SIAM J. COMPUT.
Vol. 13, No. 3, August 1984

© 1984 Society for Industrial and Applied Mathematics
008

EFFICIENT PARALLEL ALGORITHMS FOR A CLASS OF GRAPH THEORETIC PROBLEMS*

YUNG H. TSIN†‡ AND FRANCIS Y. CHIN†

Abstract. In this paper, we present efficient parallel algorithms for the following graph problems: finding the lowest common ancestors for vertex pairs of a directed tree; finding all fundamental cycles, a directed spanning forest, all bridges, all bridge-connected components, all separation vertices, all biconnected components, and testing the biconnectivity of an undirected graph. All these algorithms achieve the $O(\lg^2 n)$ time bound, with the first two algorithms using $n\lceil n/\lg n\rceil$ processors and the remaining algorithms using $n\lceil n/\lg^2 n\rceil$ processors. In all cases, our algorithms are better than the previously known algorithms and in most cases reduce the number of processors used by a factor of $n \lg n$. Moreover, our algorithms are optimal with respect to the time-processor product for dense graphs, with the exception of the first two algorithms.

The machine model we use is the PRAM which is a SIMD model allowing simultaneous reads but not simultaneous writes to the same memory location.

Key words. parallel computation, analysis of algorithms, graph algorithms, directed spanning forests, lowest common ancestors, fundamental cycles, bridges, bridge-connected components, separation vertices, biconnected components, SIMD machines, PRAM

1. Introduction. The design of efficient parallel algorithms for graph problems has been investigated by many people [2], [3], [4], [5], [7], [8], [12], [15], [16], [17]. In particular, Chin, Lam and Chen [3], [4] designed parallel algorithms for several graph problems in which the processor-time products achieve the lower bounds for the corresponding sequential algorithms for dense graphs. In this paper, we present efficient parallel algorithms for other graph problems in which the processor-time products differ from the lower bounds for their sequential counterparts for dense graphs by at most a factor of $\lg n$.[1]

We are interested in the following graph problems: finding the lowest common ancestors for $q(1 \leqq q \leqq n^2)$ vertex pairs of a directed tree; finding a complete set of fundamental cycles, a directed spanning forest, all bridges, all bridge-connected components, all biconnected components, all separation vertices and testing the biconnectivity of an undirected graph. This class of problems has also been studied by Savage [15] and Savage and Ja'Ja' [16]. They designed parallel algorithms for these problems and achieved an $O(\lg^2 n)$ time bound with the processor-time products being $O(n^2 \lg^2 n)$ for the directed spanning tree problem and being $O(n^3)$ or $O(n^2(\lg n)^m)$, where $m \geqq 3$, for the remaining problems. In this paper, we present parallel algorithms for the same class of problems. Our algorithm for the lowest common ancestors problem takes $O(\lceil q/nK\rceil \cdot \lg n + n/K)$ time with $nK (K > 0)$ processors. The algorithm for the fundamental cycles problem takes $O(\lceil |E|/nK\rceil \cdot \lg n + n/K + \lg^2 n)$ time with $nK (K \geqq 1)$ processors, where E is the edge set of the undirected graphs. The algorithms for the remaining problems all take $O(n/K + \lg^2 n)$ time with $nK (K \geqq 1)$ processors. In particular, an $O(\lg^2 n)$ time bound can be achieved with $K = \lceil n/\lg n\rceil$ for the first two problems and with $K = \lceil n/\lg^2 n\rceil$ for the remaining problems. As the processor-time products of our algorithms are at most $O(n^2 \lg n)$, for $1 \leqq K \leqq \lceil n/\lg n\rceil$, our algorithms are better than the previously known results in all cases, and in most cases

* Received by the editors April 5, 1982 and in final revised form June 1, 1983. This research was supported by the Natural Science and Engineering Research Council of Canada under grant NSERC-A4319.

† Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G 2H1.

‡ Present address: Department of Computer Science, Memorial University of Newfoundland, St. John's, Newfoundland, Canada A1C 5S7.

[1] Throughout this paper, we use $\lg n$ to denote $\lceil \log_2 n\rceil$.

use less processors by a factor of $n \lg n$. Except for the algorithms for the first two problems, the processor-time products of our algorithms are $O(n^2)$, which is optimal for dense graphs.

The computation model we use is the single-instruction stream multiple-data stream (SIMD) model. We assume that all processors have access to a common memory, and that simultaneous reads from the same location are allowed but simultaneous writes on the same location are prohibited. This model is called PRAM in [6].

In describing our parallel algorithms, we use the instruction introduced by Preparata and Vuillemin [11]. Specifically, parallel operations are controlled by

for all $i:P(i)$ **pardo** instructions **dopar**;

where $P(i)$ is a predicate of i.

2. Definitions and notation. A *graph* $G(V, E)$ consists of a finite nonempty set V of vertices and a set E of pairs of vertices called edges. If the edges are unordered pairs, then G is *undirected*; otherwise G is *directed*. Without loss of generality, we assume $V = \{1, 2, \cdots, n\}$ throughout this paper. If for every two vertices u, v in V, there is a path in G joining u and v, then G is *connected*. Each connected maximal subgraph of G is called a *component* of G. An *adjacency matrix* M of G is a $n \times n$ Boolean matrix such that $M[u, v] = 1$ if and only if $(u, v) \in E$. A *tree* is a connected undirected graph with no cycles in it. Let $T(V', E')$ be a directed graph, T is said to have a *root* r, if $r \in V'$ and every vertex $v \in V'$ is reachable from r via a directed path. If the underlying undirected graph of T is a tree, then T is a *directed tree*. If, moreover, the underlying graph of T is a subgraph of a connected undirected graph $G(V, E)$ and $V' = V$, then T is a *directed spanning tree* in G. A *directed forest* is a graph whose connected components are directed trees. If T is a directed forest such that each directed tree in T is a directed spanning tree of a component of an undirected graph G and vice versa, then T is called a *directed spanning forest* of G. If the edges of T are all reversed, the resulting graph is called an *inverted spanning forest* of G. *Inverted spanning trees*, *inverted trees*, *inverted forests*, etc. are defined similarly. Throughout this paper, we denote the "undirected" path from vertex a to vertex b in a (directed) tree by $[a *\!\!\rightarrow b]$, and by $[a *\!\!\rightarrow b)$ if vertex b is to be excluded. If the path consists of at least one edge, then the "$*$" is removed from the notation.

An inverted tree T is called an *ordered tree* if the sons of every vertex in T are ordered. If v is the ith son of a vertex in T, then the *rank* of v is i.

Let $T(V', E')$ be a directed tree, and u, $v \in V'$, the *lowest common ancestor* (LCA (u, v)) of u and v in T is the vertex $w \in V'$ such that w is a common ancestor of u and v, and any other common ancestor of u and v in T is also an ancestor of w in T. If T is a spanning tree of a connected, undirected graph G, let (u, v) be an edge in $G - T$, then the cycle in G consisting of the paths $[u *\!\!\rightarrow \mathrm{LCA}(u, v)]$, $[\mathrm{LCA}(u, v) *\!\!\rightarrow v]$ and the edge (v, u) is a *fundamental cycle* in G. Let $e \in E$, e is a *bridge* in G if and only if e is not on any cycle in G. Let B be the set of bridges in G, every connected component of the graph $G'(V, E - B)$ is a *bridge-connected component* of G. Let $a \in V$, if there exist $u, v \in V$ such that u, v, a are all distinct and such that every path connecting u and v in G passes through a, then a is called a *separation vertex* of G. A graph is *biconnected* if it contains no separation vertex. Every maximal biconnected subgraph of G is called a *biconnected component* of G. To test the *biconnectivity* of G, is to test if G is biconnected.

3. Two useful lemmas. In this paper, we will frequently use the following two lemmas in analyzing the time and processor complexities.

LEMMA 3.1. *Given n elements* $\{a_0, a_1, \cdots, a_{n-1}\}$, *let f be a function to be applied to every element. If computing* $f(a_i)$ *takes t time units and* $K(\geqq 1)$ *processors are provided, then* $f(a_i)$, $0 \leqq i \leqq n-1$, *can be computed in* $(\lceil n/K \rceil * t)$ *parallel time units.*

LEMMA 3.2 [3], [4]. *Given n elements* $\{a_0, a_1, \cdots, a_{n-1}\}$ *and K processors,* $A(n) = a_0 * a_1 * a_2 * \cdots * a_{n-1}$ *can be computed in T parallel time units where* $*$ *is any associative binary operator and*

$$T = \begin{cases} \lceil n/K \rceil - 1 + \lg K & \text{if } \lfloor n/2 \rfloor > K, \\ \lg n & \text{if } \lfloor n/2 \rfloor \leqq K. \end{cases}$$

4. Finding all paths from the vertices to the roots in an inverted forest. In this section, we present a method for constructing an array, denoted by $\mathbf{F}^+$, in which each row contains a path from a vertex to a root in an inverted forest. The array will be very useful in the design of parallel algorithms presented in the following sections.

Let $T(V', E')$ be an inverted forest with $|V'| = n$, without loss of generality, we assume $V' = \{1, 2, \cdots, n\}$. Let $\{T_j\}$ be the set of all inverted trees in T and $\{r_j\}$ be the set of all their roots.

DEFINITION. $F: V' \to V'$ is a function such that

$$F(i) = \text{the father of the vertex } i \text{ in } T \quad \text{for } i \notin \{r_j\},$$

$$F(r) = r \quad \forall r \in \{r_j\}.$$

The function F can be represented by a directed graph F which can be constructed from T by adding a self-loop at each root r_j in T.

From the function F, we define F^k, $k \geqq 0$, as follows:

DEFINITION. $F^k: V' \to V'$, $k \geqq 0$, is a function such that

$$F^0(i) = i \quad \forall i \in V',$$

$$F^k(i) = F(F^{k-1}(i)) \quad \forall i \in V', \quad k > 0.$$

If i is a vertex in T_j, $F^k(i)$ is the kth ancestor of i in T_j or r_j.

DEFINITION. For each $i \in V'$, if i is in T_j, for some j, then

$$\text{depth}(i) = \min\{k \mid F^k(i) = r_j \text{ and } 0 \leqq k \leqq n-1\}.$$

The concepts $F^k(i)$, $k \geqq 0$ and depth (i), $1 \leqq i \leqq n$, were first introduced by Savage in [15]. She showed that given the function F of a directed forest T (T could be a directed forest or an inverted forest), $F^k(i)$, $0 \leqq k \leqq n-1$, and depth (i), $1 \leqq i \leqq n$, can be computed in $O(\lg n)$ time with n^2 processors and $n\lceil n/\lg n \rceil$ processors respectively. In the following, we will show in Theorem 4.1 that $F^k(i)$, $0 \leqq k \leqq n-1$, $1 \leqq i \leqq n$, can indeed be computed in $O(\lg n)$ time with $n\lceil n/\lg n \rceil$ processors or in $O(\lg^2 n)$ time with $n\lceil n/\lg^2 n \rceil$ processors, and then depth (i) can be computed in $O(\lg n)$ additional time with n processors.

THEOREM 4.2. (i) *Given the function F of a directed or an inverted forest T,* $F^k(i)$, $i \in V'$, $0 \leqq k \leqq n-1$ *can be computed in* $O(n/K + \lg n)$ *time with* nK $(K > 0)$ *processors.*

(ii) *Given* $F^k(i)$, $0 \leqq k \leqq n-1$, $1 \leqq i \leqq n$, *and* $nK(K > 0)$ *processors, depth* (i), $1 \leqq i \leqq n$ *can be computed in* $O(\lg(n/K))$ *time if* $K \geqq 1$ *or in* $O(\lceil 1/K \rceil \lg n)$ *time if* $0 < K < 1$.

Proof. To compute F^k, for all $0 \leqq k \leqq n-1$, we proceed in two steps:

1. **for** i: $1 \leqq i \leqq n$ **pardo** $F^0(i) := i$; $F^1(i) := F(i)$ **dopar**;
2. **for** $t := 0$ **to** $\lg(n-1) - 1$ **do**
   for s: $1 \leqq s \leqq 2^t$, i: $1 \leqq i \leqq n$ **pardo**
   $F^{2^t+s}(i) := F^{2^t}(F^s(i))$
   dopar;

If nK processors are given, it is clear that step 1 can be computed in $O(\lceil 1/K \rceil)$ time (Lemma 3.1). Step 2 can be computed in

$$\sum_{t=0}^{\lg(n-1)-1} (\lceil 2^t/K \rceil) = \lg K + \sum_{t=\lg K}^{\lg(n-1)-1} (\lceil 2^t/K \rceil)$$

$$< \lg K + 1/K \sum_{t=\lg K}^{\lg(n-1)-1} 2^t + \lg(n-1) - \lg K$$

$$= O(n/K + \lg n) \text{ parallel time units.}$$

Once $F^k(i)$, $1 \leqq i \leqq n$, $0 \leqq k \leqq n-1$, are computed, depth (i), $1 \leqq i \leqq n$, can be found by performing a binary search on the ordered sequence $F^0(i), F^1(i), \cdots, F^{n-1}(i)$, for each i, searching for the left-most occurrence of r_j using $F^{n-1}(i)(=r_j)$ as the key. This takes a total of $\lceil 1/K \rceil \lg n$ time units if $0 < K < 1$. For $K \geqq 1$, the search is performed in the following way: divide the sequence into $\lceil n/K \rceil$ segments, assign one processor to each segment and perform simultaneously a binary search on each segment. After this step, every processor compares the element it finds with the preceding and succeeding elements in the sequence. There is exactly one processor which does not have all the three elements distinct or identical and this processor locates the left-most occurrence of r_j. This takes a total of $\lg \lceil n/K \rceil + 3$ parallel time units. □

The actual computations of $F^k(i)$, $1 \leqq i \leqq n$, $0 \leqq k \leqq n-1$, and depth (i), $1 \leqq i \leqq n$, are performed in an array $\mathbf{F}^+$ in which $\mathbf{F}^+[i, k]$ contains $F^k(i)$. After the computations are finished, each row of $\mathbf{F}^+$ is right shifted so that all the r_j's except the left-most one are eliminated. As a consequence, the right-most column of the array contains only the roots from $\{r_j\}$. Furthermore, for each vertex i, all occurrences of i appear only in column $(n-1)-$depth (i). For each row i, a number, $n+i$, acting as an undefined value, is inserted into the first $(n-1)-$depth (i) entries. These adjustments are done for convenience and not out of necessity and they take $O(n/K)$ time with $nK (K > 0)$ processors (Lemma 3.1). The adjusted array, $\mathbf{F}^+$, of a directed tree is depicted in Fig. 4.1. Note that the ith row in $\mathbf{F}^+$ contains the path from vertex i to a root in T.

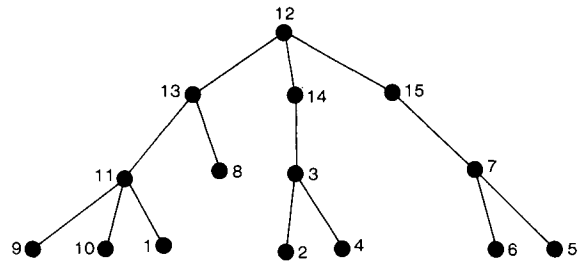
5. Finding a directed spanning forest in an undirected graph. In this section, we present an efficient parallel algorithm for finding a directed spanning forest in an undirected graph $G(V, E)$. In view of the fact that it is the inverted spanning forest of G which is useful in the design of other parallel algorithms in the following sections, the algorithm presented below actually constructs an inverted spanning forest. Nevertheless, converting an inverted spanning forest into a directed spanning forest is straightforward. This algorithm will serve as the backbone of the other algorithms presented in the following sections.

This algorithm is based on the algorithm for finding an undirected spanning forest presented in [3] and the array $\mathbf{F}^+$ presented in the last section. The latter is used to assign a direction to each edge in the undirected spanning forest generated by the former.[2]

We first give a general description for the strategy used in our algorithm. In the course of running the algorithm for finding an undirected spanning forest [3], a number of 1-tree-loop's [7][3] are generated. Each of the 1-tree-loop's is a directed graph whose vertices are supervertices generated during the previous iteration (a *supervertex* is a

[2] We assume the reader is familiar with the undirected spanning forest algorithm. For those who are not, we refer them to reference [3].

[3] A 1-tree-loop is a directed graph in which every vertex has outdegree 1 and in which there is exactly one cycle and the length of the cycle is 2.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	16	16	16	16	16	16	16	16	16	16	16	1	11	13	12
2	17	17	17	17	17	17	17	17	17	17	17	2	3	14	12
3	18	18	18	18	18	18	18	18	18	18	18	18	3	14	12
4	19	19	19	19	19	19	19	19	19	19	19	4	3	14	12
5	20	20	20	20	20	20	20	20	20	20	20	5	7	15	12
6	21	21	21	21	21	21	21	21	21	21	21	6	7	15	12
7	22	22	22	22	22	22	22	22	22	22	22	22	7	15	12
8	23	23	23	23	23	23	23	23	23	23	23	23	8	13	12
9	24	24	24	24	24	24	24	24	24	24	24	9	11	13	12
10	25	25	25	25	25	25	25	25	25	25	25	10	11	13	12
11	26	26	26	26	26	26	26	26	26	26	26	26	11	13	12
12	27	27	27	27	27	27	27	27	27	27	27	27	27	27	12
13	28	28	28	28	28	28	28	28	28	28	28	28	28	13	12
14	29	29	29	29	29	29	29	29	29	29	29	29	29	14	12
15	30	30	30	30	30	30	30	30	30	30	30	30	30	15	12

FIG. 4.1. *A directed tree and its array* $\mathbf{F}^+$. *Note that since* $n = 15$, *any number greater than* 15 *serves as an undefined value in the array.*

vertex in G or a 1-tree-loop). The edges of these 1-tree-loop's will be included in the undirected spanning forest and all these edges are directed edges, whose directions are ignored by the algorithm in [3]. If the only loop in a 1-tree-loop is destroyed by eliminating the out-going edge from the smallest-numbered-vertex, the resulting graph is an inverted tree. As a result, when the loops of all the 1-tree-loop's are destroyed in this way, the resulting graph (built by embedding the modified (acyclic) 1-tree-loop's created during one iteration into the modified (acyclic) 1-tree-loop's created during the following iteration) may well be an inverted spanning forest. Unfortunately, this is not the case in general, because some vertices may end up with two fathers. This situation is depicted in Fig. 5.1, where a directed edge (a, b) is selected during iteration $j+1$ to connect two supervertices S_1 and S_2 created during iteration j. The two graphs resulting from the two supervertices are inverted trees. However, since a is not the root r_1 of S_1, a will have two fathers after S_1 and S_2 have been included into a single supervertex. Therefore, the graph $S_1 \cup S_2$ is not an inverted tree, by definition, unless the directions of all the edges on the path from a to r_1 are reversed. The same situation occurs in $S_2 \cup S_3$ when the directed edge (c, d) is selected to connect S_2 and S_3. To overcome this difficulty, we have to reverse the directions of all edges on the path from a to r_1 and those on the path from c to r_2. The array $\mathbf{F}^+$, described in § 4, contains the path from any vertex to a root in an inverted forest T; hence we can generate the array $\mathbf{F}^+$ covering both S_1 and S_2. By retrieving the ath row and the cth row of $\mathbf{F}^+$, we can identify the set of all edges whose directions are to be reversed in S_1 and S_2 respectively.

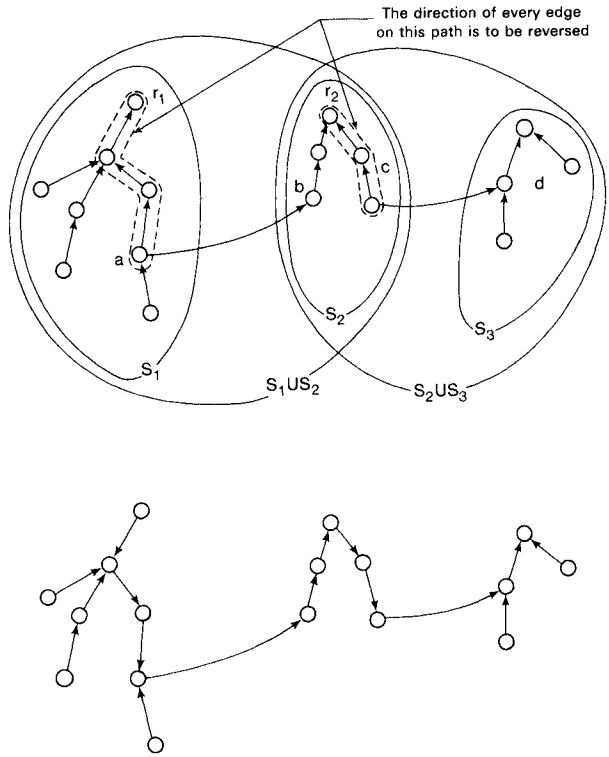


FIG. 5.1.

Our algorithm runs in two stages.

ALGORITHM DSF.

Stage 1 (* The first stage is basically a modified version of the algorithm for finding an undirected spanning forest. We refer the reader to reference [3] for the details.*)

Execute the algorithm for finding an undirected spanning tree; during each iteration j, $1 \leqq j \leqq \lg n$, record the following information:

a. Convert the forest of all 1-tree-loops generated during this iteration into a forest of inverted trees by eliminating the edge from the smallest-numbered-vertex of each 1-tree-loop and store the forest in a vector F_j. (* Note: This vector acts as the function F defined in § 4.*)

b. Record the "actual" edges in G establishing the connection specified in F_j. (* Note: The edges recorded in F_j are *pseudo* edges which connect "supervertices". They do not exist in G. However, for each pseudo edge, there exists a corresponding actual edge in G.*)

c. The vector $D[1 \,.\,.\, n]$ generated during this iteration is stored as D_j. (* Note: $D_j[v]$ is the supervertex containing vertex v when iteration j is completed.*)

Stage 2

1. Generate $\mathbf{F}_j^+$'s from F_j, $1 \leqq j \leqq \lg n$.

2. (* Adjust the directions of the edges, starting from those recorded during iteration $\lg n$, gradually down to those recorded during interation 1. *)

$R' := \{v \in V | D_{\lg n}[v] = v\}$;
(* Note: In the following **for** loop, R' contains the tails of those actual edges in G which connect two supervertices in the inverted trees generated during iteration i, where $j < i \leqq \lg n$. It includes all those vertices which have two or more fathers in the directed graph formed upon the inverted trees. *)
for $j := \lg n$ **downto** 1 **do**
begin
i) For every $r' \in R'$,
reverse the direction of every "pseudo" edge lying on the path from the supervertex $D_j[r']$ to the root of the inverted tree, in F_j, containing $D_j[r']$;
ii) Output all the "actual edges" in G corresponding to the pseudo edges in F_j;
iii) $R' = R' \cup \{v \in V | v$ is the tail of an "actual" edge output in step ii)$\}$
end;

A complete example is given in Fig. 5.2 and a detailed implementation using the method described above is given in the Appendix.

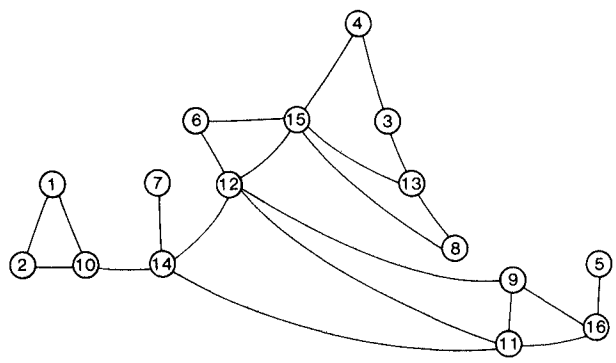


FIG. 5.2(i) $G(V, E)$.

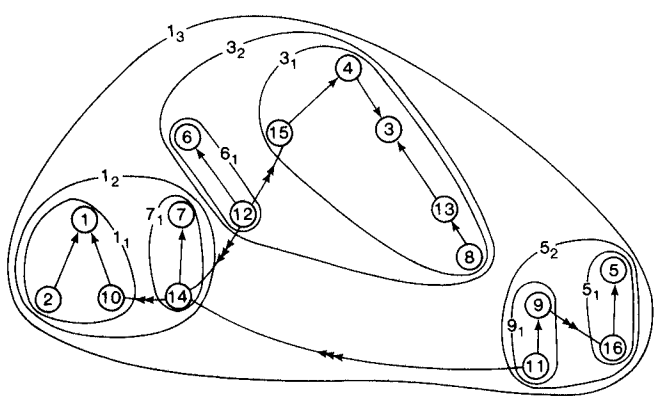


FIG. 5.2(ii). *A potential inverted spanning tree of G.* → *a directed edge selected during the first iteration;* →→ *a directed edge selected during the second iteration;* →→→ *a directed edge selected during the third iteration.*

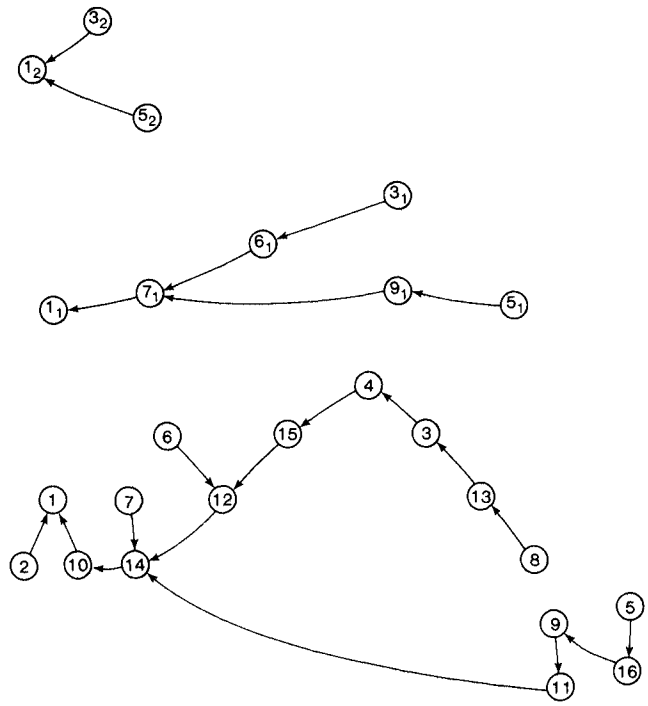


FIG 5.2(iii). *An inverted spanning tree of G.*

THEOREM 5.1. *Algorithm* DSF *correctly generates an inverted spanning forest for an undirected graph.*

Proof. (Backward induction.) In stage 1, an inverted forest F_j is correctly generated during each iteration j, $1 \leqq j \leqq \lg n$[3]. In stage 2, supposing that after processing F_i, $j \leqq i \leqq \lg n$, an inverted forest F'_j is created. Clearly, F'_j and F_j must have the same vertex set V_j. When processing F_{j-1}, it should be clear that there exists a one-to-one correspondence between the vertices in V_j and the inverted trees in F_{j-1}. This implies that no two instances of r' in R' will belong to the same inverted tree in F_{j-1}. As a result, after step i, each inverted tree in F_{j-1} is effectively modified so as to root at the supervertex $D_{j-1}[r']$. These modified inverted trees are then embedded into the inverted forest F'_j in step 2ii), the resulting directed graph F'_{j-1} is clearly an inverted forest. But $F'_{\lg n} = F_{\lg n}$ is an inverted forest initially, therefore, by induction, F'_1 must be an inverted forest and hence an inverted spanning forest for G. □

THEOREM 5.2. *Finding an inverted spanning forest takes* $O(n/K + \lg^2 n)$ *time with* nK $(K \geqq 1)$ *processors.*

Proof. Stage 1 takes $O(n/K + \lg^2 n)$ time with $nK(K \geqq 1)$ processors [3]. Since the total number of edges in the inverted forest is at most

$$\sum_{j=1}^{\lg n} \lfloor n/2^{j-1} \rfloor < 2n,$$

the creation of $\mathbf{F}_j^+$, $1 \leqq j \leqq \lg n$, in step 1 of stage 2 can be done in $O(n/K + \lg n)$ time with $nK(K \geqq 1)$ processors. Steps 2ii) and iii) each takes $O(1)$ time for each iteration.

Since the size of $\mathbf{F}_j^+$, $1 \leqq j \leqq \lg n$, is $\lceil n/2^{j-1} \rceil \times \lceil n/2^{j-1} \rceil$, step 2i) requires

$$\sum_{j=1}^{\lg n} \lceil \lceil n/2^{j-1} \rceil^2 / nK \rceil < \sum_{j=1}^{\lg n} \lceil n/2^{j-1} \rceil^2 / nK + \lg n$$
$$= O(n/K + \lg n) \quad \text{time for } \lg n \text{ iterations.}$$

Hence the theorem. □

Note that the processor-time product is $O(n^2)$, when $1 \leqq K \leqq \lceil n/\lg^2 n \rceil$, the algorithm is thus optimal for dense graphs.

6. Finding the lowest common ancestors of q vertex pairs in a directed tree. Let $T(V', E')$ be a directed tree and $V' = \{1, 2, \cdots, n\}$. We shall make use of the array $\mathbf{F}^+$ to design a parallel algorithm for finding the lowest common ancestors of q vertex pairs in T. Let a and b be a vertex pair, if c is their lowest common ancestor, then row a and row b of $\mathbf{F}^+$ will have identical contents between column $(n-1) - \text{depth}[c]$ and column $n-1$, inclusive, and will have different contents in the other columns. As a result, to determine c, we can perform a binary search on row a and row b simultaneously in the following way: if the two entries being examined in row a and row b (in the same column, of course) are different, the search is continued on the right-half, otherwise it is continued on the left-half. It takes $\lg n + 1$ time units to find c with one processor. In general, we have:

THEOREM 6.1. *Given q vertex pairs, $1 \leqq q \leqq n^2$, finding the lowest common ancestors for these vertex pairs takes $O(\lceil q/nK \rceil \cdot \lg n + n/K)$ time if $nK (K > 0)$ processors are available.*

Proof. Finding the lowest common ancestors of the q vertex pairs takes $\lceil q/nK \rceil \cdot \lg n$ time units, if $nK \leqq q \leqq n^2$ (Lemma 3.1) or $\lg n + 1$ time units, if $nK > q$. Constructing the array $\mathbf{F}^+$ takes $O(n/K + \lg n)$ time, thus finding the lowest common ancestors of $q (1 \leqq q \leqq n^2)$ vertex pairs, takes $O(\lceil q/nK \rceil \cdot \lg n + n/K)$ time with nK $(K > 0)$ processors. □

In particular, when $K = n$ and $\lceil n/\lg n \rceil$, this algorithm takes $O(\lg n)$ and $O(\lg^2 n)$ time, respectively.

7. Finding all fundamental cycles of a connected, undirected graph. Without loss of generality, we assume that the undirected graph $G(V, E)$ is connected from this section onwards.

It is known that a set of fundamental cycles of a connected, undirected graph $G(V, E)$ can be determined from a spanning tree $T(V, E')$ of G [14]. Specifically, if (a, b) is an edge in $G - T$, then (a, b) together with the paths $[b *\!\!\rightarrow \text{LCA}(a, b)]$ and $[\text{LCA}(a, b) *\!\!\rightarrow a]$ form a fundamental cycle.

Based on the above observation, we can find a set of fundamental cycles of G as follows: First, an inverted spanning tree T of G is found, using the algorithm presented in § 5, which takes $O(n/K + \lg^2 n)$ time with $nK (K \geqq 1)$ processors. The lowest common ancestor algorithm is then called to determine the lowest common ancestor for every pair of vertices (a, b) in $G - T$. The algorithm returns the ordered pair $(\mathbf{LCA}^+, \mathbf{F}^+)$ and the vector depth, where $\mathbf{LCA}^+[a, b]$ contains the lowest common ancestor of (a, b). A vector $\mathbf{P}^+$ is then created such that $\mathbf{P}^+[v]$ contains the value $(n-1) - \text{depth}[v]$, which is the column number of v in $\mathbf{F}^+$. Hence, for each (a, b) in $G - T$, the path from column $\mathbf{P}^+[a]$ to column $\mathbf{P}^+[\mathbf{LCA}^+[a, b]]$ in row a and the path from column $P^+[b]$ to column $\mathbf{P}^+[\mathbf{LCA}^+[a, b]]$ in row b of $\mathbf{F}^+$ and the edge (a, b) determine a fundamental cycle in G.

The correctness of the algorithm is easily verified. Since the number of vertex pairs $q = |E - E'|$, the algorithm obviously takes $O(\lceil |E|/nK \rceil \cdot \lg n + n/K + \lg^2 n)$ time with nK $(K \geqq 1)$ processors. In particular, the $O(\lg^2 n)$ time bound is achieved with $K = n/\lg n$. Note that the output of the algorithm is stored in an $O(n^2)$ compact data structure, which consists of the triple $(\mathbf{P}^+, \mathbf{LCA}^+, \mathbf{F}^+)$.

8. Finding the HLCA(u)'s. The algorithms we present in the following sections rely heavily on the function, HLCA(u), $\forall u \in V$, (note: The prefix H stands for highest), which is defined as follows.

DEFINITION. Let $G(V, E)$ be an undirected graph, $T(V, E')$ be its inverted spanning tree and $u \in V$. HLCA (u) = LCA (u, v), where $(u, v) \in E - E' \cup \{(u, u)\}$ and depth (LCA (u, v)) $\leqq$ depth (LCA (u, v')), $\forall (u, v') \in E - E' \cup \{(u, u)\}$.

Figure 8.1 gives an illustration of HLCA (u). The solid lines and circles represent the edges and vertices of an inverted spanning tree of an undirected graph. The dotted lines represent the edges in the graph $G - T$ emanating from a particular vertex u. To compute HLCA (u), $\forall u \in V$, we may first use the lowest common ancestor algorithm to find LCA (u, v), $\forall (u, v) \in E - E' \cup \{(u, u)\}$ and then apply Lemma 3.2 to find HLCA (u), $\forall u \in V$. However, in doing so, we will require $O(\lceil |E - E'|/nK \rceil \cdot \lg n + n/K)$ time if $nK (K > 0)$ processors are available. In this section, we show a way of finding HLCA (u), $\forall u \in V$ in $O(n/K + \lg n \lg\lg n)$ time with nK $(K \geqq 1)$ processors. This method allows us to design optimal parallel algorithms for the graph theoretic problems discussed in the following sections.

The method is based on the preorder numbering [9] of the vertices in an ordered spanning tree $T(V, E')$ of G. We denote the preorder number of a vertex v by pre (v).

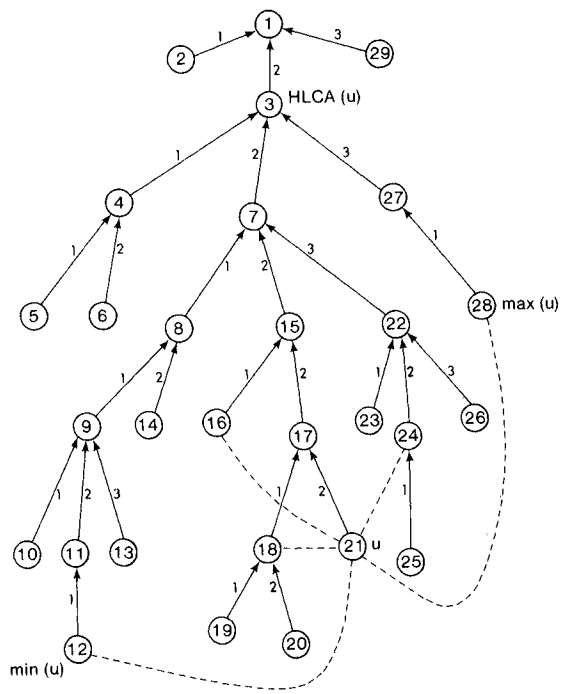


FIG. 8.1.

DEFINITION. *Let* $u, v \in V$, $u \preceq v$ iff u is an ancestor of v, $u < v$ iff u is a proper ancestor of v.

LEMMA 8.1. *Let* $u, v \in V$, $v \preceq u$ *iff* $\text{pre}(v) \leqq \text{pre}(u) < \text{pre}(v) + nd(v)$, *where* $nd(v)$ *is the number of descendants of* v.

Proof. Immediate from the definition of preorder traversal. □

LEMMA 8.2. *Let* $(u, v), (u, w) \in E - E'$;

(i) *if* $\text{pre}(v) < \text{pre}(w) < \text{pre}(u)$,
then $\text{depth}(\text{LCA}(u, v)) \leqq \text{depth}(\text{LCA}(u, w))$;

(ii) *if* $\text{pre}(v) > \text{pre}(w) > \text{pre}(u)$,
then $\text{depth}(\text{LCA}(u, v)) \leqq \text{depth}(\text{LCA}(u, w))$.

Proof. (i) By Lemma 8.1, $\text{pre}(\text{LCA}(u, v)) \leqq \text{pre}(v)$ and $\text{pre}(u) < \text{pre}(\text{LCA}(u, v)) + nd(\text{LCA}(u, v))$. Therefore $\text{pre}(\text{LCA}(u, v)) < \text{pre}(w) < \text{pre}(\text{LCA}(u, v)) + nd(\text{LCA}(u, v))$. By Lemma 8.1, $\text{LCA}(u, v) \preceq w$. Hence, $\text{depth}(\text{LCA}(u, v)) \leqq \text{depth}(\text{LCA}(u, w))$. Part (ii) can be proved similarly. □

Lemma 8.2 points out that we can reduce the problem of finding HLCA (u) to that of finding the lowest common ancestor of two particular vertices in $\{v \mid (u, v) \in E - E'\} \cup \{u\}$.

DEFINITION. Let $u \in V$, $W = \{v \mid (u, v) \in E - E'\} \cup \{u\}$.

$$\text{pmax}(u) = v, \quad \text{where } v \in W \text{ and } \text{pre}(v) \geqq \text{pre}(w), \quad \forall w \in W;$$

$$\text{pmin}(u) = v, \quad \text{where } v \in W \text{ and } \text{pre}(v) \leqq \text{pre}(w), \quad \forall w \in W.$$

COROLLARY 8.3. $\text{HLCA}(u) = (\min \preceq)\{\text{LCA}(u, \text{pmin}(u)), \text{LCA}(u, \text{pmax}(u))\}$.

Proof. Immediate from Lemma 8.2. □

COROLLARY 8.4. $\text{HLCA}(u) = \text{LCA}(\text{pmin}(u), \text{pmax}(u))$.

Proof. From Corollary 8.3, $\text{HLCA}(u) \preceq \text{pmin}(u)$ and $\text{HLCA}(u) \preceq \text{pmax}(u)$. Thus, $\text{HLCA}(u) \preceq \text{LCA}(\text{pmin}(u), \text{pmax}(u))$. By definition, $\text{pre}(\text{pmin}(u)) \leqq \text{pre}(u) \leqq \text{pre}(\text{pmax}(u))$. This implies $\text{pre}(\text{LCA}(\text{pmin}(u), \text{pmax}(u))) \leqq \text{pre}(u) < \text{pre}(\text{LCA}(\text{pmin}(u), \text{pmax}(u))) + nd(\text{LCA}(\text{pmin}(u), \text{pmax}(u)))$. By Lemma 8.1, $\text{LCA}(\text{pmin}(u), \text{pmax}(u)) \preceq u$. Therefore $\text{LCA}(\text{pmin}(u), \text{pmax}(u)) \preceq \text{LCA}(u, \text{pmin}(u))$ and $\text{LCA}(\text{pmin}(u), \text{pmax}(u)) \preceq \text{LCA}(u, \text{pmax}(u))$. By Corollary 8.3, $\text{LCA}(\text{pmin}(u), \text{pmax}(u)) \preceq \text{HLCA}(u)$. □

LEMMA 8.5. *Let* $T(V, E')$ *be a directed tree whose vertices have been labelled in preorder. Then finding* HLCA (u), $\forall u \in V$, *can be done in* $O(n/K + \lg n)$ *time with* nK $(K \geqq 1)$ *processors.*

Proof. To compute $\text{pmax}(u)$ and $\text{pmin}(u)$, $\forall u \in V$, we need $O(n/K + \lg K)$ time with nK $(K \geqq 1)$ processors (Lemma 3.2), and to find HLCA (u), $\forall u \in V$, we need to find the lowest common ancestors of the n $(\text{pmin}(u), \text{pmax}(u))$ pairs. This takes $O(n/K + \lg n)$ time with nK $(K > 0)$ processors (Theorem 6.1). □

Figure 8.1. gives an illustration of the above lemmas and corollaries. The numbers in the circles are the preorder numbers of the vertices. For instance, the preorder number of u is 21. For convenience sake, we name each vertex by its preorder number. It can be easily checked that $\text{depth}(\text{LCA}(u, 12)) < \min(\text{depth}(\text{LCA}(u, 18)), \text{depth}(\text{LCA}(u, 16)))$, and that $\text{depth}(\text{LCA}(u, 28)) < \text{depth}(\text{LCA}(u, 24))$. Furthermore, $\text{pmin}(u) = 12$, $\text{pmax}(u) = 28$, and $\text{LCA}(12, 28) = 3$, which is clearly HLCA (u).

The crucial step in computing HLCA (u), $\forall u \in V$, is to determine the preorder numbers efficiently. The usual way of numbering the vertices of a tree in preorder is to traverse the tree. However, this will result in an $O(n)$ time algorithm, which is undesirable. In the following lemma, we show that we can carry out preorder numbering in parallel without traversing the tree.

LEMMA 8.6. *Let $T(V, E')$ be an ordered tree* [9]. *For each $v \in V$,*

$$\text{pre}(v) = \sum_{u \in \text{ANC}(v)} \sum_{w \in \text{EBRO}(u)} nd(w) + na(v)$$

$$= \sum_{u \in \text{ANC}(v) - \{r\}} nds(F(u), \text{rank}(u) - 1) + 1 + \text{depth}(v),$$

where

ANC (v) *is the set of all ancestors of* v;
EBRO(u) *is the set of all elder brothers of* u;
$nd(w)$ *is the number of descendants of* w;
$na(v)$ *is the number of ancestors of* v.
$nds(v, j)$ *is the total number of descendants of the first* j *sons of* v; *and*
rank (v)*is the rank of* v, *i.e., the position of* v *among all its brothers.*

Proof. Trivial. □

Let us consider the inverted spanning tree given in Fig. 8.1. again. Consider the vertex u, pre $(u) = 21$, the ancestors of u are the vertices 21, 17, 15, 7, 3 and 1. The number of descendants of the elder brothers of each of these vertices except the root are 3, 1, 7, 3 and 1, respectively. These numbers sum up to 15. The number of ancestors of u is 6, this gives rise to a total sum of 21, which is the preorder number of u.

Using Lemma 8.6, we want to show that the preorder numbers pre (v), $\forall v \in V$ can be determined in $O(n/K + \lg n \lg\lg n)$ time with nK $(K \geqq 1)$ processors. Assuming that an inverted tree T represented by an array $T[1 \ldots 2, 1 \ldots n]$ such that $\{(T[1, i], T[2, i]) | 1 \leqq i \leqq n\} = E'$ is given (we assume $T[2, r] = 0$ for the root r).

ALGORITHM Preorder.
1. Compute the array $\mathbf{F}^+$ and the vector *depth* for T.
2. Order the sons of every vertex in T, i.e., compute rank (v), $\forall v \in V$.
3. Find $nds(v, j)$, $\forall v \in V$, $1 \leqq j \leqq n(v)$, where $n(v)$ is the number of sons of v.
4. Compute pre (v), $\forall v \in V$.

LEMMA 8.7. *Algorithm Preorder takes $O(n/K + \lg n \lg\lg n)$ time with nK $(K \geqq 1)$ processors.*

Proof. Step 1 can be done in $O(n/K + \lg n)$ time (Theorem 4.2). In step 2, the ordered pairs $\{(T[2, i], T[1, i]) | 1 \leqq i \leqq n\}$ are sorted. This can be done in $O(\lg n \lg\lg n)$ time with n processor [1]. (* In fact, for $K \geqq \lg n$, we can sort n elements in $O(\lg n)$ time [1]. However, the $O(\lg n \lg\lg n)$ time suffices for our purposes here *). Assuming that the sorted T is stored in $T'[1 \ldots 2, 1 \ldots n]$, then T' is divided into segments such that in each segment, the first row contains the same vertex v in every entry, and the second row contains the set of all sons of v in T. The relative position of vertex i in the second row of the segment in which i resides, is the rank of i, i.e., rank (i).

In step 3, $nd(v)$, $\forall v \in V$, are first computed by scanning the $((n-1) - \text{depth}(v))$th column of $\mathbf{F}^+$ and counting the number of occurrences of v. By Lemma 3.2, this takes $O(n/K + \lg K)$ time. After this, $nds(v, j)$, $\forall v \in V$, $1 \leqq j \leqq n(v)$, are computed using the following formula

$$nds(v, j) = \sum_{1 \leqq i \leqq j} nd(s_i), \qquad 1 \leqq j \leqq n(v).$$

It has been shown in [10], that the partial sums $\sum_{1 \leqq i \leqq j} a_i$, $1 \leqq j \leqq n$, can be computed in $O(\lg n)$ time if n processors are given. Since for each vertex v, v has $n(v)$ sons,

the time needed to compute $nds\,(v, j)$, $1 \leqq j \leqq n(v)$, is $O(\lg(n(v)))$ if $n(v)$ processors are assigned to v. (This is possible if we make use of the sorted array T'). As a result, all these partial sums, $nds\,(v, j)$, $1 \leqq j \leqq n(v)$, $\forall v \in V$, can be computed in parallel in $\max_{v \in V} \{O(\lg(n(v)))\} = O(\lg n)$ time with $\sum_{v \in V} n(v) = n - 1$ processors.

Finally, in step 4, pre (v), $\forall v \in V$ is computed using the formula given in Lemma 8.6. We assume $nds\,(v, 0) = 0$, $\forall v \in V$. Note that ANC (v) is available in the vth row of $\mathbf{F}^+$ starting from column $(n-1) - \text{depth}\,(v)$ to column $(n-1)$, and $na\,(v)$ equals depth $(v) + 1$. By Lemma 3.2, this takes $O(n/K + \lg K)$ time.

Summing up, pre (v), $\forall v \in V$ can be determined in $O(n/K + \lg n \lg\lg n)$ time with $nK\,(K \geqq 1)$ processors □

THEOREM 8.8. *Computing* HLCA (u), $\forall u \in V$ *can be done in* $O(n/K + \lg n \lg\lg n)$ *time with* nK $(K \geqq 1)$ *processors.*

Proof. Lemmas 8.5, 8.7. □

9. Finding all bridges in a connected, undirected graph. In this section, we present an optimal parallel algorithm for finding all bridges in a connected, undirected graph. The correctness of the algorithm is based on the following theorems.

LEMMA 9.1. *Let $G(V, E)$ be a connected, undirected graph. If $e \in E$ is a bridge of G, then e is contained in every inverted spanning tree of G.*

Proof. Trivial. □

Due to this lemma, the number of edges to be examined is greatly reduced from $O(n^2)$ to $O(n)$.

LEMMA 9.2. *e is not a bridge if and only if e is on a fundamental cycle.*

Proof. Trivial. □

THEOREM 9.3. *Let $T(V, E')$ be an inverted spanning tree of a connected, undirected graph G, and $e = (a, b) \in E'$. Then (a, b) is a bridge of G if and only if for each descendant i of a, there does not exist (i, j) in $G - T$ such that* depth (LCA $[i, j]$) < depth (a).

Proof. Let $e = (a, b) \in E'$ be a bridge in G. If there exists (i, j) in $G - T$ such that i is a descendant of a in T and depth (LCA $[i, j]$) < depth (a), then the path $[i \to j \ast\!\to \text{LCA}\,[i, j] \ast\!\to b \to a \ast\!\to i]$ is a cycle containing e. This leads to a contradiction by Lemma 9.2.

Conversely, if e is not a bridge, then by Lemma 9.2, e is on a fundamental cycle C, i.e., there exists (i, j) in $G - T$ such that

$$C: [i \to j \ast\!\to \text{LCA}\,[i, j] \ast\!\to i].$$

$e \neq (i, j)$ because e is not in $G - T$. As a result, e is either on the path $[j \to \text{LCA}\,[i, j]]$ or on the path $[\text{LCA}\,[i, j] \to i]$, implying depth $(j) \geqq$ depth $(a) >$ depth $(b) \geqq$ depth (LCA $[i, j]$) or depth $(i) \geqq$ depth $(a) >$ depth $(b) \geqq$ depth (LCA $[i, j]$). Hence in either case there exists (i, j) in $G - T$ such that i is a descendent of a and depth (LCA $[i, j]$) < depth (a). □

ALGORITHM Bridges

1. Construct an inverted spanning tree $T(V, E')$ for $G(V, E)$.
2. Compute HLCA (u), $\forall u \in V$.
3. Compute $\alpha(u)$, $\forall u \in V$, where

$$\alpha(u) = \min\{\text{depth}\,(\text{HLCA}\,(w)) | u \preceq w\}.$$

4. For each $(u, F(u)) \in E'$, check if depth $(u) \leqq \alpha(u)$. $(*(u, F(u))$ is a bridge iff depth $(u) \leqq \alpha(u)*)$

The correctness of the algorithm is supported by Theorem 9.3.

THEOREM 9.4. *Algorithm Bridges runs in $O(n/K+\lg^2 n)$ time with $nK(K \geqq 1)$ processors.*

Proof. With nK $(K \geqq 1)$ processors, step 1 takes $O(n/K+\lg^2 n)$ time (Theorem 5.2). Step 2 takes $O(n/K+\lg n \lg\lg n)$ time (Theorem 8.8). Steps 3 and 4 take $O(n/K+\lg K)$ time (Lemma 3.1 and 3.2). Hence, Algorithm Bridges runs in $O(n/K+\lg^2 n)$ time with $nK(K \geqq 1)$ processors. □

10. The bridge-connected components of a connected, undirected graph. Once the bridges of a connected, undirected graph are determined, its bridge-connected components can be determined. Specifically, we eliminate all the bridges in G and then use Algorithm MOD.CONNECT [3], [4] to find the connected components of the resulting graph. Each of the connected components thus found is a bridge-connected component of G.

The algorithm obviously runs in $O(n/K+\lg^2 n)$ time with $nK(K \geqq 1)$ processors.

11. Finding all biconnected components in a connected, undirected graph. In this section, we present an optimal parallel algorithm for finding all biconnected components of a connected, undirected graph $G(V,E)$. Since a biconnected component can be completely determined by its vertex set, it suffices to find the vertex sets of all the biconnected components of G.

DEFINITION. Let $T(V,E')$ be an inverted spanning tree of $G(V,E)$. Let $e_1=(a,F(a))$, $e_2=(b,F(b)) \in E'$. $e_1 \Delta e_2$ iff

(i) e_2 is on $[a *\!\!\rightarrow \text{HLCA}(a)]$ or e_1 is on $[b *\!\!\rightarrow \text{HCLA}(b)]$; or

(ii) $(a,b) \in E-E'$ and neither $a \leq b$ nor $b \leq a$ in T.

ALGORITHM Biconnect.

1. Find an inverted spanning tree $T(V,E')$ of $G(V,E)$.
2. Compute HLCA (v) $\forall v \in V$.
3. Construct an undirected graph $G''(E',E'')$ such that $(e_1,e_2) \in E''$ iff $e_1 \Delta e_2$.
4. Find the connected components $\{B_j\}$ of G''. (* Note: Every connected component of G'' uniquely determines the vertex set of a biconnected component in G and vice versa. *)

LEMMA 11.1. (i) *For each edge $(a,b) \in E$ there exists a unique biconnected component in G containing the edge.*

(ii) *All edges in the same cycle in G belong to the same biconnected component in G.*

From the definition, if $e_1 \Delta e_2$ then e_1 and e_2 belong to the same fundamental cycle. It is easily shown that if $e_1 \Delta e_2$ and $e_2 \Delta e_3$, then e_1 and e_3 belong to the same cycle in G. This is easily generalized to:

LEMMA 11.2. *If $e_1 \Delta e_2, e_2 \Delta e_3, \cdots, e_{l-1} \Delta e_l$, then there exists a cycle in G containing both e_1 and e_l.*

THEOREM 11.3. *e and e' belong to the same connected component in G'' if and only if e and e' belong to the same biconnected component in G.*

Proof. Let e and e' belong to the same connected component in G''. Then there exists a path: $e, e_1, \cdots, e_l, e'$ in G''. This implies that $e \Delta e_1, e_1 \Delta e_2, \cdots, e_l \Delta e'$. By Lemma 11.2, e, e' belong to the same cycle in G. By Lemma 11.1 (ii), e and e' belong to the same biconnected component in G.

Let e and e' belong to the same biconnected component in G. Then there exists a simple cycle C containing e and e' in G. Let $\mathscr{C}$ be the set of fundamental cycles such that $C = \bigcup_+ \mathscr{C}$ ($\bigcup_+$ stands for the mod-two sum). It is easily shown that there exists a subset $\{C_i\}_{1 \leq i \leq l}$ of $\mathscr{C}$ such that $e \in C_1$, $e' \in C_l$ and e_i is a common edge of C_i and C_{i+1},

$1 \leqq i < l$. Let (a_i, b_i) be the edge in $G-T$ determining C_i, $1 \leqq i \leqq l$. Let $e(a_i)$, $e(b_i)$ be the edges in T such that $e(a_i) = (a_i, F(a_i))$ and $e(b_i) = (b_i, F(b_i))$; then in each C_i, we have: (i) $e(a_i)\Delta e(b_i)$ and ($e_{i-1}\Delta e(a_i)$ or $e_{i-1}\Delta e(b_i)$) and ($e_i\Delta e(a_i)$ or $e_i\Delta e(b_i)$); or (ii) $e_{i-1}\Delta\ e(a_i)$ and $e_i\Delta e(a_i)$; or (iii) $e_{i-1}\Delta e(b_i)$ and $e_i\Delta e(b_i)$. In any of the above cases, there is a path from e_{i-1} to e_i in G''. In particular, there is a path from e to e_1 and a path from e_{l-1} to e' in G''. Joining all these paths together, we have a path from e to e' in G''. Hence, e and e' belong to the same connected component in G''. □

LEMMA 11.4. *Algorithm Biconnect runs in $O(n/K + \lg^2 n)$ time with $nK(K \geqq 1)$ processors.*

Proof. With $nK(K \geqq 1)$ processors available, step 1 takes $O(n/K + \lg^2 n)$ time (Theorem 5.2). Step 2 takes $O(n/K + \lg n \lg\lg n)$ time (Theorem 8.8). Step 3 can be carried out as follows: Construct an adjacency matrix M'' for G''. For every $e \in E'$, $M''[e, e']$ and $M''[e', e]$ are set to 1 if and only if (i) e' is on the path $[a \ast\!\!\rightarrow \text{HLCA}\ (a)]$ or (ii) (a, b) is in $G-T$ and neither $a \leqq b$ nor $b \leqq a$ in T, where $e = (a, F(a))$ and $e' = (b, F(b))$. Due to $|E'| = O(n)$ and the availability of $\mathbf{F}^+$, testing the above conditions takes $O(n/K)$ time with $nK(K \geqq 1)$ processors (Lemma 3.1). Step 4 takes $O(n/K + \lg^2 n)$ time [3], [4]. Hence, Algorithm Biconnect takes $O(n/K + \lg^2 n)$ time with $nK(K \geqq 1)$ processors. □

12. Finding all separation vertices and determining the biconnectivity of a connected, undirected graph. It is easily verified that if a is not the root r of T, then a is a separation vertex of G if and only if a is the root of $T \cap B_j$ for some j where B_j is a biconnected component of G and that r is a separation vertex if and only if r is the root of at least two distinct $T \cap B'$, $T \cap B''$. As a result, the algorithm for finding the biconnected components can be used to determine the set of all separation vertices of G as follows.

THEOREM 12.1. *The set of separation vertices can be found in $O(n/K + \lg^2 n)$ time with $nK(K \geqq 1)$ processors.*

Proof. First, the set of all biconnected components is determined. This takes $O(n/K + \lg^2 n)$ time with $nK(K \geqq 1)$ processors (Theorem 11.4). Next, the head of each $e \in E'$, head (e), is determined. This obviously takes $O(1)$ time with nK processors. Then the set of all head (e)'s are divided into groups such that those e's belonging to the same biconnected component have their head (e)'s grouped together. This involves sorting and takes $O(\lg n \lg\lg n)$ time with n processors [1]. Finally, the head (e) with the smallest depth in each group is selected, these head (e)'s form the set of separation vertices. r is included in the set if and only if r is selected from two or more groups. This takes $O(n/K + \lg K)$ time with nK processors (Lemma 3.2). □

To determine the biconnectivity of a connected, undirected graph G, we can check the numbers of separation vertices it has. Clearly, G is biconnected if and only if there are no separation vertices. This takes $O(n/K + \lg^2 n)$ time with $nK(K \geqq 1)$ processors.

For completeness, we would like to point out that the algorithm for finding all biconnected components can be used to determine the set of all bridges as well. This is based on the fact that an edge e of G is a bridge if and only if e is a biconnected component of G.

13. Conclusions. The parallel algorithms presented in this paper are optimal for dense graphs except for the problem of finding the lowest common ancestor of vertex pairs in a directed tree and the problem of finding all fundamental cycles in an undirected graph. If an optimal algorithm for finding the lowest common ancestors running in $O(n+q)/nK$ time with $nK(K \geqq 1)$ processors is found, then the algorithm

for finding the fundamental cycles presented in this paper is also improved without any modification. Moreover, this achievement will provide us with an alternate efficient way to compute HLCA (v), $\forall v \in V$ which is crucial in the design of optimal parallel algorithms for the last five problems.

The optimality of our parallel algorithms may suggest that optimal sequential algorithms can be derived from them. As a matter of fact, it has been shown that $O(|V|+|E|)$ time and space sequential algorithms for finding the bridges and biconnected components can be derived [18].

Of all the algorithms presented in this paper, only the algorithm for the lowest common ancestor problem achieves the $O(\lg n)$ time bound. It is therefore intriguing to consider whether there exist $O(\lg n)$ time algorithms for the remaining problems on our SIMD model. No one has yet proven that the $O(\lg^2 n)$ time is a lower bound and this time bound seems unlikely to be surmounted. The difficulty seems to arise from the model we use. In fact, Shiloach and Vishkin [17] have conjectured that the $O(\lg^2 n)$ time bound cannot be breached with a polynomial number of processors on our SIMD model. Recently, Reif managed to design $O(\lg n)$ time probabilistic algorithms for this class of problems [13]. His probabilistic algorithms can be converted into $O(\lg n)$ time parallel algorithms. The resulting algorithms are, however, *nonuniform* in the sense that a different program is needed for each n. Another problem of immediate interest is whether there exist parallel algorithms which are optimal for both dense and sparse graphs. Specifically, they achieve the $O(\lg^2 n)$ time bound using $\lceil m/\lg^2 n \rceil$ processors where m is the number of edges of the given graph.

Finally, we shall point out that although we assume nK, the number of processors available, satisfies the condition $K \geqq 1$ throughout this paper, it is not difficult to extend our results to cases where $0 < K < 1$ if Brent's theorem [19] is used.

Appendix.

```
ALGORITHM DSF (*To find an inverted spanning forest in an undirected graph *)
Stage 1
  {Variable declarations}
  M; array[1 .. n, 1 .. n] of 0 .. 1;
  FR+: array[1 .. 2n-1, 0 .. n-1] of 1 .. n lg n;
  depth: array[1 .. 2n-1] of 0 .. n-1;
  PTR: array[1 .. n lg n] of 1 .. 2n-1;
  DV: array[0 .. lg n, 1 .. n] of 1 .. n;
  rootv: array[1 .. 2n-1] of 1 .. n;
  B: array[1 .. 2, 1 .. n, 1 .. n] of 1 .. n;
  flag: array[1 .. n] of 0 .. 1;
  D, C: array[1 .. n] of 1 .. n;
  phase: 1 .. lg n; startpt: 1 .. 2n-1;
  Step 1: {initialization}
    for all i: 1 ≦ i ≦ n pardo
                    DV[0, i] := D[i] := i; flag [i] := 0
                    dopar;
    for all i: 1 ≦ i ≦ n lg n pardo PTR[i] := 0 dopar;
    for all i: 1 ≦ i ≦ 2n - 1 pardo
                        FR+[i, 0] := FR+[i, 1] := 0;
                        rootv[i] := 0
                        dopar;
```

```
  for all i, j; 1 ≦ i, j ≦ n pardo
                        B[1, i, j] := i; B[2, i, j] := j
                        dopar;
  phase := 0; startpt := 0;
repeat
Step 2(a):
  {Pack all defined rows in each segment together}
  S := {i|flag[i] := 0};
      {Set pointers in array PTR. second is a function extracting the second
       portion of a variable formed by the function concatenation in the
       preceding step.}
  temp := second(sort({concat(flag[i], i)|1 ≦ i ≦ n}));
  PTR[phase * n + 1 . . (phase + 1) * n] := second(sort({concat(temp[i],
  startpt + i)|1 ≦ i ≦ |S|} ∪ {concat(temp[i], 0)||S| < i ≦ n}));
  startpt := startpt + n/2 ** phase;
Step 2(b):
  for all i ∈ S pardo
              j0 := min {j|M[i, j] = 1, j ∈ S}
                  if none then j0 := i;
              C[i] := j0;
              FR+[PTR[phase * n + i], 0] := phase * n + i;
              FR+[PTR[phase * n + i], 1] := phase * n + j0
            dopar;
Step 3(a):
  {Check to see if the set S can be reduced any further;
  if not, then terminate execution}
  if (for all i ∈ S, C[i] = i) then exit;
Step 3(b):
  for all i ∈ S pardo if C[i] = i then flag[i] := 1 dopar;
Step 4:
  for all i ∈ S pardo D[i] := C[i] dopar;
Step 5:
  for j := 1 step 1 until lg n do
  for all i ∈ S pardo C[i] := C[C[i]] dopar;
Step 6(a):
  for all i ∈ S pardo D[i] := min {C[i], D[C[i]]} dopar;
Step 6(b):
  for all i: 1 ≦ i ≦ n pardo D[i] := D[D[i]] dopar;
Step 6(c): {Record the array D[i], 1 ≦ i ≦ n}
  for all i: 1 ≦ i ≦ n pardo
                    if i ∈ S
                    then DV[phase + 1, i] := D[i]
                    else DV[phase + 1, i] := D[DV[phase, i]]
                  dopar;
Step 6(d): {Convert the edge from the smallest-numbered vertex of each
1-tree-loop to a self-loop}
  for all i: D[i] = i
    pardo
      FR+[PTR[phase * n + i], 1] := FR+[PTR[phase * n + i], 0]
    dopar;
```

```
Step 7(a):
  for all i ∈ S pardo
    for all j ∈ S: j = D[j] pardo
      Choose any j0 ∈ S such that D[j0] = j and M[i, j0] = 1
        if none then j0 := j;
      M[i, j] := M[i, j0];
      B[1, i, j] := B[1, i, j0];
      B[2, i, j] := B[2, i, j0]
    dopar
  dopar;
Step 7(b):
  for all j ∈ S: j = D[j] pardo
    for all i ∈ S: i = D[i] pardo
      Choose any i0 ∈ S such that D[i0] = i and M[i0, j] = 1 if none then i0 := i;
      M[i, j] := M[i0, j];
      B[1, i, j] := B[1, i0, j];
      B[2, i, j] := B[2, i0, j]
    dopar
  dopar;
Step 7(c):
  for all i ∈ S pardo M[i, i] := 0 dopar;
Step 8:
  for all i ∈ S pardo if D[i] ≠ i then flag[i] := 1 dopar;
phase := phase + 1;
until (phase ≧ lg n);
Stage 2
Step 1: {Evaluate the array FR+}
  Compute FR+ and depth [i] for 1 ≦ i ≦ 2n − 1.
Step 2:
  phase := phase − 1;
  {Note that at this point, each vertex k left in S is the root of a in-tree recorded
   in the "last" segment}
  for all k: k ∈ S pardo
    rootv[PTR[phase*n + k]] := k
  dopar;
  repeat
    for all i: (phase * n + 1 ≦ i ≦ (phase + 1) * n
               and PTR[i] ≠ 0
               and FR+[PTR[i], (n − 1) − depth [i]]
               ≠ FR+[PTR[i], (n − 1) − depth [i] + 1]);
               {not self-loop}
    pardo {Output all the edges except the one emanating
           from the new root first}
      {Denoting FR+[PTR[i], (n − 1) − depth [i]] mod n
       and FR+[PTR[i], (n − 1) − depth [i] + 1] mod n by
       v0[i] and v1[i] respectively}
      if rootv[PTR[i]] = 0 then
        begin
        T[1, B[1, v0[i], v1!i]] := B[1, v0[i], v1[i]];
        T[2, B[1, v0[i], v1[i]]] := B[2, v0[i], v1[i]];
        end;
```

```
        {Define the roots for the next segment};
        if phase > 0
        then rootv[PTR[DV[phase - 1, B[1, v0[i], v1[i]]]+
          (phase - 1) * n]] := B[1, v0[i], v1[i]];
        {Reverse the edges if necessary}
        if rootv[PTR[i]] ≠ 0
        then for all j: ((n - 1) - depth [i] ≦ j < (n - 1))
          pardo{Denoting FR+[PTR[i], j] mod n and FR+[
          PTR[i], j + 1] mod n by v0[j] and v1[j] respectively}
          T[1, B[2, v0[j], v1[j]]] := B[2, v0[j], v1[j]];
          T[2, B[2, v0[j], v1[j]]] := B[1, v0[j], v1[j]];
          {Redefine the roots as well}
          if phase > 0 then
          begin
            rootv[PTR[DV[phase - 1, B[1, v0[j], v1[j]]]
            + (phase - 1) * n]] := 0;
            rootv[PTR[DV[phase - 1, B[2, v0[j], v1[j]]]
            + (phase - 1) * n]] := B[2, v0[j], v1[j]]
          end
          dopar
      dopar;
    {Pass the roots defined in the current and previous segments to the next
     segment}
    for all i: (phase * n + 1 ≦ i ≦ (phase + 1) * n
              and PTR[i] and rootv[PTR[i]] ≠ 0)
      pardo
        rootv[PTR[DV[phase - 1, rootv[PTR[i]]]+(phase
        -1) * n]] := rootv[PTR[i]]
      dopar;
    phase := phase - 1;
    until (phase < 0);
```

REFERENCES

[1] A. BORODIN AND J. E. HOPCROPT, *Routing, merging and sorting on parallel models of computation*, Proc. 14th ACM Symposium on Theory of Computing, San Francisco, April 1982, pp. 338–344.

[2] A. K. CHANDRA, *Maximal parallelism in matrix multiplication*, IBM Rept., RC 6193, 1975.

[3] F. Y. CHIN, J. LAM AND I-NGO CHEN, *Efficient parallel algorithms for some graph problems*, Comm. ACM, 25 (1982), pp. 659–665.

[4] ———, *Optimal parallel algorithms for the connected component problem*, IEEE Proc. Intel. Conference on Parallel Processing, 1981, pp. 170–175.

[5] D. M. ECKSTEIN AND D. A. ALTON, *Parallel graph processing using depth-first search*, Conferences on Theoretic Computer Science Univ. Waterloo, 1977, pp. 21–29.

[6] S. FORTUNE AND J. WYLLIE, *Parallelism in random access machines*, Proc. 10th Symposium on Theory of Computing, San Diego, CA, 1978, pp. 114–118.

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

[8] J. JA' JA' AND J. SIMON, *Parallel algorithms in graph theory: planarity testing*, this Journal, (1982), pp. 314–328.

[9] D. KNUTH, *The Art of Computer Programming, Vol.* 1., 2nd ed., Addison-Wesley, Reading, MA, 1973.

[10] P. KOGGE AND H. STONE, *A parallel algorithm for the efficient solution of a general class of recurrence equations*, IEEE Trans. Comput., C-22 (1973), pp. 786–792.

[11] F. P. PREPARATA AND J. VUILLEMIN, *The cube-connected cycles: A versatile network for parallel computation*, Comm. ACM, 24 (1981), pp. 300–309.

[12] E. REGHBATI AND D. G. CORNEIL, *Parallel computations in graph theory*, this Journal, 7 (1978), pp. 230–237.

[13] J. REIF, *Symmetric complementation*, Proc. 14th ACM Symposium on Theory of Computing, San Francisco, April 1982, pp. 201–214.

[14] E. M. REINGOLD, J. NIEVERGELT AND N. DEO, *Combinatorial Algorithms: Theory and Practice*, Prentice Hall, Englewood Cliffs, NJ, 1977.

[15] C. D. SAVAGE, *Parallel algorithms for graph theoretic problems*, Ph.D. dissertation, R-784, Dept. Mathematics Univ. Illinois, Urbana, 1977.

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

[17] Y. SHILOACH AND U. VISHKIN, *An $O(\log n)$ Parallel Connectivity Algorithm*, J. Algorithms, 3 (1982), pp. 57–67.

[18] Y. H. TSIN, *A generalization of Tarjan's depth first search algorithm for the biconnectivity problem*, Tech. Rept. TR82-2, Univ. Alberta, Alberta, April 1982.

[19] R. P. BRENT, *The parallel evaluation of general arithmetic expressions*, J. Assoc. Comput. Mach., 21 (1974), pp. 201–206.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.