) by \( \min\{\text{globallow}(j), \text{globallow}(j+2^i)\} \).

It is easy to verify the correctness of this computation, which takes \( O(\log n) \) time using \( O(n) \) processors.

**Step 3.** Construction of the auxiliary graph \( L'' \).

This computation requires only \( O(1) \) time using \( O(m) \) processors, since testing the appropriate condition for each possible edge of \( L'' \) 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 \(L\).

**Step 4. Finding the connected components of \(L\).**

We apply the connected components algorithm of 
Shiloach and Vishkin. The information computed in step 3 
is sufficient as input to this algorithm. Once the 
algorithm finishes, each vertex \((i,j)\) of \(L\) (advance edge 
of the spanning tree) has a \(D\)-pointer to a canonical 
"vertex" \((x,y)\) representing the connected component con-
taining \((i,j)\).

**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 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 3.

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. The
algorithm has the same resource bounds as the similar but
somewhat more complicated algorithm of Tsin and Chin [TC 82].
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 81b], 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 minimization and takes $O(\log^2 n)$ time with $O(n/\log^2 n)$ processors (see [W 79]). 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 pointer 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 $\text{low}$ and $\text{high}$. 

Computing $\text{locallow}(j)$ requires $n$ parallel minimum computations. Each takes $O(\log^2 n)$ time using $O(n/\log^2 n)$ processors [W 79], a total of $O(n^2/\log^2 n)$ processors. The remainder of the $\text{low}$ computation proceeds as in Section 3 taking $O(\log n)$ time using $O(n)$ processes. The computation of $\text{high}$ is similar.

**Step 3.** Construction of the auxiliary graph $L''$. 

This is easy in $O(\log^2 n)$ time with $O(n^2/\log^2 n)$ processors.
Step 4. Finding the connected components of $L''$.

We again apply the algorithm of Chin, Lam and Chen.

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.

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 this section, as does the algorithm of Tsin and Chin [TC 82], has optimal speed-up for sparse graphs but not for dense 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.
References


Figure 1. 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.

Subgraph 2 of the line graph for funding blocks.
This book may be kept

**FOURTEEN DAYS**

A fine will be charged for each day the book is kept overtime.

<table>
<thead>
<tr>
<th>Date</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-jan 24 1985</td>
<td></td>
</tr>
<tr>
<td>May 21, 1985</td>
<td></td>
</tr>
<tr>
<td>Jan 07, 1987</td>
<td></td>
</tr>
<tr>
<td>Jan 21, 1987</td>
<td></td>
</tr>
</tbody>
</table>
An efficient parallel biconnectivity algorithm.