An Efficient and Fast Parallel-Connected Component Algorithm

YIJIE HAN AND ROBERT A. WAGNER

Duke University, Durham, North Carolina

Abstract. A parallel algorithm for computing the connected components of undirected graphs is presented. Shared memory computation models are assumed. For a graph of e edges and n nodes, the time complexity of the algorithm is $O(e/p + (n \log n)/p + \log^2 n)$ with p processors. The algorithm can be further refined to yield time complexity $O(H(e, n, p)/p + (n \log n)/(p \log(n/p)) + \log^2 n)$, where $H(e, n, p)$ is very close to $O(e)$. These results show that linear speedup can be obtained for up to $p \leq e/\log^2 n$ processors when $e \geq n \log n$. Linear speedup can still be achieved with up to $p \leq n^{\epsilon}$ processors, $0 \leq \epsilon < 1$, for graphs satisfying $e \geq n \log^{(*)} n$. Our results can be further improved if a more efficient integer sorting algorithm is available.

Categories and Subject Descriptors: E.1 [**Data**]: Data Structures—*graphs*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*computation on discrete structures*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*graph algorithms*.

General Terms: Algorithms, Design, Theory.

Additional Key Words and Phrases: Connectivity, optimal algorithms, parallel algorithms

1. *Introduction*

We consider the problem of computing the connected components for a given undirected graph $G = (V, E)$. We present a parallel algorithm for this problem on shared memory parallel computation models (PRAM) [4, 22]. On a PRAM model, each memory cell can be accessed by any processor, thus the parallelism of a problem could hopefully be exploited to its maximum. Three shared memory models can be classified according to the ways they resolve read/write conflicts. They are the Concurrent-Read Concurrent-Write (CRCW) model, the Concurrent-Read Exclusive-Write (CREW) model, and the Exclusive-Read Exclusive-Write (EREW) model [22]. The CREW model is weaker than the CRCW model and the EREW model is the weakest one among three.

In the parallel environment, two important measures for algorithms are their time complexity (also called depth) and speedup. Time complexity T_p[1] is the time

[1] If not otherwise indicated, p, n, and e will denote the number of processors, the number of nodes, and the number of edges, respectively.

The research was partially supported by the National Science Foundation (NSF) under grant MCS 82-02610.

Authors' present addresses: Y. Han, Department of Computer Science, University of Kentucky, Lexington, KY 40506; R. A. Wagner, Department of Computer Science, Duke University, Durham, NC 27706.

units taken by the algorithm with p processors. The speedup is defined as $S = T_1/T_p$, where T_1 is the time complexity of the fastest serial algorithm for the same problem. It is obvious that $S \leq p$. Algorithms with small time complexity are regarded as fast while algorithms with speedup S close to p are regarded as efficient algorithms. When $T_p = \Theta(T_1/p)$, the algorithm achieves linear speedup with p processors. A linear speedup algorithm is an optimal algorithm.

The problem of computing in parallel the connected components for an undirected graph has been studied by many researchers [3, 5, 7–9, 11, 12, 15, 17–19, 21]. The paper of Quinn and Deo gives a good survey [16]. Hirschberg [7] first noticed that n^2 processors are enough to obtain $O(\log^2 n)^2$ time complexity for computing the connected components. The first optimal algorithm on the CREW model using up to $(n/\log n)^2$ processors was obtained for dense graphs by Chin et al [5]. The time complexity of their algorithm is $O(n^2/p + \log^2 n)$, with p processors for a graph of n nodes. Shiloach and Vishkin's CRCW algorithm [21] has time complexity $O((e \log n)/p + \log n)$, with p processors for a graph of n nodes and e edges. The work of Savage and Ja'Ja [19], Kruskal et al. [11] show improvement for sparse graphs. The results of Kruskal et al. [11] showed that it is possible to obtain optimal parallel algorithm for sparse graphs. Their algorithms have time complexity of $O(e/p + (n \log p)/p + p^{1+\epsilon})$ for the EREW model and $O(e/p + (n \log p)/p + p \log p)$ for the CREW model. Algorithms for minimum spanning forest can be used for finding the connected components. These algorithms have time complexity $O((e \log n)/p + \log^2 n)$ [11, 13] on the CREW model. No connected component algorithm is known to us that achieves linear speedup with poly-log time complexity for sparse graphs.

The difficulty of obtaining fast and efficient parallel connected component algorithms for sparse graphs seems to lie in choosing the right nodes to form supernodes, at least when one tries to use the graph contraction scheme outlined in the algorithm presented by Hirschberg et al. [9]. It is noted that, when the nodes are combined to form supernodes, the number of edges is not necessarily reduced by a large quantity. In the worst case, the number of edges eliminated is about the same as the number of nodes eliminated. Those uneliminated edges constitute inefficiency if they are examined again and again. It is not difficult to show that, in a graph contraction scheme, if the right nodes are chosen to be combined, the number of edges can be reduced significantly, leading to an efficient and fast parallel algorithm for sparse graphs. However, it seems that choosing the right nodes is still a very difficult problem if linear speedup is demanded. This is probably the reason that no algorithm has attempted an "optimal" choice of nodes.

In this paper we first present a connected component algorithm with time complexity $O(e/p + (n \log n)/p + \log^2 n)$. This algorithm is a CREW algorithm, that is, it allows concurrent read but not concurrent write. It achieves linear speedup with up to $e/\log^2 n$ processors for graphs satisfying $e \geq n \log n$. We then refine our algorithm to achieve time complexity

$$O\left(\frac{H(e, n, p)}{p} + \frac{n \log n}{p \log(n/p)} + \log^2 n\right).$$

Define $P_\alpha(1, k) = \alpha^k$, $P_\alpha(i, k) = \alpha^{P_\alpha(i-1,k)}$, and $h_\alpha(k, a) = \min\{i \mid P_\alpha(i, k) \geq a\}$, then function H can be expressed as

$$H(e, n, p) = e \cdot h_2\left(\frac{e}{n}, \log \frac{n}{p \log p}\right).$$

[2] Logarithms are to the base 2 if not specified.

$H(e, n, p)$ is very close to $O(e)$. Function h_2 is similar to the G function used to describe the time complexity of the UNION–FIND algorithm [1]. For example, if $e \geq n \log^{(*)} n$,[3] then $H(e, n, p) = O(e)$. The time complexity can also be expressed as

$$O\left(\frac{H(e, n, p)}{p} + \sum_{i=1}^{\log n} T_{\text{sort}}(2^i, p) + \log^2 n\right),$$

where $T_{\text{sort}}(n, p)$ is the time complexity for sorting n integers in the range $\{1, 2, \ldots, n\}$ with p processors. Thus, the algorithm can be further improved if a more efficient integer sorting algorithm is available.

Our algorithm does not choose the optimal nodes to combine. It simply combines nodes that are found to be connected. The efficiency of the algorithm comes from the fact that the algorithm does not look at more edges than is necessary. In fact, the edge extraction strategy of the algorithm is so efficient that the bottleneck of the algorithm now becomes the integer sorting step, which introduces the term $O((n \log n)/(p \log(n/p)))$, and the sparse graph-reduction step, which introduces the term $O(H(e, n, p)/p)$.

2. *Basics*

Let $G = (V, E)$ ($G = \langle V, E \rangle$) denote an undirected (directed) graph with node set $V = \{1, 2, \ldots, n\}$ and edge set E that is a set of unordered (ordered) pairs drawn from $V \times V$. For an undirected graph, there is an edge connecting nodes i and j iff $(i, j) \in E$. G is connected iff for every pair i, j, there is a path between i and j. A connected component of G is a maximal connected subgraph of G. We only talk about the weak connectivity of directed graphs in which a directed graph is treated as an undirected graph by viewing the edges as unordered. We assume the graph is represented as an array of edges, with all edges that are adjacent to a node stored sequentially in the array (not necessarily sorted). We also assume that all edges (i, i), $1 \leq i \leq n$, are present in the input list, so that all the isolated nodes are represented. This list of edges is initially placed in an array E. Figure 1 shows a graph and its representation.

We also need the notion of multigraphs. A multigraph $M = (S, G)$ consists of a graph G and a supernode set S. G, as defined before, is an undirected graph (V, E) or a directed graph $\langle V, E \rangle$. S is a partition of V. Figure 2 gives a multigraph. An edge (i, j) ($\langle i, j \rangle$) is an inside edge if both i and j are in the same supernode. Otherwise, the edge is an outgoing edge. A supernode is isolated if it has no outgoing edges.

The notion of tree-loop was introduced in [9]. A k-tree-loop, $k \geq 0$, is a directed graph in which every node has outdegree 1 and there is exactly one cycle of length $k + 1$. A tree-loop is a k-tree-loop for some k. The root of a 0-tree-loop is the node v such that edge (v, v) is the cycle of length 1. A star is a 0-tree-loop with all nodes in the tree-loop pointing to the root. A tree-loop graph is a collection of tree-loops.

LEMMA 1. *Let $G = \langle V, E \rangle$ be a directed graph such that each node's outdegree is exactly one. Then G is a tree-loop graph.*

PROOF. Suppose G' is a weakly connected component of G. By induction on the number of nodes it is easily shown that there is at least one cycle in G'. Suppose there is more than one cycle in G'. Consider the path connecting any two cycles.

[3] Define $\log^{(1)} n = \log n$, $\log^{(i)} n = \log(\log^{(i-1)} n)$ and $\log^{(*)} n = \log^{(i)} n$ for some i.

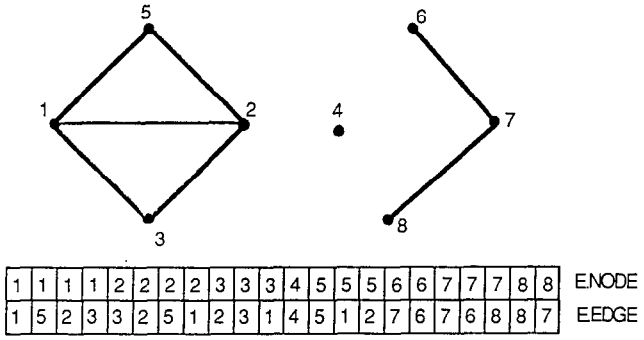

FIG. 1. A graph and its representation.

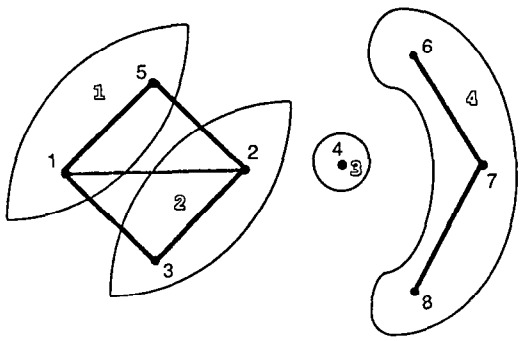

FIG. 2. A multigraph.

Using induction on the length of the path, it can be shown that such a path must be broken by the fact that every node's outdegree is one. □

The following CREW algorithm will compute the weakly connected components for a tree-loop graph. The algorithm uses a strategy presented by Lev et al. [14]. Initially, array F is the input tree-loop graph with nodes i and edges $\langle i, F[i]\rangle$, $1 \leq i \leq n$. When the algorithm finishes, each input tree-loop has been transformed into a star. The root of the star is the smallest node in the cycle of the input tree-loop.

```
CONNECTED_TREELOOP(F)
  begin
    for all i: 1 ≤ i ≤ n do in parallel
      begin
        S[i] = F[i];
        for k = 1 to log n do[4]
          begin
            if F[S[i]] < F[i] then F[i] := F[S[i]];
            S[i] := S[S[i]];
          end
        for m := 1 to logn do
          F[i] := F[F[i]];
      end
  end
```

[4] In this paper, we assume the results of division and logarithm are integers, as this will not affect the correctness of our algorithms.

THEOREM 1. *CONNECTED_TREELOOP( ) correctly computes the weakly connected components for any tree-loop graph of n nodes in $O((n \log n)/p + \log n)$ time with p processors.*

PROOF. After s iterations of the loop indexed by k, $F[i]$ is the smallest node at directed distance 2^s or less (but greater than 0) from i. Thus, at the end of this loop, array F represents a collection of 0-tree-loops with the root being the smallest node in the cycle of the original tree-loop. It follows that after $\log n$ iterations of the loop indexed by m, all nodes in a tree-loop will point to the smallest node in the cycle of the tree-loop.

The total number of operations involved in the algorithm is $O(n \log n)$. Timing $O(\log n)$ can be achieved when n processors are available. Thus, the time complexity is $O((n \log n)/p + \log n)$. □

3. *The Parallel Connected Component (PCC) Algorithm*

In this section we present our main algorithm. The optimal connected component algorithm for dense graphs [5] was obtained by reducing the number of nodes by at least one-half in $O(n^2/p + \log n)$ time. This scheme does not work for sparse graphs, since when the number of nodes is reduced by half, the number of edges is not necessarily reduced by half. We show a strategy for reducing the number of nodes and edges such that time complexity of

$$O\left(\frac{e}{p} + \frac{n \log n}{p} + \log^2 n\right)$$

is attained.

The PCC( ) algorithm has two nested loops. The outside loop is called the node loop and the inside loop is called the edge loop. The node loop requires $O(\log n)$ iterations. Each iteration of the node loop reduces the number of supernodes by at least a factor of $\frac{1}{4}$ (some may be combined with others, while some supernodes are isolated and need not be considered at the next iteration of the node loop). Thus, after $O(\log n)$ iterations of the node loop, the connected components will be obtained.

Consider now the edge loop. Suppose we have a multigraph consisting of m supernodes, where each supernode contains some number of nodes. We want to find outgoing edges for $m/2$ supernodes. We have p processors available. Suppose $p \log p \geq n$. In each iteration of the edge loop, we extract $p \log p$ edges, $(p \log p)/m$ edges from each supernode, for examination (subroutine GETEDGE( )). For a supernode, if any one of the $(p \log p)/m$ edges is an outgoing edge, then this supernode will be labeled OK. For those supernodes not labeled OK, all the $(p \log p)/m$ edges must be edges within the supernode and these edges can thus be deleted (subroutine REDUCEEDGE( )). We must distinguish between two situations here. One situation is that all the edges of a supernode have been examined. This shows that the supernode is isolated. We also label it OK. Another situation is that there are edges of the supernode that have not been looked at yet. Thus each supernode s will be left in one of the three states:

(a) There is an outgoing edge found for s. s is labeled OK.
(b) No outgoing edge is found for s. All the edges incident on s have been deleted. s is an isolated supernode. It is labeled OK.
(c) No outgoing edge is found for s. There are edges incident on s that have not been checked by PCC( ) yet. s is not labeled OK.

The edge loop will continue its iteration until the number of supernodes labeled OK exceeds $m/2$. When $p \log p < n$, we extract n/m edges from each supernode and go through the same procedure as stated above.

After at least $m/2$ supernodes have been labeled OK, PCC( ) will choose one outgoing edge for each supernode in state (a). The outgoing edge chosen for a supernode v is oriented such that v will be the tail of the edge. The algorithm CONNECTED_TREELOOP( ) is called to combine the supernodes to form new supernodes. It is not difficult to see that the number of nonisolated supernodes remaining is at most $3m/4$. (The worst case arises when all OK supernodes are combined in pairs.)

Before giving our algorithm, the data structure employed in the algorithm will be discussed briefly. Array $E[1 \cdot\cdot e]$ is the input list of edges. It has two components, *E.NODE* and *E.EDGE*, as shown in Figure 1. Array $N[1 \cdot\cdot n]$ stores relevant information concerning nodes. $N[i]$.*COUNT* is the number of remaining unexamined edges adjacent to node i. $N[i]$.*FIRSTEDGE* is the address of the first remaining edge adjacent to node i in array E. $N[i]$.*SNODE* is the connected component membership of node i. Initially, $N[i]$.*SNODE* $= i$. When the algorithm finishes, $N[i]$.*SNODE* is the component number. $M[1 \cdot\cdot n]$ is a sorted sequence of nodes such that nodes that are grouped into a supernode are listed consecutively. $M[i]$.*NODE* is the node name. $M[i]$.*SNODE* is the supernode number. Array G is used for processing supernodes. $G[i]$.*COUNT* is the number of nodes the supernode i contains. *G.SNODE* is the supernode name. *G.ISNODE* flags isolated supernodes. $G[i]$.*GROUP* is the new supernode the current supernode i will be combined into. Array *EE* is used to store the edges extracted from array E and to allow processing of these edges. *EE.NODE* and *EE.EDGE* will contain edges. *EE.SNODE* indicates *EE.NODE*'s group membership, that is, to which supernode the nodes belong. *EE.COUNT* is used to count the number of edges actually extracted from each supernode. At the beginning of each node loop, some edges are extracted from array E to array *EE*. Since arrays N and M indicate where these edges could be found in E, these edges can be extracted easily. If the edges extracted from a supernode are all inside edges, the algorithm merely updates array *N.COUNT* and *N.FIRSTEDGE* to indicate that these edges are deleted. The number of supernodes labeled OK is also counted by summing the number of supernodes that either have outgoing edges or are newly isolated supernodes.

Algorithm PCC( ) is shown below. The algorithm uses several subroutines that are tedious and provide few insights. The operations of these subroutines are outlined here. Details of these subroutines can be found in the appendix.

GETEDGE(k) copies up to k edges touching each supernode into array *EE*. It first determines how many edges touching each node in supernode s to copy, by computing separately for each supernode s the partial sum of the degree of each node belonging to s (the partial sum of a sequence of numbers $(a_1, a_2, \ldots, a_n)$ is $(b_1, b_2, \ldots, b_n)$, where $b_i = \sum_{j=1}^{i} a_j$). GETEDGE( ) then allocates processors to edges in E to perform the copying operation.

REDUCEEDGE(k) operates on the compact list of edges extracted by GETEDGE( ). REDUCEEDGE( ) determines, in parallel, whether the nodes touched by each edge belong to the same supernode or not. If not, the supernode containing the tail of the edge will be marked OK. This OK label will first be placed on edges. Then, in $O(\log p)$ time, each supernode can determine whether there is an outgoing edge labeled OK among the k edges extracted in this pass.

COMBINENODES( ) will combine the supernodes found to be connected. It uses CONNECTED_TREELOOP( ) to determine supernode adjacency. It uses a

sorting algorithm to sort the connected supernodes together. It then renames the supernodes thus keeping the integers used to represent the supernodes in a small range and rearranges the nodes in array M to denote to which new supernode each node belongs.

```
PCC(E)
  begin
    INITIALIZATION( );
    for v := 1 to log_{4/3} n do   /* Node loop. */
      begin
        oknode := 0;
        if p log p > n then k := (p log p)/m;
        else k := n/m;
        while oknode < m/2 do   /* Edge loop. */
          begin
            /* Extracting edges. */
            GETEDGE(k);
            /* Delete inside edges and count the number of supernodes labeled OK.
            */
            oknode := REDUCEEDGE(k);
          end
        /* Form new supernodes, reducing the number of supernodes by at least a factor
           of 1/4.
        */
        m := COMBINENODES(k);
      end
  end
```

THEOREM 2. *Algorithm PCC( ) correctly computes the connected components of undirected graphs in* $O(e/p + (n \log n)/p + \log^2 n)$ *time units.*

PROOF. The correctness follows from the fact that each iteration of the node loop will reduce the number of supernodes by at least a fraction of $\frac{1}{4}$; thus, if any two nodes are connected they will eventually be combined within $\log_{4/3} n$ iterations of the node loop.

Note that if x iterations of the edge loop are required during one iteration of the node loop, then at the time when the $(x-1)$th iteration finishes and the xth iteration has not yet been executed, there are more than $m/2$ supernodes not labeled OK. When $p \log p \geq n$, this means at least $((x-1) \cdot p \log p)/2$ edges have been deleted. Let the overall number of iterations of the edge loop be L, we have

$$\frac{(L - \log_{4/3} n) \cdot p \log p}{2} \leq e,$$

where $\log_{4/3} n$ is the number of iterations of the node loop. We obtain $L \leq (2e/p \log p) + \log_{4/3} n$. It is not difficult to see that, when $p \log p \geq n$, subroutines GETEDGE( ) and REDUCEEDGE( ) take $O(\log p)$ time, since only $p \log p$ edges are processed and $O((p \log p)/p + \log p) = O(\log p)$. It follows that the total time for the edge loop is $O(e/p + \log n \log p)$.

Suppose $p \log p < n$. Let x_i denote the number of iterations of the edge loop executed within the ith iteration of the node loop. The time spent will be $O(x_i \cdot n/p)$, and the number of edges deleted is at least $(x_i - 1) \cdot n/2$. Summing

over all iterations of the node loop, we have

$$\frac{\sum (x_i - 1) \cdot n}{2} = \frac{\sum x_i \cdot n}{2} - \frac{\sum n}{2} \geq \frac{\sum x_i \cdot n}{2} - \frac{n \log_{4/3} n}{2},$$

This quantity is bounded by e. Thus, $\sum x_i \cdot n \leq 2e + n \log_{4/3} n$. The timing thus obtained is

$$O\left(\frac{e}{p} + \frac{n \log n}{p}\right).$$

Subroutine COMBINENODES( ) has two parts. One part updates the node's membership. In each iteration of the node loop, this part takes $O(n/p + \log p)$ time; thus, a total of $O((n \log n)/p + \log n \log p)$ time for $O(\log n)$ iterations. Another part renames the supernodes. We execute subroutine CONNECTED_TREELOOP( ). We also sort these supernodes in each iteration of the node loop. Both operations take at most $O((m \log m)/p + \log m)$ time. This is at most $O((n \log n)/p + \log n)$ the first time they are executed, since the AKS sorting network [2] can be used here, (Note that a parallel integer sorting algorithm suffices, we will talk more about this later). The subsequent executions of CONNECTED_TREELOOP( ) and sorting are performed on a number of nodes that is geometrically decreasing. Thus, the overall time for COMBINENODES( ) is $O((n \log n)/p + \log^2 n)$.

Summarizing the above discussion and noting that $\log p \leq 2 \log n$, we obtain time complexity

$$O\left(\frac{e}{p} + \frac{n \log n}{p} + \log^2 n\right)$$

for algorithm PCC( ). □

We have some remarks concerning the algorithm PCC( ). We have used the AKS sorting network [2] for the purpose of sorting together nodes that belong to the same supernode. Although this scheme does not require more than $O(n \log n)$ time for sorting n nodes, the multiplicative constant of the time complexity of the sorting network is huge. To avoid this disadvantage, one might use a parallel integer sorting algorithm. We have considered parallel integer sorting in [23]. In [23], it is shown that time complexity $O(n \log n/\epsilon p \log(n/p))$, $0 < \epsilon \leq 1$, can be achieved for sorting n integers in the range $\{1, 2, \ldots, n\}$ with $p \leq n/2$ processors.

In each edge loop we extracted about $p \log p$ edges if $p \log p > n$. This is based on the observation stated below. On the one hand, we want to extract as few edges as possible, since the fewer edges we examine at a time, the more serialism we preserve. This serialism may be advantageous in preserving the efficiency of processor usage. On the other hand, extracting too few edges will increase the run time of the algorithm. In the case of the connected component algorithm, the quantity $p \log p$ is a good choice, for we know we could process these edges in minimum time (in $O(\log p)$ time), whereas processing fewer edges will lead to a longer run time.

We chose to continue the edge loop until the number of supernodes labeled OK exceeds a fraction of $\delta = \frac{1}{2}$ of the current supernodes. Actually δ can be any number in the range $0 < \delta < 1$. When $\delta \to 0$ or $\delta \to 1$, the multiplicative constant in the

time complexity will become very large. We put no effort into choosing an optimal δ, since this would only affect the time complexity to within a constant factor.

4. *Refinement*

In this section, we are going to modify PCC( ) to improve its efficiency for the case $e = o(n \log n)$. Our aim is to improve the term $(n \log n)/p$ in the time complexity for PCC( ), which dominates $\log^2 n$ when $p \log p < n$. The refined algorithm takes only

$$O\left(\frac{H(e, n, p)}{p} + \frac{n \log n}{p \log(n/p)} + \log^2 n\right)$$

time. This bound can still be improved if a more efficient integer sorting algorithm is available.

As will be seen, the refinement relies on a fast integer sorting algorithm, a fast matching algorithm for linear graphs and a graph-reduction technique.

The integer sorting algorithm presented in [23] takes

$$O\left(\frac{\log m}{\log((n/p) + \log p)}\left(\frac{n}{p} + \log p\right)\right)$$

time for sorting n integers in the range $\{1, \ldots, m\}$ with $p \leq n/2$ processors. When $m = n$, the result of

$$O\left(\frac{\log n}{\log((n/p) + \log p)}\left(\frac{n}{p} + \log p\right)\right)$$

is reported in [11] and [23]. Since we only consider the situation $p \log p < n$, the time complexity for sorting will become $O((n \log n)/(p \log(n/p)))$.

An a-matching ($a > 1$) of a graph $G = \langle V, E\rangle$, where $V = \{1, 2, \ldots, n\}$, is defined to be a set $F \subseteq E$ such that no two edges in F are incident on the same vertex and $|F| \geq |V|/2a$. We present a parallel algorithm for finding a 2-matching for linear graphs. A digraph G is termed linear iff the indegree and outdegree of each node in G is at most one. This algorithm uses a scheme in [6] and [23]. First, all the edges will be divided into $2 \log n$ groups. Edge $\langle a, b\rangle$ will be in group $2k - a_k$, where

$$k = \min\{i \mid \text{the } i\text{th bit of } a \textit{ XOR } b \text{ is } 1\},$$

XOR is the exclusive-OR operation, bits are counted from the least significant bit starting at 1, a_k is the kth bit of a. These groups have properties:

(a) Each edge belongs to one and only one group.
(b) For any group, if two different edges $\langle a, b\rangle \in G$ and $\langle c, d\rangle \in G$, then a, b, c, d are all distinct.

The function $g(\langle a, b\rangle)$ that gives the group membership of $\langle a, b\rangle$ can be computed in $O((n/p) + \log p)$ time (see [6] and [23]). Next, these edges can be sorted into groups in $O((n/p) + \log p)$ time, and thus edges can be examined group

by group [6] and [23]. The 2-matching algorithm can now be expressed as:

```
MATCHING(E)
/* Assume edges have been sorted by groups. */
  begin
    for all nodes i do in parallel:
      DONE[i] := false;
    for k := 1 to 2logn do
      for all edges ⟨a, b⟩ in group k do in parallel
        begin
          if DONE[a] = false && DONE[b] = false then
            begin
              DONE[a] := true;
              DONE[b] := true;
            end
          else delete(⟨a, b⟩);
        end
  end
```

When the algorithm finishes, the remaining undeleted edges constitute a matching. If a node is isolated (both its incoming and outgoing edges are deleted), then it is noted that both its predecessor and successor are not isolated. Thus, a 2-matching is obtained. MATCHING( ) takes $O(n/p + \log n)$ time.

Let us take a close look at tree-loop graphs. We use an array $F[1 \cdot\cdot n]$ to represent a tree-loop graph with n nodes. $F[i]$ represents edge $\langle i, F[i] \rangle$. If we want to calculate the indegree for each node, we can use an integer-sorting algorithm. This will take $O((n \log n)/p \log (n/p))$ time with p processors. By applying integer sorting to a tree-loop, we can build array OUT with $OUT[i] = j$ iff $\langle i, j \rangle$ is an edge in the tree-loop, $INDEGREE[i]$ that gives the indegree for node i, and array IN which, for each node, gives the nodes pointing to it. All of these can be done in $O((n \log n)/p \log(n/p))$ time.

Based on the above observation, we can delete edges in a tree-loop graph G to produce graph G_1. G_1 is a subgraph of G such that for each node i in G, if the indegree of i is greater than 1 in G, then all but one of the edges $\langle j, i \rangle$ $(j \neq i)$ will be deleted in forming G_1. The resulting graph G_1 is a directed graph with every node having indegree and outdegree no more than 1. A tree-loop in G can generate some isolated nodes when G is transformed into G_1. All these isolated nodes must be leaves in G (nodes with indegree 0). These isolated nodes will be marked inactive to produce G_1. For G_1 the matching algorithm can be applied. The execution of this algorithm will find matching nodes for at least $\frac{1}{2}$ of the nodes in G_1. After this is done, isolated nodes in G_1 can be attached to their fathers and those inactive leaves in G can then be attached to their original fathers. Thus, every node in G will be combined with some other node.

In summary, this describes a subroutine (which we call TREELOOP( )) that combines each of the nodes in a tree-loop with some other node. Observe that we actually do not require a subroutine such as CONNECTED_TREELOOP( ) to compute the connected components for tree-loop graphs. For our purpose, TREELOOP( ) is enough, since it ensures that the number of nonisolated nodes in a tree-loop is reduced by at least half. CONNECTED_TREELOOP( ) can be replaced by TREELOOP( ). TREELOOP( ) takes only $O((n \log n)/p \log (n/p))$ time, dominated by the time for integer sorting.

We would like to point out here that a linked-list prefix algorithm such as the one in [6] (which takes $O(n/p + G(n) \log n)$ time where $G(n) =$

$\min\{i \mid \log^{(i)} n < 2\}$, the CRCW version of the linked-list prefix algorithm takes $O(n/p + \log n)$ time) or even the one in [10] (which takes $O((n \log n)/p \log(n/p))$ time) is good enough for the purpose of finding a 2-matching. The time complexity for our connected component algorithm will not be affected if either one of the above prefix algorithms is used. Our objective in designing the matching algorithm is to transfer the bottleneck of our connected component algorithm to the problem of integer sorting. Since the matching algorithm takes $O(n/p + \log p)$ time, it will not become a bottleneck even if the current integer-sorting algorithm is improved. We also believe that our matching algorithm will be a useful building block for other parallel algorithms.

The third technique that will be used to improve PCC( ) is a graph-reduction technique. This is to collapse each supernode into one node and rename all the edges before continuing the execution of PCC( ). This technique was used by Chin et al. [5] to obtain the optimal parallel connected component algorithm for dense graphs. We use this technique in a somewhat different way to reduce the timing of PCC( ). Since we have array M (see Appendix) to indicate which supernode contains which node, the graph reduction can be carried out in $O(e/p + \log p)$ time.

Now we outline the improvements over algorithm PCC( ), using the fast integer-sorting algorithm, the fast matching algorithm and graph-reduction technique. The refined algorithm will be termed PCCR( ). Recall all these improvements are made under the condition $p \log p < n$. The main strategic change in PCCR( ) over PCC( ) is that graph reduction is applied at certain appropriate times.

In PCCR( ) iteration i of the node-loop reduces the number of nodes remaining from n_i to n_i/α. Let $1 < \alpha < 2$. We only consider the first $\log_\alpha n/(p \log p)$ iterations of the node loop. There are no more than $p \log p$ nodes left after these iterations of the node loop. Thus, the time needed after $\log_\alpha n/(p \log p)$ iterations of the node loop will be $O(e/p + \log^2 n)$.

At the beginning of the node loop in PCCR( ) m edges are extracted, one edge from each supernode, for examination. The edge loop is iterated until the number of supernodes labeled OK exceeds $m/2$. Suppose x_i iterations of the edge loop are executed in the ith iteration of the node loop. The time spent is then $\sum(1/\alpha)^i(x_i \cdot n)/p$ and at least $\sum(1/\alpha)^i((x_i - 1) \cdot n)/2$ edges are deleted. The number of edges deleted is bounded by e. Thus, we have $\sum(1/\alpha)^i x_i \cdot n \le 2e + C(\alpha)n$, where $C(\alpha)$ is a constant depending on α. The timing, therefore, is $O(e/p)$. (This strategy could be used in PCC( ) instead of in this refinement. However, if this strategy is used in PCC( ), the already complicated subroutines GETEDGE( ) and REDUCEEDGE( ) (see Appendix) would become even more difficult to read. The time complexity of PCC( ) would not be improved because this strategy only eliminates the term $(n \log n)/p$ for edge extraction. When PCC( ) is well understood, it is not difficult to see that extracting m edges when $p \log p < n$ will work.)

The term $(n \log n)/p$ in the time complexity now comes solely from the subroutine COMBINENODES( ) (see Appendix), since the total time needed for the edge loop is reduced to $O(e/p + \log^2 n)$. Two portions of COMBINENODES( ) induce time complexity $O((n \log n)/p)$. The first part includes the subroutines CONNECTED_TREELOOP( ) and SORT( ). As we have pointed out, CONNECTED_TREELOOP( ) can be replaced by TREELOOP( ) and an integer sorting algorithm can be used for SORT( ). Thus the time complexity for both subroutines can be reduced to $O((n \log n)/p \log(n/p))$. In fact, if a more efficient integer sorting algorithm is available, the time complexity can be improved further. The second portion of COMBINENODES( ) that results in the term $(n \log n)/p$ is the section performing node rearrangement. This portion is identified by

the statements

```
for all i: 1 ≤ i ≤ n do in parallel
  begin
    . . .
    . . .
  end
```

in COMBINENODES( ). One execution of this portion takes only $O(n/p + \log p)$ time. Since there are $\log_\alpha n$ iterations of the node loop, a total time complexity $O((n \log n)/p + \log^2 n)$ results. The graph reduction technique is used to reduce the time complexity for this portion of COMBINENODES( ). The method is: reduce the graph whenever the timing is accumulated to e/p. The first reduction will occur after e/n iterations of the node loop, because the timing accumulated is $O(n/p \cdot e/n) = O(e/p)$. This reduction will reduce the number of nodes to $(1/\alpha)^{e/n} \cdot n$. The second reduction will occur $(e/n)\alpha^{e/n}$ iterations of the node loop later, since the timing accumulated is $O((1/\alpha)^{e/n}(n/p) \cdot (e/n)\alpha^{e/n}) = O(e/p)$. And so on. It can be shown that the number of iterations of the node loop executed before the $(i + 1)$th reduction is greater than $P_\alpha(i, e/n)$. Once the number of iterations of the node loop exceeds $\log_\alpha(n/(p \log p))$, the number of nodes left is less than $p \log p$, and we are done. So we obtain $P_\alpha(i, (e/n)) \geq \log_\alpha(n/(p \log p))$. Thus, $h_\alpha(e/n, \log_\alpha(n/(p \log p)))$ is enough for i. Notice $h_\alpha(i, j) = \Theta(h_2, (i, j))$. Since there are i graph reductions in total and time between the jth and $(j + 1)$th $(j < i)$ reduction is $O(e/p)$, the time complexity for the second portion of COMBINENODES( ) is reduced to $O(H(e, n, p)/p + \log^2 n)$.

In summary, we have obtained a CREW connected component algorithm PCCR( ) with time complexity $O(H(e, n, p)/p + (n \log n)/(p \log(n/p)) + \log^2 n)$.

5. *Conclusions*

The parallel-connected component algorithm presented in this paper improves previous results. The known sequential algorithm for computing connected components uses depth-first search [1]. It seems that the depth-first search scheme is difficult to parallelize. Previous results and ours provide an alternative approach to the connected component problem. Our experience indicates that it is important to preserve serialism while maintaining minimum-time complexity in the design of parallel algorithms.

Appendix

In this appendix, we present the details of the routines used in our main algorithm PCC( ). For a clear exposition, a strong type of synchronization is assumed in the presentation of these routines. If both concurrent read and concurrent write are involved in one statement, it is assumed that concurrent read is finished before any write can take place. Such synchronization requirement can be removed by introducing temporary variables.

We need several basic routines. All these routines GROUP_PARTIALSUM (G, A), TAKING(A), and PARTIALSUM(A) take $O(n/p + \log p)$ time for problem size n with p processors. Their function can be summarized as follows:

GROUP_PARTIALSUM(G, A) computes the partial sum for every group of elements in array A. Array G is used to denote each element's group membership. Elements in one group are located sequentially in A. Suppose group k consists of elements from $A[i]$ to $A[j]$. GROUP_PARTIALSUM(G, A) will compute $\sum_{l=i}^{h} A[l]$, $i \leq h \leq j$, and deposit it in $A[h]$. Figure 3 shows the result of this

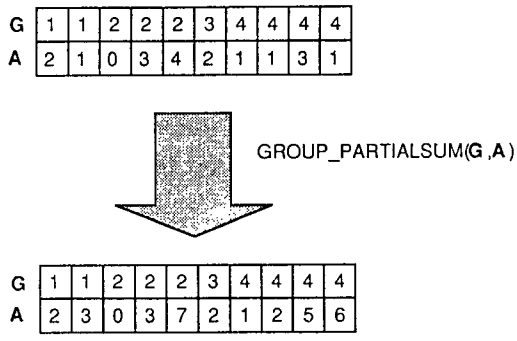

FIG. 3. The effect of GROUP_PARTIALSUM( ).

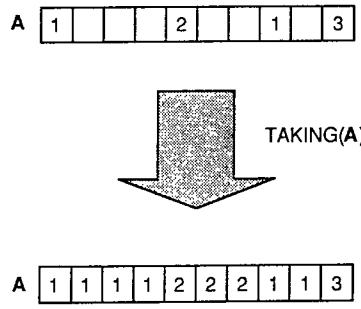

FIG. 4. The effect of TAKING( ).

operation. PARTIALSUM(A) is equivalent to GROUP_PARTIALSUM(G, A) except it assumes that there is only one group. TAKING(A) is a routine that, for each nonblank symbol, puts this symbol in all the following blank cells up to the next nonblank cell. Figure 4 illustrates this operation. The implementation of these routines on shared memory computation models is well understood. Thus, we omit the details of these routines here.

The subroutine INITIALIZATION( ) will initialize arrays N, M and variables m and *isolated*.

```
INITIALIZATION( )
  /* Initialize arrays N, M, and variables m and isolated. */
  begin
    for all j: 1 ≤ j ≤ e do in parallel
      begin
        TEMP[j] := 1;
        GROUP_PARTIALSUM(E.NODE, TEMP);
        if E[j]. NODE ≠ E[j + 1].NODE then
          begin
            N[E[j].NODE].COUNT := TEMP[j];
            N[E[j].NODE].FIRSTEDGE := j − TEMP[j] + 1;
          end
      end
    for all i: 1 ≤ i ≤ n do in parallel
      begin
        N[i].SNODE := i;
        M[i].NODE := i;
        M[i].SNODE := i;
      end
    m := n;
    isolated := 0;
  end
```

The main algorithm consists of many stages. In each stage, for some k, k edges need to be extracted from each supernode for examination. The following routine shows how this task is accomplished.

```
GETEDGE(k)
  begin
    for all i: 1 ≤ i ≤ n do in parallel
      begin
        /* Locate edges to be extracted. */
        TEMP[i] := N[M[i].NODE].COUNT;
        GROUP_PARTIALSUM(M.SNODE, TEMP);
        TEMP[i] := TEMP[i] − N[M[i].NODE].COUNT;
        /* If TEMP[i] ≥ k then the edges adjacent to node M[i].NODE
             need not be extracted.
        */
        if TEMP[i] ≥ k then TEMP[i] := NULL;
      end
    /* Find cells of array EE in which the edges adjacent to
       node M[i].NODE will be stored.
    */
    for all j: 1 ≤ j ≤ k·m do in parallel
      begin
        EE[j].NODE := 0;
        EE[j].SNODE := 0;
        EE[j].EDGE := 1;
      end
    for all i: 1 ≤ i ≤ n do in parallel
      begin
        if TEMP[i] ≠ NULL then
          begin
            EE[k*(M[i].SNODE − 1) + TEMP[i] + 1].NODE := M[i].NODE;
            EE[k*(M[i].SNODE − 1) + TEMP[i] + 1].SNODE := M[i].SNODE;
          end
          /* If the total number of edges in a supernode is less than
               k, then mark the extra cells with symbol END.
          */
          if M[i].SNODE ≠ M[i + 1].SNODE
            && TEMP[i] ≠ NULL
            && TEMP[i] + N[M[i].NODE].COUNT < k then
            begin
              EE[k*(M[i].SNODE − i) + TEMP[i]
                + N[M[i].NODE].COUNT + 1].NODE := END;
            end
      end
    for all j: 1 ≤ j ≤ k · m do in parallel
      begin
        TAKING(EE.NODE);  /* Cells with value 0 are viewed as blank cells. */
        TAKING(EE.SNODE);
        GROUP_PARTIALSUM(EE.NODE, EE.EDGE);
        /* Move edges from array E to array EE. */
        if EE[j].NODE = END then EE[j].EDGE := END;
        else EE[j].EDGE := E[N[EE[j].NODE].FIRSTEDGE
          + EE[j].EDGE − 1].EDGE;
      end
  end
```

After the edges are extracted, they need to be checked. If all the edges of a supernode are inside edges, they will be deleted. The total number of supernodes which are labeled OK will be stored in *oknode*. These operations are accomplished by the following routine.

```
REDUCEEDGE(k)
  begin
    for all j: 1 ≤ j ≤ k · m do in parallel
      begin
        /* Mark the edge if it is an inside edge. */
        if EE[j].NODE ≠ END then
          if EE[j].SNODE ≠ N[EE[j].EDGE].SNODE then EE[j].COUNT := 0;
          else EE[j].COUNT := 1;
        else EE[j].COUNT := 1;
      end
    /* For each supernode determine if there is an outgoing edge. */
    for all g: 1 ≤ g ≤ m do in parallel
      EE[(g − 1) · k + 1].COUNT := min{EE[(g − 1) · k + i].COUNT | i = 1, . . . , k};
    for all g, i: 1 ≤ g ≤ m, 1 ≤ i ≤ k do in parallel
      EE[(g − 1) · k + i].COUNT := EE[(g − 1) · k + 1].COUNT;
    GROUP_PARTIALSUM(EE.NODE, EE.COUNT);
    /* Delete edges. */
    for all j: 1 ≤ j ≤ k · m do in parallel
      if EE[j].NODE ≠ END && EE[j].NODE ≠ EE[j + 1].NODE then
        begin
          N[EE[j].NODE].COUNT := N[EE[j].NODE].COUNT − EE[j].COUNT;
          N[EE[j].NODE].FIRSTEDGE
            := N[EE[j].NODE].FIRSTEDGE + EE[j].COUNT;
        end
    for all g: 1 ≤ g ≤ m do in parallel
      begin
        /* Flag isolated supernodes. */
        if EE[k · g − 1].NODE = END
          && EE[k · g − 1].COUNT ≠ 0 then
          begin
            G[g].ISNODE := 1;
            EE[(g − 1) · k + 1].COUNT := 1;
          end
        else
          begin
            G[g].ISNODE := 0;
            EE[(g − 1) · k + 1].COUNT := 1 − EE[(g − 1) · k + 1].COUNT;
          end
        /* Count the number of supernodes labeled OK. */
        oknode := SUM(EE[(g − 1) · k + 1].COUNT | 1 ≤ g ≤ m);
      end
    return(oknode);
  end
```

The subroutine COMBINENODES( ) will combine supernodes to form new supernodes. Each execution of COMBINENODES( ) is guaranteed to reduce the number of supernodes by at least $\frac{1}{4}$. Nodes belonging to isolated supernodes will be stored at the end of array M. The variable n denoting the number of nodes will be updated to represent the number of nodes belonging to the nonisolated supernodes. These nodes will be stored at the beginning of array M. The isolated supernodes will be numbered to represent the connected components found.

```
COMBINENODES(k)
  begin
    for all j: 1 ≤ j ≤ k · m do in parallel
      /* Find an outgoing edge for each supernode. */
        if EE[j].NODE ≠ END
          && EE[j], SNODE ≠ N[EE[j].EDGE].SNODE then
            EE[j], COUNT := N[EE[j].EDGE].SNODE
```

```
        else
          EE[j].COUNT := 0;
    for all g: 1 ≤ g ≤ m do in parallel
      begin
        EE[(g − 1) · k + 1].COUNT := max{EE[(g − 1) · k + i | i = 1, 2, ..., k};
        /* F[g] is the outgoing edge. */
        if EE[(g − 1) · k + 1].COUNT ≠ 0 then
          F[g] := EE[(g − 1) · k + 1].COUNT;
        else F[g] := g;
        /* Find the connected components. */
        CONNECTED_TREELOOP(F);
        G[g].GROUP := F[g];
        G[g].SNODE := g;
      end
    for all i: 1 ≤ i ≤ n do in parallel
      begin
        /* Get the count for each supernode. */
        TEMP[i] := 1;
        GROUP_PARTIALSUM(M.SNODE, TEMP);
        if M[i].SNODE ≠ M[i + 1].SNODE then
          G[M[i].SNODE].COUNT := TEMP[i];
      end
    for all g: 1 ≤ g ≤ m do in parallel
      begin
        /* Form new supernodes. Sort by (G.ISNODE, G.GROUP). */
        SORT(G);
        /* Rename new supernodes. */
        if G[g].GROUP = G[g − 1].GROUP then
          G[g].GROUP := 0
        else G[g].GROUP := 1;
        GROUP_PARTIALSUM(G.ISNODE, G.GROUP);
        /* Rearrange M. */
        G[g].COUNT := G[g − 1].COUNT;   /* G[1].COUNT = 0. */
        PARTIALSUM(G.COUNT);
        PERM[G[g].SNODE] := g;
        if G[m].ISNODE = 1 then newisolated := G[m].GROUP;
        else newisolated := 0;
        m := G[m].GROUP;
        if G[g].ISNODE ≠ G[g − 1].ISNODE then
          m := G[g − 1].GROUP;
      end
    for all i: 1 ≤ i ≤ n do in parallel
      begin
        M[G[PERM[M[i].SNODE]].COUNT + TEMP[i]].NODE := M[i].NODE;
        M[G[PERM[M[i].SNODE]].COUNT + TEMP[i]].SNODE :=
        G[PERM[M[i].SNODE]].GROUP;
      end
    for all i: 1 ≤ i ≤ n do in parallel
      begin
        if G[M[i].SNODE].ISNODE = 1 then
          M[i].SNODE := newisolated − M[i].SNODE + 1 + isolated;
        isolated := isolated + newisolated;
        if G[M[i].SNODE].ISNODE ≠ G[M[i − 1].SNODE].ISNODE then n := i − 1;
      end
    return(m);
  end
```

ACKNOWLEDGMENT. We are grateful to the extremely careful reviewing undertaken by referees and their comments are helpful in improving many aspects of this paper.

REFERENCES

1. AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1973, p. 133.
2. AJTAI, M., KOMLÓS, J., AND SZEMERÉDI, E. An $O(n \log n)$ sorting network. In *Proceedings of the 15th ACM Symposium on Theory of Computing* (Boston, Mass., Apr. 25–27). ACM, New York, 1983, pp. 1–9.
3. AWERBUCH, B., AND SHILOACH, Y. New connectivity and MSF algorithms for ultracomputer and PRAM. In *Proceedings of the 1983 International Conference on Parallel Processing* (Aug.). IEEE, New York, 1983, pp. 175–179.
4. BORODIN, R. A., AND HOPCROFT, J. E. Routing, merging, and sorting on parallel models of computation. In *Proceedings of the 14th ACM Symposium on Theory of Computing* (San Francisco, Calif., May 5–7). ACM, New York, 1982, pp. 338–344.
5. CHIN, F. Y., LAM, J., AND CHEN, I.-N. Efficient parallel algorithms for some graph problems. *Commun. ACM 25*, 9 (Sept. 1982), 659–665.
6. HAN, Y. Designing fast and efficient parallel algorithms. Ph.D. dissertation, Dept. Computer Sci., Duke Univ., Durham, N.C., 1987.
7. HIRSCHBERG, D. S. Parallel algorithms for the transitive closure and the connected component problem. In *Proceedings of the 8th ACM Symposium on Theory of Computing* (Hershey, Pa., May 3–5). ACM, New York, 1976, pp. 55–57.
8. HIRSCHBERG, D. S. Parallel graph algorithms without memory conflicts. In *Proceedings of the 20th Allerton Conference*. Univ. of Illinois, Urbana-Champaign, Ill., 1982, pp. 257–263.
9. HIRSCHBERG, D. S., CHANDRA, A. K., AND SARWATE, D. V. Computing connected components on parallel computers. *Commun. ACM. 22*, 8 (Aug. 1979), pp. 461–464.
10. KRUSKAL, C. P., RUDOLPH, L., AND SNIR, M. The power of parallel prefix. *IEEE Tran. Comput. C-34*, 10 (Oct. 1985), 965–968.
11. KRUSKAL, C. P., RUDOLPH, L. AND SNIR, M. Efficient parallel algorithms for graph problems. In *Proceedings of the* 1986 *International Conference on Parallel Processing* (Aug.). IEEE, New York, 1986, pp. 278–284.
12. KUČERA, L. Parallel computation and conflicts in memory access. *Inf. Process. Lett. 14*, 2 (20 Apr. 1982), 93–96.
13. KWAN, S. C. AND RUZZO, W. L. Adaptive parallel algorithms for finding minimum spanning trees. In *Proceedings of the 1984 International Conference on Parallel Processing* (Aug.). IEEE, New York, 1984, pp. 439–443.
14. LEV, G. F., PIPPENGER, N., AND VALIANT, L. G. A fast parallel algorithm for routing in permutation networks. *IEEE Trans. Comput. C-30* (Feb. 1981), 93–100.
15. NATH, D., AND MAHESHWARI, S. N. Parallel algorithms for the connected components and minimal spanning tree problems. *Inf. Proc. Lett. 14*, 1 (27 Mar. 1982), 7–11.
16. QUINN, M., AND DEO, N. Parallel graph algorithms. *ACM Comput. Surv. 16*, 3 (Sept. 1984), 319–348.
17. REGHBATI(ARJOMANIDI), E., AND CORNEIL, D. G. Parallel computations in graph theory. *SIAM J. Comput. 2*, 2 (May 1978), 230–237.
18. REIF, J. H. Symmetric complementation. In *Proceedings of the 14th ACM Symposium on Theory of Computing* (San Francisco, Calif., May 5–7). ACM, New York, 1982, pp. 201–214.
19. SAVAGE, C., AND JA'JA, J. Fast, efficient parallel algorithms for some graph problems. *SIAM J. Comput. 10*, 4 (Nov. 1981), 682–690.
20. SCHWARTZ, J. T. Ultracomputers. *ACM Trans. Prog. Lang. Syst.* (Oct. 1980), pp. 484–521.
21. SHILOACH, Y., AND VISHKIN, U. An $O(\log n)$ parallel connectivity algorithm. *J. Algor. 3*, 1 (Mar. 1982), 57–67.
22. SNIR, M. On parallel searching. *SIAM J. Comput. 14*, 3 (Aug. 1985), 688–708.
23. WAGNER, R., AND HAN, Y. Parallel algorithms for bucket sorting and the data dependent prefix problem. In *Proceedings of the 1986 International Conference on Parallel Processing* (Aug.). IEEE, New York, 1986, pp. 924–930.

RECEIVED APRIL 1986; REVISED SEPTEMBER 1987, MAY 1988, AND SEPTEMBER 1989; ACCEPTED SEPTEMBER 1989