Programming Techniques
G.K. Manacher Editor

Computing Connected Components on Parallel Computers

D.S. Hirschberg
Rice University

A.K. Chandra
IBM Thomas J. Watson Research Center

D.V. Sarwate
University of Illinois

We present a parallel algorithm which uses n^2 processors to find the connected components of an undirected graph with n vertices in time $O(\log^2 n)$. An $O(\log^2 n)$ time bound also can be achieved using only $n\lceil n/\lceil \log_2 n \rceil \rceil$ processors. The algorithm can be used to find the transitive closure of a symmetric Boolean matrix. We assume that the processors have access to a common memory. Simultaneous access to the same location is permitted for fetch instructions but not for store instructions.

Key Words and Phrases: graph theory, parallel processing, algorithms, transitive closure, connected component

CR Categories: 5.25, 5.32, 6.22

Introduction

Parallel algorithms for solving various computational problems have received substantial attention recently, motivated in part by practical considerations. Among the many areas treated in the recent literature are sorting [2, 3, 7, 12, 15], the evaluation of arithmetic expressions, linear recurrences and polynomials [4, 8, 10], matrix algorithms [5, 6, 13], and graph theory [9, 14, 15]. In this paper we present a parallel algorithm CONNECT which determines the connected components of an undirected graph with n vertices in time $O(\log^2 n)$ using n^2 processors. Next, we modify the algorithm to demonstrate an observation due to F.P. Preparata and R.L. Probert, viz., $n\lceil n/\lg n \rceil$ processors[1] suffice to achieve a time bound of $O(\log^2 n)$. Finally, we show that CONNECT can be modified to compute the transitive closure of an $n \times n$ symmetric Boolean matrix in time $O(\log^2 n)$ using $n\lceil n/\lg n \rceil$ processors. We use the single instruction stream-multiple data stream (SIMD) model of parallel processors. It is assumed that the processors have access to a common memory, and that simultaneous access to the same location is permitted for fetch instructions but not for store instructions.

The Algorithm Connect

Let $V = \{0, 1, 2, \ldots, n-1\}$, and let $G = (V, E)$ denote an undirected graph with vertex set V and edge set E. We represent G by its adjacency matrix A which is an $n \times n$ symmetric Boolean matrix where $A(i, j) = 1$ if $(i, j) \in E$ and $A(i, j) = 0$ otherwise. A connected component of G is a maximal subgraph of G such that there exists a path between every pair of vertices in the subgraph. Each vertex belongs to exactly one connected component, and we use a vector D of length n to specify the connected components of G as follows. If $G_c = (V_c, E_c)$ is any connected component, then for all $i \in V_c$, $D(i)$ equals the least element of V_c. The parallel algorithm CONNECT given below iteratively computes the vector D from the adjacency matrix A for an undirected graph on n vertices.

Algorithm CONNECT

Input: The $n \times n$ adjacency matrix A for an undirected graph.
Output: The vector D of length n such that $D(i)$ equals the smallest-numbered vertex in the connected component to which i belongs.
Comment: Each of the following steps is executed in parallel for all i, $0 \le i < n$. The assignments in the various steps are considered to be done simultaneously for all i.

1. **for all** i **do** $D(i) \leftarrow i$
   do steps 2 **through** 6 **for** $\lg n$ **iterations**
2. **for all** i **do** $C(i) \leftarrow \min_j \{D(j) | A(i,j) = 1 \text{ AND } D(j) \neq D(i)\}$ if none then $D(i)$
3. **for all** i **do** $C(i) \leftarrow \min_j \{C(j) | D(j) = i \text{ AND } C(j) \neq i\}$ if none then $D(i)$
4. **for all** i **do** $D(i) \leftarrow C(i)$
5. **for** $\lg n$ **iterations do**
   for all i **do** $C(i) \leftarrow C(C(i))$
6. **for all** i **do** $D(i) \leftarrow \min\{C(i), D(C(i))\}$

[1] In this paper we denote $\lceil \log_2 n \rceil$ by $\lg n$.

The work of D.S. Hirschberg was supported by the National Science Foundation under Grant MCS-76-07683. The work of D.V. Sarwate was supported by the Joint Services Electronics Program under Contract DAAG-29-78-C-0016.

A preliminary version of this paper was presented at the 8th Annual ACM Symposium on the Theory of Computing, 1976.

Authors' addresses: D.S. Hirschberg, Department of Electrical Engineering, Rice University, Houston, TX 77001; A.K. Chandra, Computer Sciences Department, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598; D.V. Sarwate, Coordinated Science Laboratory, University of Illinois, Urbana, IL 61801.

An informal description of the actions of CONNECT is as follows. During the first iteration, the edges connecting each vertex to neighboring vertices are examined (steps 2 and 3), and sets of vertices which are known to be connected are identified (steps 4–6). In effect, each such set of vertices is merged into a "supervertex," which is specified by the vector D as follows. For each i in a supervertex, $D(i)$ equals the smallest-numbered vertex in the supervertex. In succeeding iterations, the edges connecting each *supervertex* to neighboring *supervertices* are examined in steps 2 and 3, and sets of supervertices are merged in steps 4–6. The process continues until all the vertices in a connected component have been merged into one gigantic supervertex. Further iterations have no effect on this supervertex because there are no edges connecting it to other supervertices. We will show later that lg n iterations are sufficient to collapse each connected component into a single supervertex.

The following definitions and lemmas are used in proving that Algorithm CONNECT finds the connected components of an undirected graph.

Definition. A *k-tree-loop*, $k \geq 0$, is a directed graph in which every vertex has outdegree 1 (i.e. exactly one edge leaves each vertex) and in which there is exactly one cycle, the length of the cycle being $k + 1$. A *tree-loop* is a k-tree-loop for some k.

Notice that a k-tree-loop has at least $k + 1$ vertices. The reason for the name tree-loop is that if one of the edges in the cycle is deleted, we obtain a rooted tree, the root being the vertex with no edge leaving it. The direction of all the edges in this tree is the reverse of that given in the usual definition (see e.g. [1, p. 52] of rooted trees. We are interested in the sequel in 1-tree-loops, which are defined by the vector C in the algorithm, and in a special case of 0-tree-loops (called clubs) which are defined by the vector D in the algorithm.

Definition. The *root* of a 0-tree-loop is the vertex v such that the edge (v, v) is the cycle of length 1. A *club* is a 0-tree-loop in which all the edges enter the root.

LEMMA 1. *Let $G_c = (V_c, E_c)$ denote a connected component of G such that $|V_c| \geq 2$ and define the function C: $V_c \rightarrow V_c$ by $C(i) = min\{j | A[i, j] = 1$ AND $j \neq i\}$. The function C defines a directed graph $G_c(C) = (V_c, E'')$ where $E'' = \{(i, C(i)) | i \in V_c\}$. Then $G_c(C)$ is a collection of 1-tree-loops, and the smallest-numbered vertex in each tree-loop is in the cycle of the tree-loop.*

PROOF. From the fact that C is a function, it is easy to see that $G_c(C)$ is a collection of tree-loops. Since $C(i) \neq i$, none of the tree-loops can be a 0-tree-loop. If a tree-loop in $G_c(C)$ is a k-tree-loop, let $v_0, v_1, v_2, \ldots, v_k, v_0$ denote the successive vertices in the cycle (i.e. $C(v_i) = v_{i+1}$ for $i = 0, 1, \ldots, k - 1$ and $C(v_k) = v_0$) where, without loss of generality, $v_0 = \min\{v_0, v_1, \ldots, v_k\}$. However, in the graph G_c, both v_0 and v_2 are neighbors of v_1. Hence $C(v_1) = v_2 > v_0$ which contradicts the definition of C except when $k = 1$. In this case, the two vertices in the loop are v_0 and v_1 with $C(v_0) = v_1$ and $C(v_1) = v_0$. A similar argument shows that the smallest-numbered vertex in the tree-loop must be in the cycle of the tree-loop. □

Let us make the usual definition of $C^k(v)$ as $C^1(v) = C(v)$ and $C^k(v) = C(C^{k-1}(v))$ for $k > 1$.

LEMMA 2. *Let C be defined as in Lemma 1 and let v be any vertex in a tree-loop of $G_c(C)$. Let v_0 and v_1 denote the two vertices in the cycle of the tree-loop. Then, for all $N \geq n - 2$, one of the two numbers $C^N(v)$ and $C^{N+1}(v)$ equals v_0 and the other equals v_1.*

PROOF. The result follows easily from the fact that the path from v to the nearer of v_0 and v_1 is of length at most $n - 2$. □

The two lemmas above are the basis of the method used in the algorithm to identify vertices in the same connected component. The function C sets up tree-loops in each connected component, the function C^N is computed iteratively, and then $D(v) \leftarrow \min\{C^N(v), C^{N+1}(v)\}$ sets up clubs (supervertices) with roots v_0. There are, of course, numerous bookkeeping details to be settled, and we take up these in the proof of the following theorem.

THEOREM. *Algorithm CONNECT computes the connected components of the undirected graph G specified by the symmetric Boolean matrix A.*

PROOF. It is easy to verify that for the trivial connected components consisting of isolated vertices i, $D(i)$ is set to i at step 1 and remains unchanged throughout the execution of the algorithm. In the remainder of this proof, we consider connected components with two or more vertices only.

Let $G(D) = (V, E_D)$ denote the directed graph defined by D where $E_D = \{(i, D(i)) | i \in V\}$. After the execution of step 1, and just prior to the execution of step 2, $G(D)$ satisfies the following properties:

(i) $G(D)$ is a set of clubs (with disjoint vertex sets).
(ii) The root of each club is the smallest-numbered vertex in the club.
(iii) The vertex set of any club is a subset of the vertex set of some connected component.

We show that if $G(D)$ satisfies properties (i)–(iii) just prior to the execution of step 2, then after executing steps 2–6, the new function D (computed at step 6) is such that the new $G(D)$ also has properties (i)–(iii). Furthermore, the numbers of clubs in each connected component is reduced by a factor of at least 2, provided that there were at least two clubs in the connected component just prior to step 2.

It is instructive to observe what happens during the first iteration of steps 2–6. Since D is the identity function, the function C defined at step 2 is exactly the function of Lemma 1, and sets up 1-tree-loops in each connected component of G. Step 3 does not change C because the only j satisfying $D(j) = i$ is i itself, and $C(i) \neq i$. In step 4, the function C is copied into D, while in step 5, C is transformed to C^N where $N = 2^{\lg n} \geq n$. Step 6 sets $D(i)$ to $\min\{C^N(i), D(C^N(i))\}$ which is the same as $\min\{C^N(i), C^{N+1}(i)\}$. It follows from Lemma 2 that for all i, $D(i)$ equals the smallest-numbered vertex in

the tree-loop that contained i. Thus the set of vertices in each tree-loop has been merged into a club. It is easy to see that after the first iteration, $G(D)$ satisfies properties (i)–(iii). Since each 1-tree-loop contained at least two vertices, the number of clubs in each nontrivial connected component is no more than half the number of vertices (clubs) that it contained originally.

As mentioned earlier, further iterations of steps 2–6 merge supervertices, i.e. clubs. Connections between supervertices may be defined as follows. Let V_r denote the set of roots of clubs in $G(D)$ and let $G_r = (V_r, E_r)$ denote an undirected graph where for $i \neq j$, $(v_i, v_j) \in E_r$ if and only if there exist vertices v_i' and v_j' in the clubs of v_i and v_j, respectively, such that $(v_i', v_j') \in E$. In other words, supervertices are neighbors if and only if there is an edge connecting some pair of member vertices. The function C is set up in steps 2 and 3. In step 2, each vertex i examines the club memberships of its neighbors and sets $C(i)$ to the smallest-numbered neighboring club. In step 3, each $i \in V_r$ examines its own club members (specified by $D(j) = i$) and picks the smallest-numbered of all the smallest-numbered clubs that the members found. In short, the function $C: V_r \rightarrow V_r$ is such that for all $i \in V_r$, $C(i)$ equals the smallest-numbered vertex that is adjacent to i in G_r. As in Lemma 1, C defines a collection of 1-tree-loops on G_r. Next let us consider vertices $i \notin V_r$. For such vertices, there is no j such that $D(j) = i$ and thus at step 3, $C(i)$ is reset to $D(i)$. Hence, $C: V \rightarrow V$ defines a collection of 1-tree-loops on G because each nonroot is pointing to a root and the roots are in 1-tree-loops. It follows (as in the discussion of the first iteration of steps 2–6) that after step 6, the new function D is such that $G(D)$ satisfies properties (i)–(iii). Furthermore, each 1-tree-loop involves two or more vertices in G_r, i.e. two or more clubs, and hence in each connected component that contained at least two clubs, the number of clubs is decreased by a factor of at least 2.

From the above discussion, it is clear that the number of clubs in each connected component decreases by a factor of at least 2 at each iteration until the connected component consists of a single club. It is easy to verify that further iterations do not affect such single clubs. Since there are at most n vertices (clubs) to begin with, lg n iterations suffice to reduce each connected component to a single club, where club membership is defined by D. □

We have shown that CONNECT computes the connected components of the graph G specified by the symmetric Boolean matrix A. The transitive closure of A, denoted by A^*, is given by $A^*(i, j) = 1$ if and only if there is a path in G from i to j, i.e. if and only if i and j are in the same connected component. Hence we obtain an algorithm for the transitive closure of symmetric matrices by adding the following step to CONNECT:

7. **for all** i, j **do if** $D(i) = D(j)$ **then** $A^*(i, j) \leftarrow 1$

and by changing the input-output specifications appropriately.

Time and Processor Bounds

The main loop of the program is executed lg n times, while within the loop, the iteration at step 5 is executed lg n times. Thus the algorithm requires $\Omega(\log^2 n)$ time regardless of the number of processors used. Let us suppose that n^2 processors are available. Steps 1, 4, and 6 require only $O(1)$ time whenever $\Omega(n)$ processors are available, while step 5 requires $O(\log n)$ time with the same processor requirements. We now show that steps 2 and 3 also can be programmed to execute in time $O(\log n)$ to give an $O(\log^2 n)$ time bound using n^2 processors. The program for step 2 is

Step 2. The following steps are performed in parallel for $0 \leq i, j < n$.

2(a) **For all** i, j **do**
  if $A(i, j) = 1$ AND $D(j) \neq D(i)$ **then** $Temp(i, j) \leftarrow D(j)$
  else $Temp(i, j) \leftarrow \infty$

2(b) **For** $k \leftarrow 0$ **until** $(\lg n) - 1$ **do**
  for all i, j **do** $Temp(i, j) \leftarrow$
  $\min\{Temp(i, j),$
  $Temp(i, j + 2^k \bmod n)\}$

2(c) **For all** i **do**
  if $Temp(i, 0) = \infty$ **then** $C(i) \leftarrow D(i)$
  else $C(i) \leftarrow Temp(i, 0)$

Here ∞ means any number exceeding $n - 1$. In step 2(a) the numbers whose minimum is to be computed are stored in the array *Temp*. In step 2(b), the minimum is found as follows. At the first iteration $Temp(i, 0)$ is compared with $Temp(i, 1)$ as is $Temp(i, 2)$ with $Temp(i, 3)$, $Temp(i, 4)$ with $Temp(i, 5)$ ... etc. At the second iteration, $Temp(i, 0)$ is compared with $Temp(i, 2)$, $Temp(i, 4)$ with $Temp(i, 6)$ etc. The former comparison finds $\min\{Temp(i, j),\ 0 \leq j \leq 3\}$, while the latter finds $\min\{Temp(i, j),\ 4 \leq j \leq 7\}$ etc. Thus, in lg n iterations the minimum is found. The method is simple but wasteful of processors in that, for example, the result of comparing $Temp(i, 1)$ with $Temp(i, 2)$ is not used at all. Obviously, for each value of i, $\lceil n/2 \rceil$ processors would suffice for the first iteration, $\lceil \lceil n/2 \rceil / 2 \rceil$ for the second etc. The program for step 3 is similar and will not be stated separately. The net result is the following theorem.

THEOREM. *Algorithm CONNECT finds the connected components of an undirected graph with n vertices in time $O(\log^2 n)$ using n^2 processors.*

COROLLARY. *The transitive closure of an $n \times n$ symmetric Boolean matrix can be found in time $O(\log^2 n)$ using n^2 processors.*

The reduction in the number of processors that was observed by Preparata and Probert occurs as follows. We partition the integers $\{0 \leq j < n\}$ into $\lceil n/\lg n \rceil$ subsets of the form $\{k \lg n \leq j < (k + 1) \lg n\}$ where $0 \leq k < \lceil n/\lg n \rceil$. Each such subset (except possibly the one with $k = \lceil n/\lg n \rceil - 1$) has lg n elements. The idea is to compute the n^2 entries of the array *Temp* in time $O(\log n)$ using $n\lceil n/\lg n \rceil$ processors. In order to compute the minimum value of $Temp(i, j)$ for $0 \leq j < n$, we first compute the minimum values for j in the range $k \lg n \leq j < (k + 1) \lg n$. These are found in time

$O(\log n)$ via sequential search. Then, the minimum of the $\lceil n/\lg n \rceil$ candidate minima is found (as in step 2(b) above) in time $O(\log n - \log\log n)$ using $\lceil n/\lg n \rceil$ processors at each step. The grubby details are as follows.

Step 2. The following steps are performed in parallel for $0 \le i < n$, and $0 \le k < \lceil n/\lg n \rceil$.

2(a) **For** $l \leftarrow 0$ **until** $(\lg n) - 1$ **do**
  for all i, k **do**
    if $A(i, l + k \lg n) = 1$ AND $D(i) \neq D(l + k \lg n)$
      then $Temp(i, l + k \lg n) \leftarrow D(l + k \lg n)$
      else $Temp(i, l + k \lg n) \leftarrow \infty$

2(b) **For** $l \leftarrow 1$ **until** $(\lg n) - 1$ **do**
  for all i, k **do**
    $Temp(i, k \lg n) \leftarrow \min\{Temp(i, k \lg n), Temp(i, l + k \lg n)\}$

2(c) **For** $l \leftarrow 0$ **until** $(\lg\lceil n/\lg n\rceil) - 1$ **do**
  for all i, k **do**
    $Temp(i, k \lg n) \leftarrow \min\{Temp(i, k \lg n), Temp(i, (k + 2^l)\lg n \bmod n)\}$

2(d) **For all** i **do**
  if $Temp(i, 0) = \infty$ **then** $C(i) \leftarrow D(i)$
    else $C(i) \leftarrow Temp(i, 0)$

In the above program, we have ignored the fact that one of the $\lceil n/\lg n \rceil$ subsets may contain fewer than $\lg n$ elements. One way around this is to pad the arrays A, C, and D approximately. Another possibility is to replace $l + k \lg n$ by $(l + k \lg n) \bmod n$. The program for step 3 is similar and we have the following theorem.

THEOREM. *Algorithm CONNECT finds the connected components of an undirected graph with n vertices in time $O(\log^2 n)$ using $n\lceil n/\lg n \rceil$ processors.*

COROLLARY. *The transitive closure of an $n \times n$ symmetric Boolean matrix can be found in time $O(\log^2 n)$ using $n\lceil n/\lg n \rceil$ processors.*

Remark. In [5], it is shown that the transitive closure of an arbitrary $n \times n$ Boolean matrix can be found in time $O(\log^2 n)$ using $O(n^{\log_2 7}/\log n)$ processors. Although the exponent of n may be reduced slightly by using some recent results of Pan [11], it is clear that symmetry reduces processor requirements significantly for the transitive closure problem.

Received October 1975; revised October 1978

References

1. Aho, A.V., Hopcroft, J.E., and Ullman, J.D. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.
2. Batcher, K.E. Sorting networks and their applications. Proc. AFIPS 1968 SJCC, Vol. 32, AFIPS Press, Montvale, N.J., pp. 307–314.
3. Baudet, G., and Stevenson, D. Optimal sorting algorithms for parallel computers. *IEEE Trans. Comptrs. C-27* (Jan. 1978), 84–87.
4. Brent, R.P. The parallel evaluation of general arithmetic expressions. *J. ACM 21*, (April 1974), 201–206.
5. Chandra, A.K. Maximal parallelism in matrix multiplication. IBM Tech. Rep. RC6193, Sept. 1976.
6. Csanky, L. Fast parallel matrix inversion algorithms, *SIAM J. Comptng. 5* (Dec. 1976), 618–623.
7. Hirschberg, D.S. Fast parallel sorting algorithms. *Comm. ACM 21*, 8 (Aug. 1978), 657–661.
8. Hyafil, L., and Kung, H.T. The complexity of parallel evaluation of linear recurrences. *J. ACM 24*, (July 1977), 513–521.
9. Levitt, K.N., and Kautz, W.H. Cellular arrays for the solution of graph problems. *Comm. ACM 15*, (Sept. 1972), 789–801.
10. Munro, I., and Paterson, M. Optimal algorithms for parallel polynomial evaluation. *J. Comp. Syst. Sci. 7* (1973), 183–198.
11. Pan, V.Y. Strassen's algorithm is not optimal. Proc. 19th Annu. Symp. on Foundations of Comptr. Sci., 1978, pp. 166–176.
12. Preparata, F.P. New parallel-sorting schemes. *IEEE Trans. Comptrs. C-27* (July 1978), 669–673.
13. Preparata, F.P., and Sarwate, D.V. An improved parallel processor bound in fast matrix inversion. *Inf. Proc. Letters 7* (April 1978), 148–150.
14. Reghbati, E., and Corneil, D.G. Parallel computations in graph theory. *SIAM J. Comptng. 7* (May 1978), 230–237.
15. Savage, C.D. Parallel algorithms for graph theoretic problems. Ph.D. Th., U. of Illinois, Urbana, Ill., Aug. 1977.
16. Thompson, C.D., and Kung, H.T. Sorting on a mesh-connected parallel computer. *Comm. ACM 20*, 4 (April 1977), 263–271.