FINDING CONNECTED COMPONENTS AND CONNECTED ONES ON A MESH-CONNECTED PARALLEL COMPUTER

DAVID NASSIMI* and SARTAJ SAHNI†

Abstract. Let \( G = (V, E) \) be an undirected graph in which no vertex has degree more than \( d \). Let \(|V| = n^d = 2^p\). In this paper we present an \( O(q^d(q+d)\log n) \) algorithm to find the connected components of \( G \) on a \( q \)-dimensional \( n \times n \times \cdots \times n \) mesh-connected parallel computer. When \( d = 2 \), the connected components can be found in \( O(q^4n) \) time. We also show that the connected ones problem can be solved in \( O(q^5n) \) time.

Key words. connected components, connected ones, mesh-connected computer, parallel algorithm, complexity

1. Introduction. A mesh-connected parallel computer (MCC) is an SIMD (Single Instruction Stream, Multiple Data Stream) computer. It consists of \( N = 2^p \) processing elements (PEs). In a \( q \)-dimensional \( n \times n \times \cdots \times n \) MCC, the PEs may be thought of as logically arranged in a \( q \)-dimensional \( n \times n \times \cdots \times n \) array. The PE at location \((i_0, \ldots, i_0)\) of the array is connected to the PEs at locations \((i_{i-1}, \ldots, i_0 \pm 1, \ldots, i_0)\) \(0 \leq j < q\), provided they exist. Two PEs are adjacent if they are connected. A PE may transmit data to an adjacent PE in a unit-route. In an MCC, each PE has some local memory. Each word and register of local memory has a storage capacity of \( \log N \) bits (all logarithms in this paper are base 2). The PEs are synchronized and operate under the control of a single instruction stream which is determined by the control unit. An enable mask may be used to select a subset of PEs that will perform the instruction to be executed at any given time. All enabled PEs perform the same instruction.

Parallel algorithms for MCCs and closely related models (such as cellular automaton and parallel processing arrays) have been studied by several researchers. Efficient sorting algorithms can be found in [11] and [15]. Algorithms for certain graph and matrix problems appear in [1] to [4], [8] to [10], and [16]. General routing problems are considered in [12], [13], and [14]. [6] and [7] cover language recognition type problems. Several other references to work on MCCs exist. Most of these can be found by tracing through the references of the papers cited above.

In this paper we shall address the following problems:

(i) Let \( G = (V, E) \) be an undirected graph with \( N = n^d = 2^p \) vertices. Let the degree of each vertex be at most \( d \). Find the connected components of \( G \). We shall call this problem: connected components for degree \( d \) graphs. The initial configuration for this problem has the adjacency list for vertex \( i \), \( A(i, 0: d-1) \), stored in \( d \) registers of PE(i).

(ii) Connected components for degree 2 graphs: This is a special case of (i) (i.e. with \( d = 2 \)). It arises in the design of a parallel algorithm to set-up the Benes permutation network (Nassimi and Sahni [14]).

(iii) Connected ones: In this problem each PE has a register called \( A \). Initially \( A(i) = 1 \) or 0, \( 0 \leq i < N \). Two ones are said to be adjacent if they are in the \( A \) registers of two adjacent PEs. The transitive closure, \( R^* \), of this adjacency relation defines the connectivity of the ones. A one in \( PE(i) \) is connected to a one in \( PE(j) \) iff \( R^*(i, j) = 1 \). The connected ones problem is to determine whether all ones are connected.

Hirschberg [5] has obtained an \( O((\log^2 N)) \) parallel algorithm to find the connected components of any \( N \) vertex undirected graph using an SIMD machine with \( N^2 \) PEs.

* Received by the editors October 5, 1979, and in final form February 20, 1980. This research was supported in part by the National Science Foundation under grant MCS 78-15455.
† Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, Illinois 60201.
‡ Department of Computer Science, University of Minnesota, Minneapolis, Minnesota 55455.
The parallel machine model used by him is different from the one we are using. In his model, all PEs share a common memory. So, data transfers between PEs may be made in \( O(1) \) time via this memory. We obtain an \( O(q^3(q + d)n \log n) \) algorithm to find the connected components of an \( n^q \)-vertex graph on an \( n^q \) PE MCC. Our algorithm can, however, be used only on graphs with vertex degree at most \( d \). The basic idea behind our algorithm is quite similar to the idea underlying Hirschberg's algorithm. The implementation of this scheme on an MCC requires the use of novel strategies. Our algorithm for problem (ii) runs in \( O(q^3 n) \) time.

The connected ones problem has been studied earlier by several researchers ([1], [8], and [9]). The problem is of importance in pattern recognition. The parallel computer model used in [1], [8], and [9] is called a parallel processing array (PPA). The essential difference between a PPA and an MCC is that each PE in a PPA is a finite state automaton capable of storing only a fixed number of bits of information. Each PE in an MCC can store \( O(\log N) \) bits of information. While other differences between the two models exist, these are not so crucial. For the PPA model, Levialdi [9] has an \( O(n) \) algorithm to solve the connected ones problem for an \( n \times n \) PPA. This algorithm can be easily run on an \( n \times n \) MCC in \( O(n) \) time. Kosaraju [8] has extended this result for the case of \( n \times n \times d \) PPA. Kosaraju's algorithm solves the connected ones problem for \( n \times n \times d \) PPA can be solved in \( O(n) \) time. It is not known if the connected ones problem for \( n \times n \times n \) PPA can be solved in \( O(n) \) time. We are able to solve the connected ones problem for \( q \)-dimensional \( n \times n \times d \) MCC in \( O(q^4 n) \) time. The underlying idea behind our algorithm is the same as that for our algorithm for problems (i) and (ii).

In §2 we discuss the solution strategy to be used in obtaining our algorithms for the three problems cited above. In §3 we introduce the notation and terminology to be used in later sections. In this section we also introduce the subalgorithms that have been developed in [13] and [15] and which will be used to arrive at the algorithms of this paper. Section 3 also contains a new MCC algorithm. This algorithm generates reduced min-trees (this term will be defined in §2). Sections 4, 5, and 6 respectively contain the algorithms for problems (i), (ii) and (iii).

2. Solution strategy. The solution strategy used to arrive at the algorithms of §§4, 5 and 6 is quite similar to that used by Hirschberg [5]. First, let us define two terms. A min-tree is a tree in which the index of each node is less than the index of each of its children. A reduced min-tree is a min-tree of height at most two. (The height of a tree is the maximum level in the tree. The root is at level 1.) The algorithms developed in this paper will partition the set of vertices in a graph (or the set of ones) into a set of reduced min-trees such that two vertices (or two ones) will be in the same reduced min-tree iff they are in the same connected component (or in the same set of connected ones).

While finding the connected components of a graph, we shall maintain several sets of vertices. These sets will represent a partition of the vertex set of the given graph, \( G \). All vertices in the same set will be in the same connected component of \( G \). Each set will be represented as a reduced min-tree with \( R(i) \) pointing to the root of the tree. The basic strategy in our connected components algorithm is to combine together sets of vertices while retaining the property that all vertices in the same set are in the same component of \( G \). This is continued until no more set combination is possible. It is a trivial matter to see that at this point, each set of vertices defines a connected component of \( G \).

We illustrate the preceding strategy by an example. Consider the 9-vertex graph of Fig. 1(a). Initially, no two vertices are known to be in the same component. So, we begin with 9 sets of vertices. These are given by the 9-tree forest of Fig. 1(b). The arrows give
the $R$ values. Observe that $R(i) = i$ for all nodes (as stated earlier, $R(i)$ gives the root of the tree). Two trees with roots $i$ and $j$, $i \neq j$, are adjacent iff tree $i$ contains a node $i'$ and tree $j$ contains a node $j'$ such that $i'$ and $j'$ are adjacent in $G$. The tree with root $j$ is the min-adjacent tree to the tree with root $i$ iff the tree with root $i$ has no adjacent tree with root less than $j$. To determine whether two sets should be combined, we first determine the min-adjacent tree $A(i)$ for each root $i$. The $A$ values are given below each root of Fig. 1(b). As can be seen, for some $i$, $A(i) > i$. This is undesirable, since changing $R(i)$ to $A(i)$ to effect a set combination would result in nontrees (i.e. graphs with cycles). This is remedied by changing all $R(i)$ with $A(i) > i$ to $A(A(i))$.

**Lemma 1.** $A(A(i)) \leq i$.

**Proof.** Let $A(i) = j$. Since trees $i$ and $j$ are adjacent, $A(j) \leq i$. □

From Lemma 1 it follows that if we replace $R(i)$ by $A(i)$ if $A(i) \leq i$, and by $A(A(i))$ otherwise, then we will be combining together min-adjacent trees and will be left behind with min-trees. Fig. 1(c) gives the updated $A$ values and Figure 1(d) shows the min-trees resulting from the combination just described. All min-trees of Fig. 1(d) are also reduced min-trees. Repeating the above combination process, we first find that
A(1) = 2, A(2) = 1, A(3) = 2 and A(4) = 3. Updating A(i) for A(i) > i, we get A(1) = 1. The resulting min-tree is given in Fig. 1(e). The corresponding reduced min-tree is given in Fig. 1(f). The next lemma shows that the combination process described above need be repeated at most log \( N \) times for an \( N \)-vertex graph.

**Lemma 2.** If the vertices of a connected component are in \( r \) distinct trees, \( r > 1 \), then following a tree combination as described above, they are in at most \( \lceil r/2 \rceil \) trees.

**Proof.** Let the roots before the combination be \( I = \{i_1, i_2, \ldots, i_r\} \). Let \( A(i), i \in I \), be the root of the min-adjacent tree for root \( i \). And, let \( I = I^C \cup I^D \) where \( I^D = \{i | i \in I \text{ and } A(i) > i\} \), and \( I^C = \{i | i \in I \text{ and } A(i) < i\} \). The only candidates for root nodes following the combination are nodes \( I^D \) (since for any node \( i \in I^C \), \( R(i) \) is changed to \( A(i) < i \)). For a node \( i \in I^C \), \( R(i) \) becomes \( A(A(i)) = A(j) \), where \( j = A(i) \) and \( A(j) \leq i < j \). Thus, the number of new roots, \( r' \), is also no more than the number of nodes of \( I \) with \( A(j) < j \). That is, \( r' \leq |I^D| \). And since \( r' \leq |I^D| \), we conclude \( r' \leq \lceil r/2 \rceil \). \( \Box \)

3. **Terminology, notation and subalgorithms.** Throughout, we shall assume that we are dealing with an \( n \times n \times n \times \cdots \times n \) \( q \)-dimensional MCC with \( N = n^q = 2^n \) PEs. For any integer \( i \), \( (i)_s \), \( s \leq i \), will denote the number with binary representation \((i)_s \). Bit 0 is the least significant bit. By a \( 2^k \)-block of PEs we shall mean a block of \( 2^k \) consecutively indexed PEs whose indices differ only in the least significant \( k \) bits. We shall index the PEs in a \( q \)-dimensional MCC in shuffled row-major order (see [13] or [15]). When this indexing scheme is used, the index of the PE in position \((i_q-1, \ldots, i_0)\) of the \( q \)-dimensional PE array is obtained by merging together the binary representations of \((i_q-1, \ldots, i_0)\). Let \( t = \log n \). When shuffled row-major indexing is used, the binary representation of the index of the PE in position \((i_q-1, i_{q-2}, \ldots, i_0)\) is \((i_q-1)_t \cdots (i_{q-1})_{t-1} (i_0)_{t-1} (i_q-1)_{t-2} (i_{q-2})_{t-2} \cdots (i_0)_{t-2} \cdots (i_q-1)_{0} (i_{q-2})_0 \cdots (i_0)_0\). (Recall that \((i_q-1)_{t-1}\) denotes bit \( t - 1 \) of \((i_q-1)\)) Fig. 2 gives the PE indices resulting when the shuffled row-major indexing scheme is used on a \( 4 \times 4 \) MCC.

![Fig. 2. Shuffled row-major indexing.](image)

If we have a \( q \)-dimensional MCC with \( N = n^q = 2^n \) PEs, then each \( 2^n \)-block of PEs will form an \( n/2 \times n/2 \times n \times \cdots \times n \) array, each \( 2^{n-2} \)-block will form an \( n/2 \times n/2 \times n \times \cdots \times n \) array, and each \( 2^{n-q} \)-block will form an \( n/2 \times n/2 \times \cdots \times n/2 \) array when shuffled row-major indexing is used. In general, a \( 2^k \)-block forms an \( m_q-1 \times m_{q-2} \times \cdots \times m_0 \) array, where

\[
\begin{align*}
m_i &= \begin{cases} 
2^{[k/q]} & 0 \leq i < d, \ d = k \mod q, \\
2^{[k/q]} - 1 & d \leq i < q,
\end{cases}
\end{align*}
\]

To see this, note that the most significant bit of a PE index comes from dimension \( q - 1 \), the next from dimension \( q - 2 \), and so on. In general, bit \( i \) of a PE index is the \([i/q]\)th bit of dimension \( i \mod q \).

In words, \((1)\) states that when shuffled row-major indexing is used, then each \( 2^k \)-block forms an \( m_{q-1} \times m_{q-2} \times \cdots \times m_0 \) array such that \( \sum_{i=0}^{q-1} m_i \) is minimized. As an
example, consider \(2^2\)-blocks for a \(4 \times 4\) MCC using row-major and shuffled row-major indexing (Fig. 3). The quantity \(m_0 + m_1\) is 4 for the shuffled row-major indexing but 5 for the row-major indexing.

\[
\begin{array}{cccc}
0 & 1 & 4 & 5 \\
2 & 3 & 6 & 7 \\
8 & 9 & 12 & 13 \\
10 & 11 & 14 & 15 \\
\end{array}
\]  
\(\text{Shuffled row-major}\)

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
4 & 5 & 6 & 7 \\
8 & 9 & 10 & 11 \\
12 & 13 & 14 & 15 \\
\end{array}
\]  
\(\text{Row-major}\)

**Fig. 3.** \(2^2\)-blocks.

Our algorithms will make use of two kinds of assignment statements. The first kind is an assignment requiring no data routing. This will be denoted by the use of the symbol `:=`. The second kind will require data routing among the PEs and will be denoted by `->`. PE selectivity will be done by providing a masking function. For example, the statement

\[
X(i) := Y(i), ((i)_b = 0)
\]

has selectivity function \((i)_b = 0\), and the assignment \(X := Y\) is to be carried out only on PEs with bit \(b = 0\). When a selectivity function is provided for a \texttt{for} loop, the instructions in the \texttt{for} loop body are to be performed only on PEs satisfying the selectivity function.

The discussion of the next three sections will make use of the following algorithms.

(i) SORT.

This algorithm sorts data items \(O(i), 0 \leq i < N = n^q\), into nondecreasing order assuming a shuffled row-major indexing scheme. Its complexity is \(O(q^2n)\) (see [15]).

(ii) RAR (Random Access Read).

In an RAR, each PE specifies a PE index from which it wishes to receive data. The RAR algorithm described in [13] routes data from the source PEs to the PEs desiring to receive the data. Each PE is allowed to request data from at most one PE. Several PEs can request data from the same PE. The algorithm of [13] has complexity \(O(q^2n)\). An RAR in a \(2^q\)-block takes \(O(q^2\frac{q^2}{q})\) time.

(iii) RAW (Random Access Write).

In an RAW, each PE specifies the PE to which its data is to be sent. If two or more PEs specify the same destination PE, then the RAW algorithm of [13] can be set either to send all pieces of data to the destination or to send a selected one (say, the one with minimum key). In the former case, if data from at most \(d\) PEs is to go to one PE, then the time needed is \(O(qn(q + d))\). In the latter case, the time needed is \(O(q^2n)\). An RAW in a \(2^q\)-block takes \(O(q^2\frac{q^2}{q} q + d)\) or \(O(q^2\frac{q^2}{q})\) time, respectively.

(iv) RANK.

The rank of a selected record in a PE is the number of PEs of smaller index which contain a selected record. For example, assume we have 8 PEs each containing one record. Let the key values of these 8 records be \((6, 4, 2, 2^*, 6, 6^*, 3^*, 4^*)\) where an asterisk over a key value denotes a flagged or selected record. The ranks of the flagged records are \((- , - , - , 0, - , 1, 2, 3)\). Nassimi and Sahni [13] present an algorithm to rank records in each \(2^q\)-block of PEs. Let \(s = \lfloor k/q \rfloor\), \(d = k \mod q\), and let \(m_{s}\) be as given in (1). The number of unit routes needed by the ranking algorithm of [13] is
$2 \sum_{d=0}^{k-2} (m_i - 1) = (q + d)2^{r-1} - 2q = O(q2^{[k/2]})$. When $k = \log N = q \log n$, the number of unit-routes becomes $2q(n - 1)$.

(v) CONCENTRATE.

Let $G(i, r, 0 \leq r \leq j)$ be a set of records with $G(i)$ initially in PE($i$). Assume that the records have been ranked so that the rank $H(i)$ of record $G(i)$ is $r$. A concentrate results in record $G(i)$ being moved to PE($r$), $0 \leq r \leq j$. The algorithm given in [13] carries out this function using the same number of unit-routes as used by the ranking algorithm. This algorithm permutes records so that no record is destroyed.

(vi) REDUCE.

Let $R(0: N-1)$ define a set of min-trees (see § 2). $R(i) = i$ if $i$ is a root of a min-tree and $R(i) < i$ otherwise. Procedure REDUCE($k$) generates the set of reduced min-trees when each of the original min-trees is confined to a $2^k$-block of PEs. A $2^k$-block may contain more than one min-tree. Note that $R(i)$ denotes a register in PE($i$). In line 3, we assume that the condition $(R(i)_{k-1:b} = (i)_{k-1:b})$ will be true for all $i$ when $b > k - 1$.

\begin{verbatim}
line  procedure REDUCE(k)
1   global R
2   for b := 1 to k do
3     R(i) ← R(R(i)), (R(i)_{k-1:b} = (i)_{k-1:b})
4   end
5 end REDUCE
\end{verbatim}

Algorithm 1

The correctness of REDUCE may be established by induction on $b$. We shall show that following iteration $b = r$ of the for loop, the following condition holds:

$C^r$: For any $i$, $R(i) = j \leq i$. Furthermore, either $j$ is a root, or $i$ and $j$ are in different $2^r$-blocks.

Observe that initially $C^0$ holds. For the induction step, assume that $C^{r-1}$ holds following iteration $b = r - 1$. During iteration $b = r$, PE($i$) is enabled to update its $R(i)$ if $i$ and $R(i) = j$ are in the same $2^r$-block. If $j$ is a root, then the update does not alter $R(i)$. If $j$ is not a root, then from $C^{r-1}$, it follows that $i$ and $j$ are in different $2^{r-1}$-blocks such that $(i)_{r-1} = 1$ and $(j)_{r-1} = 0$. Let $R(j) = l$. Following the update, we get $R(i) = l < i$. If $l$ is not a root, then it must be in a lower $2^{r-1}$-block with respect to $j$; this means that $i$ and $l$ are in different $2^r$-blocks. Hence, $C^r$ will hold after iteration $b = r$. The correctness of REDUCE($k$) will follow from $C^k$ since each of the original min-trees was restricted to a $2^k$-block.

Line 3 of REDUCE is an RAR in a $2^k$-block, and requires $O(q2^{[k/2]})$ time. The complexity of REDUCE($k$) is therefore $O(q3^k/q)$. When $k = p = \log N$, the complexity becomes $O(q^3 n)$.

4. Connected components for degree $d$ graphs. Let $G$ be a graph with $2^k$ vertices. No vertex has degree more than $d$. Let $\text{ADJ}(i, j)$, $0 \leq j < d$, be the adjacency list for vertex $i$, $0 \leq i < 2^k$. If vertex $i$ has degree $d_i < d$ then we assume that $\text{ADJ}(i, j) = \infty$, $d_i \leq j < d$. We also assume that $\text{ADJ}(i, j)$, $0 \leq j < d$, denotes memory cells/registers associated with PE($i$), $0 \leq i < 2^k$. A shuffled row-major indexing of PEs is assumed. The graph resides in a $2^k$-block of PEs.

Procedure CONNECT($k$, $d$) is a direct implementation of the strategy outlined in § 2. Line 1 initializes the vertex sets to contain one vertex each. Each set is represented as a min-tree. The for loop of lines 2–14 iterates the set combination process $k = \log 2^k$
times. Lines 3–13 implement the set combination process. In lines 3–9 we determine for each vertex \( j \), its min-adjacent tree. This is defined as:

\[
\text{CANDID}(j) = \min \{ R(t) | t = \text{ADJ}(j, l) \text{ for some } l \text{ and } R(t) \neq R(j) \}.
\]

Line 10 then finds the min-adjacent tree for each root \( i \). Line 11 takes care of trees that have no adjacent trees, and line 12 updates \( R \) according to Lemma 1. From the discussion of § 2, it follows that after line 12, \( R \) defines a set of min-trees. Line 13 produces a reduced min-tree from each tree.

As far as the complexity of CONNECT is concerned, we observe that lines 6 and 12 require RARs while line 10 is an RAW. Hence, each of these three lines requires \( O(q^22^{k/q}) \) time. Line 13 requires \( O(q^22^{k/q}) \) time. The overall complexity of CONNECT is therefore \( O(k(q^22^{k/q} + dq^22^{k/q})) = O(kq^2(q + d)2^{k/q}) \). Note that in this much time we can find the connected components of several \( 2^k \)-vertex graphs by using each \( 2^k \)-block of PEs for a different graph. Also note that when \( k = \log N \), the complexity of CONNECT becomes \( O(q^2(q + d)N^{1/q} \log N) = O(q^2(q + d)n \log n) \).

```
line procedure CONNECT(k, d)
    //ADJ(i, 0: d-1) gives the adjacency list for PE(i). //
    //d is the degree of the graph. \( 2^k \) is the number of PEs//
1  R(i) := i; //start with single-node trees//
2  for b := 0 to k-1 do //merge trees//
3      CANDID(j) := \infty
4      for e := 0 to d-1 do //get smallest neighboring root//
5          TEMP(j) := \infty
6          TEMP(j) := R(ADJ(i, e)); //get root from neighbor//
7          TEMP(j) := \infty, (TEMP(j) := R(j)) //discard if your own root//
8          CANDID(j) := \min(CANDID(j), TEMP(j))
9  end //find min-adjacent tree//
10  R(i) := \min\{CANDID(j) | R(j) = i\}
11  R(i) := i, (R(i) = \infty) //this tree cannot grow//
12  R(i) := R(R(i)), (R(i) > i) //convert to min-tree//
13  call REDUCE(k) //reduce min-tree//
14  end
15 end CONNECT
```

Algorithm 2.

5. Connected components for degree 2 graphs. As stated earlier, this special case arises in setting up the Benes permutation network [14]. We shall show that this special case can be solved in \( O(q^4n) \) time on a \( q \)-dimensional MCC with \( N = n^q \) PEs. As in the previous section we begin with the adjacency list of vertex \( i \) in \( \text{PE}(i) \). The indices of the (at most) two vertices adjacent to vertex \( i \) are stored in \( \text{ADJ}(i, 0) \) and \( \text{ADJ}(i, 1) \). If vertex \( i \) is of degree 1 then \( \text{ADJ}(i, 1) = \infty \); if the vertex has degree 0 then \( \text{ADJ}(i, 0) = \text{ADJ}(i, 1) = \infty \). Our connected components algorithm will produce reduced min-trees with the property that \( R(i) = R(j) \) iff \( i \) and \( j \) are in the same connected component.

Our algorithm for degree 2 graphs uses a strategy slightly different from that described in § 2. Consider the example graph of Fig. 1(a). As before, we begin with each vertex in a set of its own. Vertices in a set are represented by a min-tree. Fig. 1(b) gives the initial forest denoting the sets. Each tree determines its min-adjacent tree. These
are given outside the nodes in Fig. 1(b). Fig. 1(d) depicts the min-trees following a tree combination step making use of Lemma 1. At this point, rather than repeat the tree-combination step and get the min-tree of Fig. 1(e), we form a new graph which contains only the root nodes of Fig. 1(d), i.e., nodes 1, 2, 3, and 4. This graph is called the root graph. In the new root graph, vertices $i$ and $j$ are adjacent iff the trees with roots $i$ and $j$ are adjacent. The resulting graph is given in Figure 4(a). Note that, in general, if vertices $i_1, i_2, \ldots, i_r$ are in the same min-tree following the tree-combination step, then at most two of the vertices $i_1, i_2, \ldots, i_r$ can be adjacent to vertices not in the tree. To see this, observe that the graph we began with was of degree 2. Each vertex in a min-tree is adjacent to at least one other vertex in the tree (provided the tree has more than one vertex). If every vertex of $i_1, i_2, \ldots, i_r$ is adjacent to two other vertices in the same min-tree, then none of $i_1, i_2, \ldots, i_r$ can be adjacent to a vertex not in the min-tree. So, assume there is a vertex (say $i_1$) which is adjacent to exactly one of $i_2, \ldots, i_r$. Without
loss of generality, we may assume this vertex to be $i_2$. $i_2$ must be adjacent to one of $i_3, \ldots, i_t$, otherwise $i_3, \ldots, i_t$ cannot be in the same min-tree as $i_1$ and $i_2$. Repeating this argument, we see that the vertices in a tree can be relabeled so that $i_j$ is adjacent to $i_{j-1}$, $1 \leq j < t$. So vertices $i_t$ and $i_j$ are the only vertices that can be adjacent to a vertex not in the tree. Let the root of this tree be $i_t$ and let $u$ and $v$ be the two possible vertices adjacent to $i_t$ and $i_j$ and not in this tree. Let $R(u)$ and $R(v)$ be the roots of the trees containing $u$ and $v$ respectively. In the new graph of root nodes, only $R(u)$ and $R(v)$ can be adjacent to vertex $i_t$. So, the new root graph is also of degree 2.

The tree combination step of § 2 is repeated on the root graph of Fig. 4(a). This yields Fig. 4(b), 4(c) and 4(d). The min-tree of Fig. 4(d) is reduced to get Fig. 4(e). At this point, if more than one reduced min-tree existed, we would form a new root graph and re-apply the tree combination step. The tree combination-root graph formation process has to be repeated at most $\log N$ times if we start with an $N$ node graph (see Lemma 2). Our algorithm will repeat this basic step exactly $\log N$ times. Once we are finished with this step, we will be left with one min-tree for each component (as in Fig. 4(f)). The min-tree for each component is then reduced as in Fig. 4(g).

```
line  procedure CONNECT2(p)
//ADJ(i, 0; 1) gives the adjacency list for PE(i)//
//2^p = N is the number of PEs//
1 ORG(i) := R(i) := i //original address of a node//
2 LIVE(i) := 1
3 LIVE(i) := 0, (ADJ(i, 0) = ADJ(i, 1) = \infty)
4 b := p //initial graph is in a 2^p-block//
5 loop. (LIVE(i) = 1 and i < 2^p) //active set of PEs//
6 R(i) := \min(ADJ(i, 0), ADJ(i, 1))
7 R(i) := R(R(i)), (R(i) > i) //convert to min-trees//
8 call REDUCE(b)
9 if b = 1 then[R(i) := ORG(R(i))]
10 exit from loop] // go to line 25//
11 for e := 0, 1 do
12 ADJ(i, e) := R(ADJ(i, e))
13 ADJ(i, e) := \infty, (ADJ(i, e) = R(i))
14 end
15 CANDID(i) := \min(ADJ(i, 0), ADJ(i, 1))
16 {ADJ(i, 0), ADJ(i, 1)} := {CANDID(j) | R(j) = i}
17 R(i) := ORG(R(i)) //cut off all nodes from roots//
18 LIVE(i) := 0, (R(i) \neq ORG(i) or ADJ(i, 0) = ADJ(i, 1) = \infty)
//update adjacency lists of live nodes before moving://
19 call RANK(b) //rank live nodes. Let H be the rank.//
20 ADJ(i, 0) := H(ADJ(i, 0))
21 ADJ(i, 1) := H(ADJ(i, 1))
22 call CONCENTRATE(b) //concentrate live nodes. The records//
// are G = (ORG, LIVE, ADJ, R)//
23 b := b - 1
24 repeat //go to line 5//
25 R(ORG(i)) := R(i) //send all nodes back to origin; use SORT.//
26 call REDUCE(p)
27 end CONNECT2
```

Algorithm 3
The reason the addition of the root graph formation step leads to an asymptotically faster algorithm than procedure CONNECT (Algorithm 2) is that a root graph cannot contain more than \( \frac{N}{2} \) nodes of degree one or two if the original graph contained \( N \) nodes. This follows from Lemma 2. As a result, the root graph can be concentrated into a \( 2^{r-1} \)-block of PEs before tree combination. The next root graph formed will have at most \( \frac{N}{4} \) nodes and so can be concentrated into a \( 2^{r-2} \)-block of PEs. Thus, following each tree combination step, we can localize the next root graph to a smaller block of PEs. Tree combination in a smaller block of PEs takes less time than in a bigger block and so the resulting algorithm is faster.

Now, let us look at the details of the algorithm. Since our algorithm will be concentrating root graphs into smaller blocks of PEs, we will have a need to know the originating PE for each vertex in a root graph. ORG(i) will be used to denote the originating PE for the vertex currently in PE(i). The variable LIVE will be used to distinguish between nodes in the current root graph and other nodes: LIVE(i) = 1 if the vertex currently in PE(i) is in the root graph and has degree more than zero; LIVE(i) = 0 for all other PEs. We may regard the initial graph as a root graph. Procedure CONNECT2 (Algorithm 3) is a formal specification of our algorithm. Lines 1 to 3 initialize ORG, R, and LIVE. The variable \( b \) is used to denote the current block size. It is initialized to \( p \) (i.e. the block size is \( 2^p = N \)) in line 4. Lines 5 to 24 define the basic tree combination-root graph formation step. The PE selectivity function specified in line 5 requires that the statements within the loop body be executed only on "live" PEs that are in the "first" \( 2^{k-1} \)-block (the "first" \( 2^k \)-block contains PEs 0, 1, \( \ldots \), \( 2^k - 1 \)). Additional selectivity functions provided within the loop further restrict the PEs on which certain statements are to be executed.

At the start of each iteration of the loop of lines 5–24, we have a new root graph. Each vertex in this graph is in a different set. So, to find the min-adjacent tree for any single-node tree \( T \), we need only find the least indexed vertex adjacent to the sole vertex in \( T \). This is done in line 6. Line 7 updates \( R \) according to Lemma 2. The min-trees created in lines 6 and 7 are reduced in line 8. If \( b = 1 \) then only two vertices could be present in the root graph. So, only one min-tree can result following lines 6 and 7. Hence, no further iterations are needed, and the loop is exited from line 10. If \( b \neq 1 \), then further iterations of the tree combination step may be needed. So, we proceed to set up the new root graph. In lines 11 to 15 each live vertex determines whether it is adjacent to another live vertex in a different min-tree. Following line 15, CANDID(i) ≠ \( \infty \) if the vertex in PE(i) is adjacent to a live vertex in a different tree. If CANDID(i) ≠ \( \infty \), then CANDID(i) equals the index of the PE containing the adjacent live vertex's root. In line 16 the (at most) two vertices that will be adjacent to root \( i \) in the new root graph are recorded in \( \text{ADJ}(i, 0) \) and \( \text{ADJ}(i, 1) \). Note that one or both of these values may be \( \infty \). At this point the \( R \) value of each live PE(\( i \)) is updated to be the originating PE of the vertex in PE(\( R(i) \)) (line 17). Following this, \( R(i) = \text{ORG}(i) \) only for root nodes. Line 18 "kills" nodes that are not to be in the new root graph as well as nodes that would have a degree of zero in the new root graph.

Following the creation of the new root graph, the new root graph is to be concentrated into a smaller block of PEs. Procedure RANK ranks all the nodes in the new root graph. The rank, \( H(d) \), of a node gives the PE to which it is to be routed during the concentration. \( \text{ADJ}(i, 0 : 1) \) is updated in lines 20 and 21 to reflect the PE indices of the adjacent nodes following the concentration. Line 22 actually concentrates the nodes in the new root graph. The records being concentrated are \( \text{ORG}(i), \text{LIVE}(i), \text{ADJ}(i, 0 : 1), R(i) \) for \( \text{LIVE}(i) = 1 \) and \( i < 2^k \). Note that if there are \( j \) live PEs, \( j \leq 2^k - 1 \), then following a concentration the live records occupy PEs 0, 1, \( \ldots \), \( j - 1 \) and
the remaining records are in PEs $j, \ldots, N - 1$. No record is destroyed during concentration. Also, CONCENTRATE does not change the relative order of live records since for two live records $i$ and $j$, $H(i) > H(j)$ if $i > j$. As a result of this, we can use $\text{ADJ}(i, 0: 1)$ rather than $\text{ORG(ADJ}(i, 0))$ and $\text{ORG(ADJ}(i, 1))$ when finding min-adjacent trees (line 6). Note that for $R(i)$ as given in line 6, we have $\text{ORG}(R(i)) = \min\{\text{ORG}(\text{ADJ}(i, 0)), \text{ORG}(\text{ADJ}(i, 1))\}$; assume $\text{ORG}(\infty) = \infty$. Thus, in lines 6, 7, 12, 13, 15, 16, 20, 21 $R$ and ADJ are really PE indexes and not vertex indexes. In lines 9 and 17 $R$ is reset to be a vertex index. So, on exit from the loop, all $R$ values are vertex indexes. Line 25 sends every vertex back to its originating PE. It is easy to see that following this, we shall have $R(i) \leq i$ for every $i$, $0 \leq i < N$. The min-trees are finally reduced in line 26.

To obtain the complexity of procedure CONNECT2, we need be concerned only with lines 7, 8, 9, 12, 16, 17, 19, 20, 21, 22, 25 and 26. The remaining lines contribute a total of $O(p)$ time. Lines 7, 9, 12, 17, 20 and 21 are RARs in a $2^b$-block. Line 16 is an RAW in a $2^b$-block. During this RAW, only the smallest two values destined for a given PE are to reach the destination PE. Using the complexity figures given in § 3 for RARs, RAWs, REDUCE, CONCENTRATE and RANK in a $2^b$-block, we see that each iteration of the loop of lines 5 to 24 takes $O(q^{2(\log q')})$ time. So, the overall time spent in this loop is $O(q^n)$ where $N = n^2 = 2^b$. Line 25 requires a sort on the field ORG. This takes only $O(q^n)$ time. Line 26 takes $O(q^n)$ time. So, CONNECT2 has time-complexity $O(q^n)$.

6. Connected ones. Before describing our algorithm to solve the connected ones problem, we introduce some terminology. Two $2^b$-blocks are siblings if they together form a $2^{b+1}$-block. A $2^b$-block is a left $2^b$-block if it contains only PEs with bit $b$ equal to zero. It is a right $2^b$-block if all PEs have bit $b$ equal to one. A PE in a $2^b$-block is a boundary PE if it is adjacent to a PE in its sibling $2^b$-block. Let $d = b \mod q$. From the discussion in § 3, we know that bit $b$ of a PE index is bit $[b/q]$ of dimension $d$ when shuffled row-major indexing is used. So, a $2^{b+1}$-block results from combining two sibling $2^b$-blocks along dimension $d$. Also, a $2^b$-block defines a PE array of size $m_{q-1} \times m_{q-2} \times \cdots \times m_0$, where the $m_d$ are as given by (1). In particular, $m_d = 2^{(b/d)}$. The number of boundary PEs, $t$, in each $2^b$-block is therefore $t = 2^{b - (b/d)}$.

Our algorithm for the connected ones problem actually partitions all the PEs into sets such that PE($i$) and PE($j$) are in the same set iff $A(i) = A(j) = 1$ and these two ones are connected. On termination of the algorithm, each partition is represented by a reduced min-tree. We shall have $R(i) = R(j) \neq \infty$ iff $A(i) = A(j) = 1$ and these two ones are connected. $R(i) = \infty$ iff $A(i) = 0$. To determine if all the ones are connected we need only check if the number of distinct $R$ values (not counting $\infty$) is more than one. This is easy to do.

Our algorithm begins by considering each $2^0$-block. For the lone PE in a $2^0$-block, $R(i) = i$ if $A(i) = 1$, and $R(i) = \infty$ if $A(i) = 0$. From the sets of connected ones in each $2^b$-block, we construct the sets of connected ones in each $2^{b+1}$-block, $0 \leq b < p$. (Recall that the MCC has $N = 2^b$ PEs.) The sets of connected ones in a $2^{b+1}$-block are obtained by combining together the sets for the two $2^b$-blocks contained in the $2^{b+1}$-block. Sets are combined in the same manner as before, i.e., min-adjacent trees are combined together. In defining the set adjacencies for purposes of this combination, it is sufficient to consider only boundary node adjacency. Let $i$ be a boundary PE in a left $2^b$-block and let $j$ be a boundary PE in the corresponding right $2^b$-block. PE($i$) is a live boundary PE iff it is adjacent to a PE($j$) in its sibling $2^b$-block and $A(i) = A(j) = 1$. Note that each live boundary node is adjacent to exactly one other live boundary node in its $2^{b+1}$-block. A
PE is a live root PE if it is the root of a min-tree containing a live boundary PE (a live root PE can also be a live boundary PE). Thus, to obtain the sets of connected ones in a $2^{b+1}$-block, we need only attempt to combine those $2^b$-block sets with a live root. This combination can be carried out by considering only the adjacencies of the live boundary nodes.

With this introduction, we are ready to look at the details of procedure CONNECT ONES (Algorithm 4). This procedure uses a sub-procedure ROUTE($E$, $d$, $i$) which transmits the data in the $E$ register of each PE to the $E$ register of a PE that is $i$ units away along dimension $d$. $i$ maybe positive or negative depending on the direction along dimension $d$ that the route is to be performed. Lines 1 and 2 set-up the reduced min-trees corresponding to $2^b$-blocks. Lines 3–30 build the reduced min-trees for each $2^{b+1}$-block, $0 \leq b < p$. Lines 4–11 determine the live boundary PEs. This is done by first determining the dimension, $d$, along which the member $2^b$-blocks are combined (line 4). Following line 7, $E(i) = 1$ iff PE($i$) is a live boundary PE in a left $2^b$-block and following line 10, LIVE($i$) = 1 iff PE($i$) is a live boundary PE. Line 12 identifies the live root-PEs. Since the number, $t$, of boundary PEs in a $2^k$-block is $2^k - 1$, the number of live PEs (including live root PEs) in a $2^{b+1}$ block is no more than $\min(2^{b+1}, 4t)$. Live min-trees are combined by first concentrating the live nodes in each $2^{b+1}$-block into a “corner” of that block. This requires us first to rank the live nodes (line 13) and then to set-up the adjacency list for each live boundary node. As remarked earlier, each live boundary node is adjacent to exactly one live boundary node. Lines 16–22 set-up Adj($i$) = $j$ for each live boundary node $i$. $j$ is the PE index to which $i$’s adjacent live boundary node will be moved. The concentration of live nodes is performed in line 24. The records being concentrated are $G(i)$ = (R($i$), Adj($i$), LIVE($i$), ORG($i$)). (As before, CONCENTRATE permutes the records in each $2^{b+1}$-block so no record is destroyed.) CONNECT is the same as procedure CONNECT of § 4 except that line 1 is omitted and only PEs with LIVE = 1 are involved in any computation. Following line 1, live nodes $i$ and $j$ have the property that $R(i) = R(j)$ iff ORG($i$) and ORG($j$) are in the same set of connected ones for the $2^{b+1}$-block containing PEs $i$ and $j$. Lines 27 and 28 move the reduced min-trees created in line 26 back to the originating PEs. The reduced min-trees together with the PEs that were not live before the concentrate (line 24) form min-trees of height at most 3. Line 29 reduces these min-trees (note that $R(i) = i$ for a root). Line 29 may be restricted to PEs which had LIVE($i$) = 0 before line 24 was executed.

Let $m = 2^{|b/a|}$ and $m' = 2^{|k/a|}$, where $k$ is as given in line 25 of the algorithm. Lines 6, 9, 18, and 21 represent unit-distance routing and so have a complexity of $O(1)$. Line 12 is an RAW in $2^b$-blocks and requires $O(q^2m)$ unit-routes. Line 13 takes $O(qm)$ time. Lines 15 and 29 are RARs in $2^b$-blocks and take $O(q^2m)$ time. Line 24 has complexity $O(qm)$. Lines 26 and 27 have complexity $O(q^4m \log m')$ and $O(q^2m')$ respectively. Finally, line 28 is a sort, and has complexity $O(q^2m)$. Since

$$q^4m' \log m' = q^3m \frac{bq - b}{2^{|b/a|^2}} = O(q^3m),$$

each iteration of the for loop takes $O(q^3m)$ time. The overall complexity of CONNECT ONES is therefore $O(q^4n)$.

An $O(n)$ algorithm for the connected ones problem can also be arrived at using other block combination strategies. For example, if we have found the sets of connected ones in each $2^{b+1}$-block then the sets for each $2^b$-block may be found by combining together the sets in the $2^b2^{b+4}$-blocks that make up the $2^b$-block. A boundary PE of a
2^{p-q}-block can be adjacent to at most q boundary PEs in the remaining $2^q - 1$ blocks making up a $2^p$-block. If we set up $\text{ADJ}(i, 0; q - 1)$ in a manner similar to the setting up of $\text{ADJ}(i)$ in procedure CONNECT ONES then we can make a call to CONNECT' (as in line 26) with $k$ replaced by $k'$ and 1 replaced by $q$. $k'$ is the maximum number of live nodes in a $2^k$-block.

Another possibility is to reduce the number of block combination steps to a constant. For example, if $q = 3$ then we can consider $n^{3/4} \times n^{3/4} \times n^{3/4}$ blocks. Each PE in such an $n^{9/4}$-block is ones-adjacent to at most 6 other PEs in that block (two PEs are ones-adjacent if they are adjacent and they both contain a one). Applying procedure CONNECT to each $n^{9/4}$-block in parallel takes $O(n^{3/4} \log n)$ time. The $n^{9/4}$-blocks can be combined together as before. The total number of boundary nodes in a $n^{3/4} \times n^{3/4} \times n^{3/4}$ block is 6 $n^{3/4}$. The number of blocks is $n^{9/4}$. So, the total number of live nodes (including roots) is at most $12n^{9/4}$. The time to concentrate these nodes is $O(n)$, and the time to run CONNECT' is $O(n^{3/4} \log n)$. So, the overall time is $O(n)$.

line \hspace{1cm} procedure CONNECT ONES (p)
  //Find connected 1's, $2^n = n^q = N$ is the number of PEs.//
  1. \hspace{1cm} \hspace{1cm} $R(i) := \infty$, $A(i) = 0$  // A is 0/1 pattern  //
  2. \hspace{1cm} \hspace{1cm} $R(i) := i$, $A(i) = 1$
  3. \hspace{1cm} \hspace{1cm} for $b := 0$ to $p - 1$ do //combine pairs of $2^b$-blocks.//
  4. \hspace{1cm} \hspace{1cm} $d := b \mod q$
  5. \hspace{1cm} \hspace{1cm} $E(i) := (A(i_0) \star A(i))$
  6. \hspace{1cm} \hspace{1cm} call ROUTE($E_d$, $d, -1$) //unit distance route to left half//
  7. \hspace{1cm} \hspace{1cm} $E(i) := (1 - (i_0) \star A(i) \star E(i))$
  8. \hspace{1cm} \hspace{1cm} LIVE($i) := E(i)$ //live PEs in left block//
  9. \hspace{1cm} \hspace{1cm} call ROUTE($E_d$, $d, +1$) //unit-distance route to right//
  10. \hspace{1cm} \hspace{1cm} LIVE($i) := 1$, $(E(i) = 1)$ //live PEs in right block//
  11. \hspace{1cm} \hspace{1cm} BORDER($i) := LIVE(i)$ //mark live border PEs//
  12. \hspace{1cm} \hspace{1cm} LIVE($R(i) \_1) := 1$, (LIVE($i) = 1)$
  13. \hspace{1cm} \hspace{1cm} call RANK($b + 1$) //rank live nodes. Let $H = \text{rank.}$//
  14. \hspace{1cm} \hspace{1cm} $H(i) := H(i) + 2^{\lfloor b/2 \rfloor} \star (i_{b-1} \_2) + 1$ //Add block bias//
  15. \hspace{1cm} \hspace{1cm} $R(i) \_H := R(R(i))$, (LIVE($i = 1)$) //update before move//
  16. \hspace{1cm} \hspace{1cm} ADJ($i) := \infty$ //build ADJ for live boundary PEs://
  17. \hspace{1cm} \hspace{1cm} $E(i) := H(i)$
  18. \hspace{1cm} \hspace{1cm} call ROUTE($E_d$, $d, -1)$
  19. \hspace{1cm} \hspace{1cm} ADJ($i) := E(i)$, $(i_0 = 0$ and BORDER($i) = 1)$
  20. \hspace{1cm} \hspace{1cm} $E(i) := H(i)$
  21. \hspace{1cm} \hspace{1cm} call ROUTE($E_d$, $d, +1$)
  22. \hspace{1cm} \hspace{1cm} ADJ($i) := E(i)$, $(i_0 = 1$ and BORDER($i) = 1)$
  23. \hspace{1cm} \hspace{1cm} ORG($i) := i$
  //Let record $G(i) = (R(i), \text{ADJ}(i), \text{LIVE}(i), \text{ORG}(i))$//
  24. \hspace{1cm} \hspace{1cm} call CONCENTRATE($b + 1$)
  25. \hspace{1cm} \hspace{1cm} $k := \min[b + 1, b - \lfloor b/q \rfloor + 2]$ //live nodes in a block $\leq 2^k$//
  26. \hspace{1cm} \hspace{1cm} call CONNECT'($k$, 1) //same as CONNECT without line 1//
  27. \hspace{1cm} \hspace{1cm} $R(i) \_H := \text{ORG}(R(i))$, (LIVE($i) = 1)$
  28. \hspace{1cm} \hspace{1cm} $R(\text{ORG}(i) = R(i)$ //move back//
  29. \hspace{1cm} \hspace{1cm} $R(i) \_H := R(R(i))$ //reduce the min-trees//
  30. \hspace{1cm} \hspace{1cm} end
  31. \hspace{1cm} \hspace{1cm} end CONNECT ONES

Algorithm 4
REFERENCES


