PROCEEDINGS
OF THE
1986 INTERNATIONAL CONFERENCE
ON
PARALLEL PROCESSING

August 19–22, 1986

Editors
Kai Hwang
Steven M. Jacobs
Earl E. Swartzlander

Co-Sponsored by

Department of Electrical Engineering
PENN STATE UNIVERSITY
University Park, Pennsylvania

and the

IEEE Computer Society
In Cooperation with the

acm

Association for Computing Machinery
EFFICIENT PARALLEL ALGORITHMS FOR GRAPH PROBLEMS

Clyde P. Kruskal
Department of Computer Science
University of Maryland
College Park, Maryland 20742

Larry Rudolph
Institute of Mathematics and Computer Science
The Hebrew University of Jerusalem
Jerusalem, Israel

Marc Snir
Institute of Mathematics and Computer Science
The Hebrew University of Jerusalem
Jerusalem, Israel

ABSTRACT New efficient techniques for manipulating linked structures in parallel are presented. These techniques work on the EREW machine, which is the weakest shared memory machine. Using these techniques we develop algorithms for connected components, spanning trees, minimum spanning trees, and other graph problems. All of these algorithms achieve linear speed-up for all but the sparsest graphs. We also present a new parallel radix sort algorithm that is optimal for keys taken from a small range.

1. INTRODUCTION

We present new parallel algorithms for several important graph problems including spanning forest, connected components, bi-connected components, and (undirected) minimum cost spanning tree (MST). All of the algorithms are improvements over the best results known in several respects: they assume the weakest model of shared memory parallel computation, i.e. the EREW (exclusive read, exclusive write) model; they are deterministic; they are fast on all graph densities. In addition, the input is not required to be in any prearranged order, such as adjacency lists; the input can consist of an unordered list of edges. When the problem size is large relative to the number of processors and the graphs are not extremely sparse (i.e. as long as $n = o(e)$), the algorithms achieve a linear speedup, which means they have optimal processor-time product. As a by product, we obtain a parallel version of radix sort that is optimal as long as the range of the elements sorted is at most polynomially larger than the number of processors.

The algorithms employ a set of useful techniques for efficient parallel manipulation of data structures to avoid conflicts between processors. The most powerful and original technique is a recursive two-step, block-based algorithm. It is used in an algorithm for solving the "parallel prefix" problem on a linked list of elements. In a quite different way, it is used to convert an edge list representation of a graph into an adjacency list representation. The former representation is how a graph is often defined; the latter representation is often required for efficient parallel access to edges of the graph. Variants of the technique are shown to be useful for performing table lookup, update without conflicts, and radix sort.

Sections 2, 3, and 4 describe basic building blocks. Section 2 reviews the model and some work on solving linked list problems in parallel. In Section 3, we show how a graph can be converted from an edge list representation to an adjacency list representation efficiently in parallel. We also present the parallel radix sort algorithm. Section 4 shows how tree problems can be solved efficiently in parallel. We then use the basic building blocks in Section 5 to obtain parallel algorithms for finding spanning trees, connected components, biconnected components, and minimum spanning trees. A chart at the end of the paper summarizes our results and shows how they compare to the current best.

2. PRELIMINARIES

2.1. The Model

We assume the PRAM computation model: a PRAM consists of $p$ autonomous processors, all having access to a common memory. At each step, each processor forms one operation from its instruction stream. Instructions accessing shared memory are also assumed to be accomplished in one cycle. Borrowing the notation of [12], we distinguish three variants of the PRAM family: In a Concurrent Read, Concurrent Write (CRCW) model, processors may simultaneously access the same memory location; there are various schemes for resolving write conflicts. In a Concurrent Read, Exclusive Write (CREW) model, several processors may simultaneously read the value of a memory location, but exclusive access is required for write. Finally, in the Exclusive Read, Exclusive Write (EREW) model, a memory location cannot be simultaneously accessed by more than one processor. Our algorithms can be implemented on the weakest of these three models: the EREW PRAM.

2.2. Linear Linked List Problems

We now review our algorithm for solving problems on linear linked lists [7]; it will be extended in the following sections to more general linked structures. Although our results for linear linked lists have since been improved by Cole and Vishkin [4], their technique does not seem to extend to more general linked structures.

The product computation problem is to compute the product $a_0 \circ a_1 \circ \cdots \circ a_{n-1}$ given $n$ elements $a_0, a_1, \ldots, a_{n-1}$ and a binary, associative operation, denoted $\circ$, e.g. ordinary, matrix, and Boolean addition and multiplication. The initial prefix problem is to compute all $n$ initial prefixes $a_0, a_0 \circ a_1, \ldots, a_0 \circ a_1 \circ \cdots \circ a_{n-1}$. The initial prefix problem when solved in parallel is known as parallel prefix. When the elements are laid out in known memory locations there are well known methods to solve both problems in time $O(n/p + \log p)$ [9].

The problem is harder to solve when the elements are stored in a linear linked list. The locations of the elements is given, but it is not known which element is which. Only the location of the first element is given along with a map Succ from the $i$th element to the $(i+1)$st element, and a map Pred from the $i$th element to the $(i-1)$st element (the second mapping can be computed from the first in $O(n/p)$ time).
It is easy to solve the product and initial prefix problems sequentially in \(O(n)\) time by starting at the first element and following the links. Wyllie [15] shows that this problem can be solved with \(p < n\) processors in \(O(n \log n / p)\) time, using a recursive doubling technique. The algorithm never achieves linear speedup.

An efficient parallel solution to this problem has the following general form.

**General Parallel Prefix Algorithm**

If the list contains more than one element then

begin
    pick a set \(S\) of nonadjacent elements in the list;
    for each \(a \in S\) do
        \begin{align*}
            &\text{begin} \{\text{replace a pair of adjacent elements by one} \} \\
            &\quad \text{val}(\text{succ}(a)) := \text{val}(a) \odot \text{val}(\text{succ}(a)); \\
            &\quad \text{succ}(\text{pred}(a)) := \text{succ}(a); \\
            &\quad \text{pred}(\text{succ}(a)) := \text{pred}(a) \\
            &\text{end;}
        \end{align*}

        apply algorithm recursively to the list;
        for each \(a \in S\) do
            \begin{align*}
                &\text{begin} \{\text{expand element back into a pair} \} \\
                &\quad \text{val}(a) := \text{val}(\text{pred}(a)) \odot \text{val}(a); \\
                &\quad \text{succ}(\text{pred}(a)) := a; \\
                &\quad \text{pred}(\text{succ}(a)) := a \\
                &\text{end}
            \end{align*}

    \end{align*}
end

The first part of the algorithm solves the product problem, successively compacting the list until only one item is left; the second part, where the recursion unfolds, expands the list back, and computes the missing partial products. Any parallel algorithm that solves the product problem by successive compaction can be used to create a parallel prefix algorithm, that expands the list back by matching step by step the compression operations done by the parallel product algorithm. The resulting algorithm will have twice the running time of the original algorithm. We shall henceforth consider only the product part.

It is not obvious how to pick at each iteration a set \(S\) of \(\Omega(p)\) nonadjacent elements, while devoting only constant time per element. Assume w.l.o.g. that both \(n\) and \(p\) are powers of 2 (this affects the result only by a constant factor), and that \(p \leq n/2\). The product algorithm consists of \(\Theta(\log n)\) iterations. At each iteration the elements are partitioned into \(n/p\) blocks, each block containing \(p\) elements. The processors visit the blocks one by one; the \(i\)th processor always visits the \(i\)th element within a block. It pairs this element with its successor, only if the element is not marked as deleted, and its successor belongs to another block. It takes \(O(n/p)\) time to process all \(n/p\) blocks.

After visiting the blocks, every element has formed the product with a neighbor, provided it and its neighbor were in different blocks; all pairings remaining to be done at this phase are internal to a block. Since there are \(n/p\) blocks and \(p\) processors, \(p^2/n\) processors can be assigned to each block.

If \(p^2/n > 1\), then we recursively apply the same routine within each block. If \(p^2/n \leq 1\), assign each processor \(n/p^2\) blocks, and pair elements sequentially with each block. Note that \(p\) and \(p^2/n\) are powers of 2.

After this, if an element has not been paired with one of its neighbors, then both of its neighbors must be paired (except for the first and last elements of the list), so at most \(2n/3 + O(1)\) elements are left.

Let \(H_p(n)\) be the number of steps required by one iteration on a list of length \(n\).

\[
H_p(n) = O\left(\frac{n}{p}\right) + H_{2p/n}(p) \quad \text{if } p > 1,
\]

\[
H_p(n) = O\left(n/p\right) \quad \text{otherwise}.
\]

It is easy to check that

\[
H_p(n) = O\left(\frac{n \log n}{p \log(n/p)}\right).
\]

After each iteration, the remaining elements are packed to the top of array, being careful to maintain the integrity of the pointers (which can be done by first determining the location of each element, then updating pointers to point to the new locations, and finally moving the elements). Once there are less than \(2p\) elements, the \(O(n \log n / p)\) parallel algorithm is used to complete the algorithm.

Let \(T_p(n)\) be the number of steps it takes to solve a problem of size \(n\), i.e. the execution time of the procedure Product. If \(n < 2p\), \(T_p(n) = O(\log n)\) using the standard parallel algorithm. Otherwise,

\[
T_p(n) = O\left(\frac{H_p(n) + n/p + \log p}{p} + T_p\left(\frac{2n}{3}\right)\right).
\]

It is not hard to show that

\[
T_p(n) = O\left(\frac{n \log n}{p \log(2n/p)}\right).
\]

We obtain

**Theorem 2.1:** The parallel prefix problem for items represented by a linked list can be solved on an EREW machine in \(O(\frac{n \log n}{p \log(n/p)})\) time for \(p < n\).

*Note:* This yields, for any constant \(0 < c < 1\), a linear speedup when \(p \leq n^{1-c}\) (or equivalently when the time \(T_p(n) \geq n^c\)).

This algorithm can be used to solve the linked list packing problem: Given a list of \(n\) items, a subset of which are active, pack the active elements to contiguous locations of memory (starting at some specified location). We can use the parallel prefix algorithm to rank the active elements, and then move each one to the location indicated by its rank plus the starting location. As a by product, the elements are packed in order so that the \(i\)th element is contiguous to the \((i-1)\)st and \((i+1)\)st elements.

A slight variation of the parallel prefix algorithm can be used to assign to each node on the list the value \(a_1 \cdot \cdots \cdot a_n\). The second (list expansion) phase is modified so as to broadcast the product computed in the first phase. We call this the product broadcast algorithm. For example, we can use this algorithm to label each node in a list with the name of the first node in the list, or with the name of the least node in the list. The product broadcast algorithm can also be run on a linked circular list, provided that \(o\) is a commutative operation.

If the parallel prefix algorithm is applied to a union of disjoint lists, then it will compute initial prefixes in parallel.
each list. Similarly, if the product broadcast algorithm is applied to a union of disjoint lists (circular lists), then it will assign to each node the product of the elements in the list it belongs to.

3. GRAPH ADJACENCY LIST CONSTRUCTION

The adjacency lists representation of a (directed) graph is particularly useful for many parallel graph algorithms. Unfortunately, this is not always the input format of a graph. The graph adjacency list construction problem has as input the list of the edges of the graph (presented as pair of nodes). The edges are stored in an arbitrary order in an array. One constructs an adjacency lists representation of the graph, that is an array \( L[1..n] \) of linked lists, where \( L[i] \) is the list of nodes adjacent to node \( i \).

Constructing the adjacency lists for a graph with \( n \) nodes and \( e \) edges requires only \( O(e + n) \) serial time and \( O(e + n) \) space. The array \( L[1..n] \) is initialized to nil in time \( O(e) \). The adjacency lists are then constructed in time \( O(e) \) by inserting edges successively in the adjacency lists.

The parallel version using \( p \) processors is not so simple. When the graph is dense, \( e = \Omega(n^2) \), a matrix can be used to avoid interference. When the graph is sparse one must be careful to insure that multiple processors do not try to add elements to the same adjacency list at the same time.

Assume w.l.o.g. that \( p \) and \( e \) are powers of 2, and that \( p \leq n/2 \). We create an adjacency list representation, where each entry in the array \( L[i] \) has a pointer to the head and a pointer to the tail of the \( i \) th list. Each entry in the edge list will point to the corresponding entry in an adjacency list. Each adjacency list is doubly linked.

If \( p^2 \leq e \) then the edges are split into \( p \) blocks each of size \( e/p \). Each processor is assigned a block and it sequentially creates adjacency lists for the edges in its block. This has to be done with some care: since \( e/p \) might be much less than \( n \), one cannot afford to pay the \( O(n) \) overhead of initializing the array entries to nil. Instead we initialize correctly only those entries that are going to be used. There are two passes over the data. In the first pass no insertions are made; for each edge, (the header of) the list the edge is to be inserted into is initialized to nil. In the second pass the insertions are actually made. We create a structure where nonempty adjacency lists have valid format, whereas empty adjacency lists may contain garbage. This phase requires \( O(e/p) \) time and \( O(pn + e) \) space.

Next all the partial adjacency lists are linked up: We start with an empty structure and serially link to it the adjacency lists of the \( p \) blocks, one at a time. All \( p \) processors always process the same block at the same time. As above this is done in two passes. In the first pass each processor picks an edge, and if this edge is the first in its list then the processor initializes the corresponding array entry in the new structure to nil. In the second pass each processor picks an edge, and if this edge is the first in its list then the processor links that list at the head of the corresponding adjacency list in the new structure. Since each block structure contains one entry per node there are no conflicts.

The time to process a block in the second phase is \( O(e/p) \). There are \( p \) blocks, so the total time for the second phase is \( O(e/p) \). The total time for the algorithm is \( O(e/p) \) and the space is \( O(pn + e) \).

If \( p^2 > e \) then the edges are split into \( e/p \) blocks of size \( p \). The algorithm is recursively applied to each block in parallel, with \( p^2/e \) processors per block. Note that \( p^2/e \) and \( p \) are powers of 2. Next the \( e/p \) partial adjacency lists are linked up, with each block being processed once at a time by the \( p \) processors. The same two-pass method is used to avoid initialization overheads. This part of the algorithm takes \( O(e/p) \) time and \( O(pn + e) \) space.

Let \( T_p(e,n) \) be the running time of this algorithm, and let \( S_p(e,n) \) be the space used by the algorithm. Then if \( p^2 \leq e \),

\[
T_p(e,n) = O(e/p).
\]

If \( p^2 > e \), we get

\[
T_p(e,n) = T_{p^2}(e/p,n) + O(e/p).
\]

This recursion solves to

\[
T_p(e,n) = O\left(\frac{e \log(e/p)}{p \log(e/p)}\right).
\]

It is easy to modify the above algorithm to set the empty list headers to nil at an additional expense of \( O((n + e)/p) \) time.

There are at most \( p \) graph adjacency list construction tasks running in parallel, with a total number of edges \( e \). Thus, the space used is \( O(pn + e) \).

Theorem 3.1: The edge list to adjacency list conversion problem can be solved on an EREW machine with \( p \leq e/2 \) processors in \( O\left(\frac{e \log e}{p \log(e/p)} + \frac{n}{p}\right) \) time and \( O(pn + e) \) space.

Note: This yields a linear speedup when \( p \leq e^{1-k} \) for any constant \( 1 > k > 0 \); the space requirement is no more than for an adjacency matrix.

Corollary 3.2: A list of \( e \) numbers in the range \( 1..n^k \) can be sorted with \( p \leq e/2 \) processors in time

\[
O\left(k \left(\frac{e \log(e/p)}{p \log(e/p)} + \frac{n}{p}\right)\right) \text{ and space } O(np + e).
\]

Proof: Assume \( k = 1 \). For each key value \( i \) in the range \( 1..n \) we create a linked list containing the items with key value \( i \). This is done in the same manner an adjacency list was created in the previous algorithm. It requires time \( O\left(e \frac{\log e}{p \log(e/p)} + \frac{n}{p}\right) \).

We then pack the list of nonempty buckets, in time \( O(n/p + \log p) \). A sorted list can now be created in time \( O(n/p) \).

The result for general \( k \) is obtained by running \( k \) phases of a radix sort algorithm, with radix \( n^k \).

Corollary 3.3: A list of \( m \) numbers in the range \( 1,...,R \) can be sorted with \( p \leq m \) processors in time

\[
O\left(\frac{m \log R}{p \log(m/p)}\right).
\]

Proof: We use the previous corollary, with \( R = n^k \). The time is minimized when \( n = \frac{m}{p \log(m/p)} \). Thereby, the sort takes time \( O\left(\frac{m \log R}{p \log(m/p)}\right) \).

The sorting time is optimal when \( p \leq R^{O(1)} \).

Corollary 3.4: Given an adjacency lists representation of a graph with \( e \) edges and \( n \) nodes, the adjacency lists representation of the reversed graph (i.e. the graph where the direction of each edge is reversed) can be computed in time...
O\left(\frac{\log e}{p} + \frac{m}{p}\right).

These results can also be used to compute the composition of mappings. Let \( f \) be a function with domain \( 1 \ldots m \) and range \( 1 \ldots n \), and let \( g \) be a function with domain \( 1 \ldots n \). Assume that \( f \) is given as a (not necessarily sorted) list of pairs \( \langle i, f(i) \rangle \), and \( g \) is represented by an array of values, with the \( i \)th entry storing \( g(i) \). We compute a representation of \( \circ g \) as a list of pairs \( \langle i, g(f(i)) \rangle \) using the following steps.

1. The pairs \( \langle i, f(i) \rangle \) are stored in bucket lists, indexed by the value of the second coordinate \( f(i) \).
2. The first element of each list is marked.
3. Each marked element \( \langle i, j \rangle (j=f(i)) \) performs a look-up for the value of \( g(j) \), and stores this value.
4. The value of \( g(j) \) is broadcast to all elements in the bucket, using the product broadcast algorithm.

Empty buckets are never accessed, so that it is not necessary to initialize them. We obtain

Theorem 3.5: The composition problem can be solved with \( p \) processors on an EREW machine in time \( O\left(\frac{m \log m}{p} + \frac{n}{p}\right) \) and space \( O(np + m) \).

If the function \( g \) is also represented by an unsorted list of pairs, the composition can be computed in time \( O\left(\frac{m+n \log (m+n)}{p} + \frac{n}{p}\right) \), using the composition algorithm given in \([11]\).

4. TREE PROBLEMS

4.1. Tree Recursions

The parallel prefix algorithm can be extended to "parallel prefix" computations on a tree. This has been independently noted for mesh connected computers [1,13].

Let \( T \) be tree represented by its adjacency list (each child has an edge directed to its parent). For each node \( u \) of \( T \) let \( \text{val}(u) \) be a value stored at that node. Let

\[ F(u) = \sum_{\text{child} \; v \; \text{of} \; u} \text{val}(v) \]

(\text{where a node is taken to be descendant of itself)}. The addition is an arbitrary associative and commutative operation. The function \( F \) can also be defined by the recursion

\[ F(u) = \sum_{\text{child} \; v \; \text{of} \; u} F(v) + \text{val}(u). \]

We wish to compute the value of \( F \) at each node of the tree. The parallel prefix problem is a particular case of this general problem, when the tree degenerates into a linked list. Other interesting cases are

1. Computing \( d(u) \), the number of descendants of each node, including itself. We take \( \text{val}(u) = 1 \), and + is usual addition.
2. Computing the path length of the tree. We take \( \text{val}(u) = d(u) \), and + is usual addition.
3. Computing the height of each node in the tree. We have \( \text{val}(u) = 1 \), and + is the minimum operation.

An efficient parallel algorithm for solving tree recursions has the following form:

**General Parallel Tree Recursion Algorithm**

If Tree contains more than one element then begin

Pick a set \( S \) of nodes such that

1. no two nodes in \( S \) are adjacent or have the same parent
2. each node of \( S \) has at most one child;
   for each node \( u \in S \) do begin
   add the value of \( u \) to the value of parent \( (u) \);
   delete \( u \);
   if \( u \) has a child then
   link the child to parent \((u)\)
   end;
   execute algorithm recursively;
   for each node \( u \in S \) do begin
   link \( u \) to parent \((u)\);
   if \( u \) has a child then
   begin
   add the value of child \((u)\) to \( u \);
   link child \((u)\) to \( u \)
   end
   end
end

It is easy to see that this algorithm solves the tree recursion problem correctly. One has to show that a set \( S \) containing a fixed fraction of the nodes can be picked at each iteration, while spending only constant time per node.

Assume that the tree \( T \) has degree bounded by two. We shall divide each iteration in the execution of the algorithm in two stages, one where left children are active, the second where right children are active. This guarantees that children of the same parent do not conflict. Conflicts between a child and its parent are handled exactly as in the parallel prefix algorithm, by dividing the edges into blocks that are handled one at a time.

At least half of the nodes of \( T \) have no more than one child. At least \( 1/3 \) of the nodes with no more than one child are paired at each iteration. We thus obtain that the recursion can be solved with \( p \) processors for a binary tree with \( n \) nodes in time \( O\left(\frac{n \log n}{p} \right) \) in the EREW model.

Let \( T \) be a general tree. The tree \( T \) can be mapped onto a binary tree \( T^b \), as described in [8, § 2.3.2]. The mapping is readily computed from the adjacency list representation of \( T \), with edges going from parent to child: \( v := \text{left}(u) \), if \( v \) is the first child in the adjacency list of \( u \); \( v := \text{right}(u) \) if \( v \) follows \( u \) in the adjacency list of their common parent.

Let \( F^b \) be the function computed on the binary tree \( T^b \) by solving the tree recursion problem on this tree. Then \( F^b(u) = \text{val}(u) \), taken over all descendants of \( u \) and descendants of elder siblings of \( u \). We have

\[ F(v) = \begin{cases} F^b(u) & \text{if } \text{right}(u) = \text{nil} \\ F^b(\text{left}(u)) + \text{val}(u) & \text{otherwise} \end{cases} \]

Thus \( F \) can be easily computed from \( F^b \). We obtain

**Theorem 4.1:** Given an adjacency list representation of a valued tree with \( n \) nodes, the tree recursion problem for the tree can be solved on an EREW machine with \( p \) processors in time \( O\left(\frac{n \log n}{p} \right) \) for \( p < n \).
4.2. Tree Traversals

Given a tree with \( n \) nodes represented by an adjacency list (with edges going from parent to child) a linear linked list of the nodes arranged in preorder can be produced by \( p \) processors in time \( O(n/p) \). A similar construction is given by Wyllie [15] for binary trees.

Let \( u_L \) and \( u_R \) be two copies of the node \( u \) in the tree. Define the mapping \( \text{next} \) as follows.

\[
\text{next}(u_L) = \begin{cases} 
  u_L & \text{if } u \text{ is first child of } u \\
  u_R & \text{if } u \text{ has no children} \\
  \text{next}(u) & \text{if } v \text{ follows } u \text{ in the adjacency list of } \text{parent}(u) 
\end{cases}
\]

\[
\text{next}(u_R) = \begin{cases} 
  \text{parent}(u)_R & \text{if } u \text{ is last child of } \text{parent}(u) 
\end{cases}
\]

This mapping corresponds to a traversal of the tree in the order Root-Children-Root \((u_L \text{ represents the first traversal of } u, \text{ and } u_R \text{ represents the second traversal})\).

Each node of the tree (with the exception of the root) occurs exactly once in an adjacency list. The linked list defined by the relation \( \text{next} \) can be created by \( p \) processors in the time it takes to traverse a linked list in parallel. Each processor creates those nodes of the list corresponding to the nodes of the tree it was assigned. The linked list traversal is required for each last child to determine its parent.

A preorder list of the nodes can be obtained in time \( O(n \log n/p) \) from this list by deleting all the “\( u_R \)” nodes, and shrinking the list accordingly. This is done by running the parallel product algorithm.

Applying parallel prefix to this list will provide the preorder number of each node with linear speedup for \( p \leq n/3 \).

If we apply this algorithm to a forest, then we shall obtain a preorder list for each tree in the forest, and compute the preorder number of each node in the tree it belongs to. Postorder on trees or forests and inorder on binary trees can be handled in a similar manner. Note too that the adjacency list of a tree can be recreated from the list defined by the \( \text{next} \) relation in time \( O(n/p) \).

4.3. Unicyclic Graphs

In this section we show how a “unicyclic” graph can be converted to a forest with the same connected components. Although algorithms on unicyclic graphs are not by themselves particularly important, this routine is useful in later sections. A unicyclic graph is a tree with one cycle; it consists of a set of \( n \) nodes, each with a parent pointer to another node in the set. Since each node in a unicyclic graph has a unique parent, one can define on such a graph the mapping \( \text{next} \) used in the last section.

Lemma 4.2: Let \( G \) be a unicyclic graph, and let \( \text{next} \) be the mapping defined above. Let \( G_{\text{next}} \) be the graph defined by this mapping: The nodes of \( G_{\text{next}} \) consist of two copies of each node of \( G \), and \( uL \) is an edge of \( G_{\text{next}} \) iff \( v = \text{next}(u) \). Then

1. The graph \( G_{\text{next}} \) consists of exactly two cycles.
2. The nodes \( u_L \) and \( u_R \) occur in distinct cycles of \( G_{\text{next}} \) iff \( u \) is on the cycle of \( G \).

Proof: Each node has exactly one successor and one predecessor in \( G_{\text{next}} \), so that this graph is a union of cycles. The node \( u \) is on the cycle of \( G \) iff a traversal starting at \( u \) reaches \( u \) again before it backtracks. Hence \( u \) is on a cycle of \( G_{\text{next}} \) starting from \( u \), before it reaches \( u_R \). The graph \( G \) can be transformed in a tree by deleting one edge from its cycle. This causes the deletion of one edge, and the change of one edge in \( G_{\text{next}} \), thereby obtaining a linear graph. It follows that \( G_{\text{next}} \) contains exactly two cycles.

Let \( G \) be a graph that is the union of disjoint unicyclical components. The graph \( G' \) is a forest decomposition of \( G \) if \( G' \) is obtained from \( G \) by deleting one edge from each cycle, thereby replacing each unicyclical component with a tree.

Theorem 4.3: Let \( G \) be a graph with \( n \) nodes that is the disjoint union of unicyclical components, represented by its adjacency list. The adjacency list of a forest decomposition of \( G \) can be computed with \( p \) processors in time \( O(n \log n / p \log (n/p)) \) in the EREW model.

Proof: The following algorithm will do.

1. Construct the linked structure defined by the \( \text{next} \) relation.
2. Assign a distinct label to each circular list (e.g. the least name, in lexicographic order, of an item in the list); label each item with the label of the list it belongs.
3. Mark those items \( u_L \) such that \( u_R \) occurs in a list distinct from the list containing \( u_L \).
4. Mark on each list the least item \( u_L \) that was marked in the previous phase.
5. Delete from the original adjacency list the edge leading to \( u \), for each item \( u_L \) that was marked at the previous stage.

Each of these phases can be implemented to run within the required time.

5. SPANNING FOREST AND CONNECTED COMPONENTS

Connected components can be derived from a spanning forest by creating a preorder list for each tree, and marking each node with the label of the first node in its list. This can be done with \( p < n/2 \) processors in the EREW model in time \( O(n \log n / p \log (n/p)) \).

Our algorithms for finding a spanning forest are in the spirit of most of the previous algorithms for this problem: (super) nodes are continually combined into super nodes. When no more combining is possible, each super node will represent a connected component, and a spanning tree will also exist for each component.

Eckstein [5] has shown that a spanning forest can be found in \( O(e/p + n) \) time using either depth-first or breadth-first search [5]. This is optimal when the number of processors is small relative to the denseness of the graph, i.e. \( p = O(e/n) \).

Let \( G \) be an (undirected) graph with \( e \) edges and \( n \) nodes. Our algorithms use two auxiliary graphs: the graph \( G_f \) is a forest that is a subgraph of \( G \); the graph \( G_s \) consists
of the supernodes of \( G \). Each connected component (tree) of \( G_f \) is represented by one supernode in \( G_s \). Two supernodes are connected if the corresponding trees are connected by an edge in \( G \). Initially, \( G_f \) contains all the nodes of \( G \) and no edges; \( G_s \) is initially identical to \( G \). At each iteration supernodes that are connected in \( G_s \) are combined into a new supernode; edges are added to \( G_f \) to merge the corresponding trees into a new tree. The algorithm terminates when \( G_s \) does not contain any more edges. At that point \( G_f \) is a spanning forest for \( G \).

The graph \( G_f \) is represented by marking in the adjacency list of \( G \) those edges that belong to \( G_f \). Separate adjacency lists represent \( G_s \). In order to achieve the desired running time, the number of accesses to each edge entry in this list must be bounded by a constant. Therefore, it is not possible to update the edge entries at each iteration. Instead, each edge in an adjacency list of a supernode of \( G_s \) is represented by an entry with the name of the incident node in \( G \). Some of the edges in this adjacency list may be self-loops, i.e. entries in the list of a supernode with the name of a node belonging to this supernode.

We keep a membership list for each supernode, i.e. a list of the nodes belonging to each supernode. Each supernode has a weight count, which is the number of nodes belonging to it. We also keep an inverse directory, i.e. an array that indicates for each node the name of the supernode it belongs to.

At the end of each iteration we pack the adjacency list of \( G_s \) so that nodes with nonempty lists of incident edges are in the first locations. As we do not want to modify the edge entries, the packing is actually done on a separate array of pointers to the supernodes.

The initial adjacency list for \( G_s \) can be created in \( O((n+e)/p) \) steps; it can be packed in time \( O(n/p + \log p) \). The membership list and inverse directory for supernodes can be created in \( O(n/p) \) time.

We terminate the “graph compaction” iterations when the number of nonisolated supernodes is \( O(\max(n/p, p^{1+\varepsilon}) \) for \( \varepsilon > 0 \). We can use Eckenstein’s algorithm to terminate in time \( O(n/p + p^{1+\varepsilon} + \varepsilon) \).

Assume that the number of supernodes with nonempty adjacency lists is at least \( p^{1+\varepsilon} \). At each iteration we perform the following steps.

(a) Pick the first edge from each nonempty adjacency list of \( G_s \). Delete these edges from the graph \( G_s \).

(b) Delete from this set of edges those edges that are self loops.

(c) Create an adjacency list for the graph consisting of the remaining edges, and their endpoints. This graph consists of a union of unicyclic components.

(d) Create a generating forest \( F \) for the graph, by deleting edges from the new adjacency list. Each tree in this forest will be replaced by one new supernode.

(e) Mark the edges of \( F \) as belonging to \( G_f \).

(f) Select in each tree of \( F \) the supernode of largest weight (this is the new supernode), and label all the supernodes in the tree with the name of the selected supernode. Mark all supernodes with the exception of the new supernodes \( F \) as inactive.

(g) Update the membership list and inverse directory for supernodes.

(h) Link the adjacency lists of supernodes marked inactive in (f) to the adjacency list of the new supernode they belong to.

(i) Pack the list of supernodes with nonempty adjacency lists.

It is easy to check that this iteration performs a valid compaction of the graph. We shall now estimate the running time of each step. Let \( a \) be the number of active supernodes (i.e. supernodes with nonempty adjacency lists) at the start of the iteration.

The edges picked in (a) are \(<\text{supernode}, \text{node}>\) pairs. We compute the \(<\text{supernode}, \text{supernode}>\) edge represented by this pair by composing the mapping represented by the pairs picked in (a) with the mapping represented by the supernode inverse directory. This is done in time \( O(\frac{a \log a}{p \log(a/p)}) \). Self loops can then be deleted in time \( O(a/p) \).

Thus, step (b) can be performed in time \( O(a/p) \). It is easy to see that each of the remaining steps, with the possible exception of step (g) can also be executed within the same time bound. As \( a \) edges are deleted from \( G_s \) at this iteration, it follows that the total time spent executing these steps is \( O(\varepsilon/p) \). We shall now estimate the time spent in updating membership lists and inverse directories for supernodes.

Let \( m \) be the number of nodes that change supernode during the iteration. The number of affected supernodes is bounded by \( a \). Using the membership list it is possible to create a list of affected nodes in time \( O(a/p) \). This can be used to update the inverse directory in time \( O(m/p) \). The membership list (and the weights associated with it) is updated by moving the member lists of inactivated supernodes in time \( O(a/p) \).

Let \( u_i \) be the number of times node \( i \) changes supernode membership. The last discussion shows that the total amount of time spent while executing step (g) is \( O(\varepsilon + \sum_{i=1}^{n} u_i) \).

Since smaller supernodes are merged into larger ones, the size of the last supernode \( i \) belongs to is at least \( 2^\varepsilon \). When the last merge is performed we have more than \( n/p \) components. This implies the inequality

\[
\sum_{i=1}^{n} \frac{1}{2^\varepsilon} > \frac{n}{p}.
\]

The maximum of \( \sum_{i=1}^{n} u_i \) under the above constraint is equal to \( n \log p \). Thus

\[
\sum_{i=1}^{n} u_i < n \log p.
\]

Hence, the total time spent performing step (g) is

\[
O(\frac{\varepsilon}{p} + \frac{n \log p}{p}).
\]

It follows that the total time taken by the algorithm is

\[
O(\frac{\varepsilon}{p} + \frac{n \log p}{p} + p^{1+\varepsilon}).
\]

Theorem 5.1: A spanning forest for a graph with \( n \) nodes and \( e \) edges can be computed by \( p < n/2 \) processors in the EREW model in time

\[
O(\frac{\varepsilon}{p} + \frac{n \log p}{p} + p^{1+\varepsilon}).
\]
and space

\[ O(pn + e). \]

**Corollary 5.2:** The connected components of a graph with \( n \) nodes and \( e \) edges can be computed with \( p < (n + e)/2 \) processors in the EREW model in time

\[ O\left( \frac{n + \log(n + e)}{p} \right) \]

and space \( O(pn + e) \).

### 4.1. Biconnected Components

Tarjan and Vishkin \[14\] developed a parallel algorithm to compute the biconnected components of a connected graph. Their algorithm, however, required that the graph be dense, the input already be in the form of an adjacency list, and that the model of parallel computation allow concurrent reads and writes. Given our algorithms to compute the spanning forest of a graph as well as the ability of construct preorder and postorder numberings and lists of a tree, we can use their method but improve the result:

**Corollary 5.3:** The biconnected components of a graph can be computed in \( O\left( \frac{n}{p} + \frac{n \log n}{p} + p \right) \) time, provided \( p \leq n \).

**Proof:** The algorithm of Tarjan and Vishkin requires the computation of a spanning tree for the graph, the computation of the preorder number of each node in this spanning tree, and the solution of several tree recurrences. We can perform each of these operations within the claimed time bound. \( \square \)

### 4.2. Minimum Spanning Tree

The spanning forest algorithm can be modified to give a minimum spanning tree algorithm. A sequential algorithm for minimum spanning tree merges supernodes iteratively. At each iteration a supernode is chosen; the least cost edge outgoing this supernode is used to merge it with another supernode. The order the supernodes are chosen is arbitrary (see \[3\]).

We can merge supernodes in parallel provided that the merging is consistent with some serial execution of the algorithm above. Let \( G_{S} \), the supernode graph, and \( G_{F} \), the spanning forest graph, be defined as in the previous algorithm: A general parallel algorithm that finds a minimum cost spanning tree for a connected graph \( G = (V, E) \) has the following form.

**General parallel algorithm for MST**

\[ G_{F} = (V, empty); G_{S} = G; \]

while \( G_{S} \) has more than one node do

begin

1. pick least cost outgoing edge from each node of \( G_{S} \);
2. let \( U \) be the subgraph of \( G \) containing these edges (\( U \) is the disjoint union of unicyclical graphs);
3. delete from each cycle of \( U \) an edge with largest cost; let \( F \) be the resulting graph (\( F \) is a forest);
4. add the edges of \( F \) to \( G_{F} \);
5. combine supernodes of \( G_{S} \) that belong to the same use of \( F \) into one supernode

end

The number of supernodes is at least halved at each iteration, so that at most \( \log n \) iterations are performed.

We implement this general algorithm using data structures similar to those used for the spanning tree algorithm: We keep an adjacency lists representation of \( G_{S} \), and represent \( G_{F} \) by marking edges in the adjacency list of \( G \). We assume that \( p < n, e \).

1. A least cost edge is picked in each adjacency list of \( G_{F} \) by running a product broadcast algorithm on the lists of edges. This can be done in time \( O\left( \frac{e \log e}{p} \right) \).
2. A trivial modification of the algorithm of Theorem 4.3 can be used to delete from each cycle of \( U \) an edge of highest cost (rather than an edge with least index, as done in the original algorithm). This is done in time

\[ O\left( \frac{n \log n}{p \log(n/p)} \right) \]

3. The new edges can be added to \( G_{F} \) in time \( O(e/p) \).
4. A supernode is selected from each tree of \( F \), and each node of the tree is marked with the label of this supernode, by running a product broadcast algorithm in time \( O\left( \frac{n \log n}{p \log(n/p)} \right) \). The edges can be now updated to point to the new supernodes in time

\[ O\left( \frac{e \log e}{p} \right) \]

This is done by computing the composition of two mappings: the mapping that maps an edge to the old incident nodes, and the mapping that maps these nodes to the new supernode they belong to. A new adjacency lists representation is built from these updated edges in time \( O\left( \frac{e \log(e/p)}{p} \right) \). While the new list is created self-loops can be deleted.

Each iteration takes time

\[ O\left( \frac{e \log e}{p} + \frac{n \log n}{p \log(n/p)} \right) \]

Since there are at most \( \log n \) iterations we obtain

**Theorem 5.4:** A minimum cost spanning tree of a connected graph can be computed with \( p \leq e, n \) processors on an EREW machine in time

\[ O\left( \frac{(e + n) \log n}{p \log((e + n)/p)} \right) \]

**Note:** This algorithm is efficient relative to Kruskal's algorithm provided that \( p = O((e + n)\log e) \).

A minimum cost spanning forest for a general graph can be computed within the same time bounds: the algorithm is modified so that supernodes with no outgoing edges are not visited. This is done by packing the list of nodes at each iteration.

### 6. Conclusion

Many simple and fast serial algorithms are often hard to parallelize. In many cases there is enough work that can be performed in parallel, but the challenge is to ensure that the processors do not conflict when performing this work, and that different processors do not replicate the same computation. The problem is harder when sparse structures are handled: If a compact data representation is used then the data layout is irregular, and it is hard to distribute work efficiently among processors. If, on the other hand, a regular data structure is used then superfluous work is performed.
We have presented techniques for working with sparse, irregular structures, and used these techniques to solve several important graph problems. We believe these techniques are generally applicable to other sparse problems. We still do not know whether large, extremely sparse graph problems can be solved efficiently, i.e. when \( p \ll n \) and \( e = \Theta(n) \).

REFERENCES


