CONNECTED COMPONENTS ALGORITHMS FOR MESH-CONNECTED PARALLEL COMPUTERS

STEVE GODDARD, SUBODH KUMAR, AND JAN F. PRINS

ABSTRACT. We present a new CREW PRAM algorithm for finding connected components. For a graph G with n vertices and m edges, algorithm $\mathcal{A}_0$ requires at most $O(\log n)$ parallel steps and performs $O((n+m)\log n)$ work in the worst case. The advantage our algorithm has over others in the literature is that it can be adapted to a 2-D mesh-connected communication model in which all CREW operations are replaced by $O(\log n)$ parallel row and column operations without increasing the time complexity.

We present the mapping of $\mathcal{A}_0$ to a mesh-connected computer and describe two implementations, $\mathcal{A}_1$ and $\mathcal{A}_2$. Algorithm $\mathcal{A}_1$, which uses an adjacency matrix to represent the graph, performs $O(n^2 \log n)$ work. Hence, it only achieves work efficiency on dense graphs. The second implementation, $\mathcal{A}_2$, uses a sparse representation of the adjacency matrix and again performs $O(\log n)$ row and column operations but reduces the work to $O((m+n)\log n)$ on all graphs.

We report MasPar MP-1 performance figures for implementations of the algorithms described. The implementations are exercised on a variety of parametrically generated graphs, differing in structure and connectivity. These graphs are generated externally and read in as input for the algorithms, permitting comparison of different implementations on identical graphs.

1. INTRODUCTION

The problem of rapidly finding the connected components of an undirected graph presents some substantial challenges for parallel computers.

First, parallel algorithms for this problem developed for the PRAM model make extensive use of concurrent reads and writes (CRCW) to the shared memory, and this abstraction is poorly supported by current parallel computers. Hence great care has to be taken to minimize the impact of these operations.

Second, the standard sequential algorithm for this problem (based on depth-first search) has optimal time complexity and small multiplicative constants, using only a few operations per vertex and edge in the graph. Parallel algorithms for this problem rely on completely different techniques, and in many cases do not have optimal work complexity or else perform a much larger number of operations per vertex and edge. Thus achieving high absolute performance from parallel implementations can be difficult.

In this paper we develop a new parallel algorithm for connected components that is designed for the 2-D mesh communications model instead of the shared memory

Date: June 1996.

1991 *Mathematics Subject Classification.* 68Q22; Secondary 68Q22,68R10 .

Key words and phrases. Connected Components Algorithms, Mesh-Connected Computers, MasPar.

CRCW model and has variants with reasonable work efficiency for sparse and dense graphs.

The initial presentation of the algorithm, $\mathcal{A}_0$, is for the CREW-PRAM model of computation and is based on ideas found in the CRCW-PRAM algorithms for sparse graphs developed by Shiloach *et al.* in [SV82, AS87]. For a graph G with n vertices and m edges, $\mathcal{A}_0$ requires at most $O(\log n)$ parallel steps and performs $O((n+m)\log n)$ work (hence, like [SV82, AS87], is not quite work efficient).

$\mathcal{A}_0$ differs from [SV82, AS87] in that it can be adapted to a 2-D mesh-connected communication model in which all CREW operations are replaced by parallel row and column operations. In the case of the MasPar MP-1 and MP-2 machines that are the implementation targets for this work, row and column operations can use the high-bandwidth mesh network and offer better performance than concurrent read operations on global memory, which use the lower-bandwidth general-routing network. Even in machines like the Intel Paragon and the Cray T3D/T3E, where the general-routing network is based on the mesh connections, the regularity of the communication pattern and the elimination of read contention can still favor the use of row and column operations.

Algorithm $\mathcal{A}_1$ is the adaptation of $\mathcal{A}_0$ to the mesh, and is based on an adjacency matrix representation of G. This algorithm performs $O(\log n)$ parallel row and column reduction and broadcast operations, but performs $O(n^2 \log n)$ work, hence achieves very poor work efficiency on sparse graphs. Since sparse graphs are typical in applications requiring high-speed determination of connected components (see [Gre93]), this is unsatisfactory.

Algorithm $\mathcal{A}_2$ uses a sparse representation of the adjacency matrix and again performs $O(\log n)$ row and column operations but reduces the work to $O((m+n)\log n)$ on all graphs. A cyclic decomposition of the underlying adjacency matrix over processors, tends to distribute the sparse edge set uniformly over processors while insuring that the communication structure of the row and column operations is preserved.

On a 4096 node graph, our implementation of $\mathcal{A}_1$ on an 8,192 processor MasPar MP-1 (at approximately 0.2 Mops/sec per processor) achieves a performance varying from 10^5 to 10^8 edges per second with increasing density of the graph. Our implementation of $\mathcal{A}_2$ improves on $\mathcal{A}_1$ by about a factor of three for sparse graphs.

The rest of the paper is organized as follows. We introduce the basic PRAM algorithm $\mathcal{A}_0$ in section 2. Section 3 describes the implementation of $\mathcal{A}_1$ and $\mathcal{A}_2$ under a mesh-connected communication model. Section 4 reports on the implementation of $\mathcal{A}_1$ and $\mathcal{A}_2$ on the MasPar MP-1 and gives performance statistics. We discuss other connected components algorithms and compare their results with $\mathcal{A}_2$ in section 5. Section 6 shares our plans for further improvements to the algorithms. Finally we present our conclusions and ideas for continuing research in section 7.

2. Main Algorithm

Let $G = (V, E)$ be an undirected graph, with vertices $V = \{1, \ldots, n\}$ and $|E| = m$. For $u,v \in V$, there is a *path* between u and v, written as $u \rightsquigarrow v$, iff there exists a sequence of vertices $[w_1 \ldots w_k]$ such that $w_1 = u, w_k = v,$ and $\forall\, i : 1 \leq i < k :: (w_i, w_{i+1}) \in E$.

The connected component problem is to compute for each $v \in V$ a label $P(v)$ such that $\forall\, u, v \in V : u \rightsquigarrow v$ iff $P(u) = P(v)$. We require that any labeling

function satisfy $P : V \to V$ and $\forall\ u, v \in V : P(u) \leq u$. Under these conditions P is a *parent function* and induces a forest of *rooted trees* on V, with each tree rooted by some vertex r for which $r = P(r)$. A *rooted star* is a tree T of height one with a root r such that $P(v) = r$ for each vertex $v \in T$. Our solution to the connected components problem sets $P(v)$ to be the smallest vertex reachable from v, which defines a rooted star for each component.

Our PRAM algorithm, $\mathcal{A}_0$, starts with $P(v) = \min(v, \min\{u \mid (u, v) \in E\})$ for all $v \in V$ (i.e. for each v, the smallest vertex within distance one of v), and iteratively improves P until it converges on the solution. Note that the initial parent function may contain a tree of height as much as $n - 1$. Each iteration of the algorithm changes P as follows.

- *opportunistic pointer jumping:* Define the *chain* from a vertex $u \in V$ to be the sequence of vertices from u to the root of the tree containing u. The opportunistic pointer jumping step attempts to decrease the height of chains through a pointer doubling operation for each vertex u of the form $P'(u) := P(P(u))$, shrinking the height h of the chain to $\lceil \frac{h}{2} \rceil$. However, u can perform pointer jumping either through its own chain or through that of one of its neighbors in G. The neighbor v of u with the least value for $P(v)$ determines the chain into which u performs a pointer jumping step (see Figure 1). Therefore a vertex may switch trees or leave one rooted star for another as part of this step.

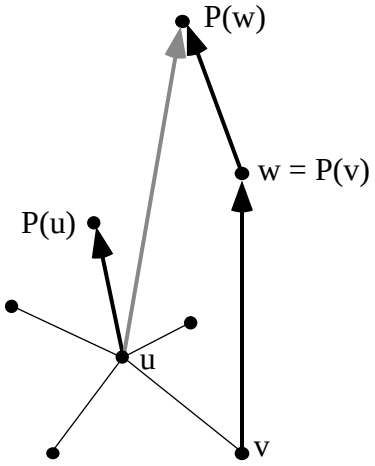

FIGURE 1. Opportunistic Pointer Jumping: Vertex u finds that of all of its neighboring vertices, v has the smallest numbered parent. Therefore, vertex u changes its parent to $\mathrm{P}(w) = \mathrm{P}(\mathrm{P}(v))$ rather than $\mathrm{P}(\mathrm{P}(u))$.

- *tree hanging:* If a vertex v with parent $u = P(v)$ switches chains so that $P'(v)$ ends up a smaller value than $P'(u)$, then $P'(u)$ is switched to $P'(v)$. There may be multiple children of u that can improve $P'(u)$, in which case $P'(u)$ is set to the minimum of the new parents of all its children (see Figure 2). Subsequently a single normal pointer jumping step is used to ensure that rooted stars can be hung onto a tree without changing the tree's height. This step is not necessary for correct execution of $\mathcal{A}_0$, but simplifies the time

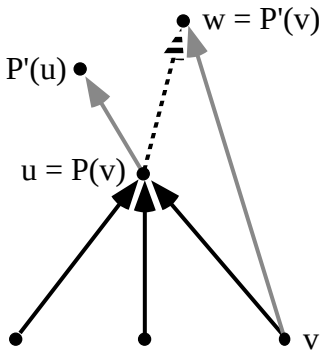

FIGURE 2. Tree Hanging: Vertex u finds that its former child, vertex v, has found a smaller numbered parent during the opportunistic pointer jumping step (i.e., $w = \mathrm{P}'(v) < P'(u)$). The tree hanging step changes the parent of u from $\mathrm{P}'(u)$ to $w = \mathrm{P}'(v)$ if w is the minimum of the new parents of the old children of u.

complexity proof. The tree hanging operation is critical for rapid convergence and plays the same role as the grafting operation of [SV82].

The pseudo code of $\mathcal{A}_0$ follows.

FOREACH vertex u **IN** G
  $P(u) := \min\{u, \min\{v \mid \text{vertex } v \text{ is adjacent to } u \text{ in G}\}\}$
REPEAT
  FOREACH vertex u **IN** G /* *Opportunistic Pointer Jumping* */
    $OldP(u) := P(u)$
    $P'(u) := P(\min\{P(u), \min\{P(v) \mid \text{vertex } v \text{ is adjacent to vertex } u \text{ in G}\}\})$
  FOREACH vertex u **IN** G /* *Tree hanging* */
    $P(u) := \min\{P'(u), \min\{P'(v) \mid P(v) = u\}\}$
  FOREACH vertex u **IN** G /* *Normal Pointer Jumping* */
    $P(u) := P(P(u))$
UNTIL $P = OldP$

Algorithm $\mathcal{A}_0$ uses the combination of opportunistic pointer jumping and tree hanging to pull stagnant stars into other trees that comprise the same connected component. Since vertices are always trying to decrease their parent function, eventually all trees of a connected component are combined and contracted to a single rooted star. While the Shiloach-Vishkin algorithms of [SV82, AS87] must be very particular about how trees are grafted, $\mathcal{A}_0$ hangs a tree on any lower numbered vertex without concern for its height in the tree.

2.1. **Correctness.** Algorithm $\mathcal{A}_0$ terminates when all vertices of a connected component have the same parent.

Theorem 2.1. *On termination of* $\mathcal{A}_0$, $\forall u, v \in V$, $P(u) = P(v) \iff u \leftrightsquigarrow v$

Proof ($\Rightarrow$): $\forall u \in V$, $u \leftrightsquigarrow \mathrm{P}(u)$ is an invariant of the loop: each change to $\mathrm{P}(u)$ preserves $u \leftrightsquigarrow \mathrm{P}(u)$. Hence if $\mathrm{P}(u) = \mathrm{P}(v)$, then $u \leftrightsquigarrow \mathrm{P}(u) = \mathrm{P}(v) \leftrightsquigarrow v$ and therefore $u \leftrightsquigarrow v$.
($\Leftarrow$): By contradiction. Assume $\mathcal{A}_0$ has terminated with $\mathrm{P}(u) \neq \mathrm{P}(v)$ and $u \leftrightsquigarrow v$. Then $\exists\ (w_i, w_{i+1})$ on $u \leftrightsquigarrow v$ such that $\mathrm{P}(w_i) \neq \mathrm{P}(w_{i+1})$. This means that $\mathrm{P}(w_i) <$

$\mathrm{P}(w_{i+1})$ or $\mathrm{P}(w_i) > \mathrm{P}(w_{i+1})$. In either case, the opportunistic pointer jumping step would change $\mathrm{P}(w_i)$ or $\mathrm{P}(w_{i+1})$ so that the termination condition (OldP = P) could not hold. This contradicts the assumption that $\mathcal{A}_0$ has terminated. ■

2.2. **Complexity.** Termination of $\mathcal{A}_0$ is guaranteed because each iteration of $\mathcal{A}_0$ satisfies $\forall u \in V : P(u) \leq OldP(u)$, which can be established by observing that (1) $P(u) \leq u$ is an invariant of $\mathcal{A}_0$ and (2) each change to $P(u)$ can only decrease its value or leave it the same. Since P is strictly decreasing on each iteration on which $\mathcal{A}_0$ does not terminate, and is bounded below by the connected components labeling, $\mathcal{A}_0$ must terminate.

In practice, we have observed that the number of iterations of the outer loop is very small, but the complexity is $O(\log n)$. To see this, observe that $\log n$ steps will reduce any tree formed by P, that does not have another tree hook onto it, to a rooted star. Any chain that did have trees hook onto it, may take another $\log n$ steps to shrink to height one. Note that the number of trees never increases. In most iterations both the number of trees and the height of each tree decreases. In the worst case, if all trees shrink to rooted stars without forming complete components, it takes one more step to get the final connected components of P and another $\log n$ iterations to shorten these new chains to rooted stars.

3. Implementation on a mesh

The connected components algorithm of section 2 has the nice property that it can be mapped to a mesh-connected computer without needing costly concurrent read operations in a shared memory. The pseudo code presented below represents the mapping to the mesh utilizing only the row and column communication primitives shown below, which are quite efficient on computers like the MasPar. The graph edges are stored in the adjacency matrix A. Matrices Q and M are used to store intermediate results. The functions P, P' and $OldP$ from $\mathcal{A}_0$ are each represented as matrices in which the function values are replicated either on each row (P and $OldP$) or each column (P').

During the opportunistic pointer jumping and tree hanging steps, the parent values are stored in the columns of P', such that the parent of vertex j is stored in column j. The normal pointer jumping step then returns the parent values to the rows of P for the next iteration. All values needed at any step in the algorithm are in the row or column of one of the matrices. Hence, only row or column communications is required.

Functions:

$MinCol(M^{n\times n}) : Q^{n\times n}$
$MinRow(M^{n\times n}) : Q^{n\times n}$
$MinNeighbor(A^{n\times n}, u) : v$

$MinCol()$ finds the minimum value in each column of the matrix M and copies these values to every entry of the respective columns of the return matrix Q. $MinRow()$ performs the respective operations on the rows of the matrix. The third function, $MinNeighbor()$, returns the minimum of u and the minimum vertex adjacent to u in the adjacency matrix A (all necessary information for this function can be found in column u). Note that all three of these functions can be implemented quite efficiently on most mesh computers.

Variables:

$A^{n\times n}$ adjancency matrix representation of the graph
$P', P^{n\times n}$ values of parent function $P()$
$M, Q^{n\times n}$ storage matrices

Initializations:

$$A(i,j) := \begin{cases} \text{True if } (i,j) \in E \text{ or } i = i \\ \text{False otherwise} \end{cases}$$
$$P_{i,j} := MinNeighbor(A, i)$$

Pseudo-Code for Connected Components Algorithm on a Mesh:
REPEAT
$OldP := P$
/ begin opportunistic pointer jumping */*
FORALL i, j **IN** $[1..n], [1..n]$
$$M(i,j) := \begin{cases} P_{ij} \text{ if } A_{ij} \\ \infty \text{ otherwise} \end{cases}$$
$Q := MinCol(M)$
FORALL i, j **IN** $[1..n], [1..n]$
$$M(i,j) := \begin{cases} P_{ij} \text{ if } Q_{ij} = i \\ \infty \text{ otherwise} \end{cases}$$
$P' := MinCol(M)$
/ begin tree hanging */*
FORALL i, j **IN** $[1..n], [1..n]$
$$M(i,j) := \begin{cases} P'_{ij} \text{ if } P_{ij} = j \\ \infty \text{ otherwise} \end{cases}$$
$Q := MinCol(M)$
FORALL i, j **IN** $[1..n], [1..n]$
$P'(i,j) := min(P'_{ij}, Q_{ij})$
/ begin normal pointer jumping */*
FORALL i, j **IN** $[1..n], [1..n]$
$$M(i,j) := \begin{cases} P'_{ij} \text{ if } i = j \\ \infty \text{ otherwise} \end{cases}$$
$Q := MinRow(M)$
FORALL i, j **IN** $[1..n], [1..n]$
$$M(i,j) := \begin{cases} P'_{ij} \text{ if } Q_{ij} = j \\ \infty \text{ otherwise} \end{cases}$$
$P := MinRow(M)$
UNTIL $P = OldP$

The opportunistic pointer jumping and tree hanging steps both serve to move vertices closer to the correct root for the connected component to which they belong. A vertex may switch from following one parent to another if it finds that one of its neighbors has found a smaller vertex (or root). These two steps serve to group trees into connected components while reducing each connected component to a rooted star. In contrast the normal pointer jumping step of the algorithm can only shrink a tree to a rooted star, something the opportunistic pointer jumping step also does. Hence, in practice we find that replacing the normal pointer jumping step

with another opportunistic pointer jumping step improves the performance of the algorithm. Since the next iteration will do another opportunistic pointer jumping step, we can further simplify the implementation of the algorithm by dropping the last pointer jumping step so the loop only consists of one opportunistic pointer jumping step followed by tree hanging.

A graph can be represented as a matrix or an adjacency list. The variation in representation gives rise to different behavior. While the first representation is well suited to dense graphs it is quite wasteful for sparse graphs. Algorithm $\mathcal{A}_1$, described in section 3.1, stores the graph as an adjacency matrix. Algorithm $\mathcal{A}_2$, described in section 3.2, stores the graph as an adjacency list. $\mathcal{A}_1$ is better suited for dense graphs while $\mathcal{A}_2$ is better suited for sparse graphs as shown in section 4.

3.1. **Algorithm $\mathcal{A}_1$: Matrix Representation.** Consider a mesh of $p \times p$ processors (or PEs, for processing elements). In the following discussion we assume that wrap-around connections exist on the mesh, though it is not essential to the algorithm.

We combine matrices P and A by storing P_{ij} in A_{ij} if $(i,j) \in E$ and ∞ otherwise. Call this matrix M. We then use the diagonals of M to store OldP(), P'_{ij}. The matrix M is distributed by mapping $M(i,j)$ to PE($i \bmod p, j \bmod p$). Combining this mapping with our algorithm, PE(a,b) only accesses $M(i,j)$ if either $i \bmod p = a$ or $j \bmod p = b$. In both cases the required data is found within the row or column.

The work complexity of algorithm $\mathcal{A}_1$ is $O(n^2 \log n)$, each PE does $O((\frac{n}{p})^2)$ work per iteration. The total number of iterations is $O(\log n)$ and the total number of PEs is p^2.

3.2. **Algorithm $\mathcal{A}_2$: Adjacency List Representation.** If most of the elements in M are ∞, we waste space and time performing operations on the adjacency matrix. So instead of distributing the entire matrix we can distribute only the non ∞ entries in the matrix — just the edges. Our approach is to store a *sparse* adjacency matrix. We use the cyclic decomposition from $\mathcal{A}_1$: an edge (u,v) is stored at the processor PE($u \bmod p, v \bmod p$), but we only store edges present in the graph and elide the ∞ values. Thus we have a list of (u,v) values at each processor. We can implement the row and column minimum operations by merging lists between processors, retaining the minimum u value for elements with equal v values or vice versa (for this to yield a constant cost per edge, we must keep the lists in sorted order). This implementation doesn't spoil the communication characteristics of the algorithm since the required information can still be found in the row or the column, but we may end up with load balancing problems if the edges are not uniformly distributed in the graph (see Section 6.1).

The adjacency list is constructed and pre-processed in parallel at each processor. The list is sorted using $min(u,v)$ as the key, and the $RowMin$ and $ColMin$ operations work locally on the list before communicating with other processors. The list pre-processing and the higher operation count per row or column operation increases the per edge cost compared to $\mathcal{A}_1$, but this is offset by the reduced work (relative to the n^2 adjacency matrix when the graph is sufficiently sparse).

The work complexity of this implementation is $O((n+m)\log n)$, since we represent only the m edges and the n values for P, P', etc. The number of iterations stays the same. In our current implementation of $\mathcal{A}_2$, each processor still does

some work for vertices that do not have adjacent edges mapping onto that processor; eliminating this work will further improve the timings.

4. Performance Results

We tested the performance of algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ on star, chain, tertiary, 2D-mesh, 3D-mesh and random shaped graphs. We found that star graphs provided the best results while long chains yielded the worst. Since the star and chain graphs were created to exploit strengths and weakness specific to our algorithm, we do not present those timing results. We have chosen instead (for brevity) to report results measured on the canonical graph benchmarks of random and tertiary graphs and variations thereof.

Many algorithms are quite sensitive to the structure of the graph. For example, a class of graphs that are recursively defined as graphs of vertices which are themselves graphs (with different density and structure) pose difficulties for some connected component algorithms. Such graphs are sometimes called *hard graphs*. Due to the dependence of our algorithm on a chain's length, and not the actual structure of the graph, such graphs do not negatively impact the performance of our algorithms. In some cases our algorithms execute faster on hard graphs than the 'simple' graphs reported in this paper because hard graphs have more dense connections that shorten the chains in the graph.

Section 4.1 defines the graph terms we use to describe our suite of benchmark graphs. Section 4.2 describes how we built our test graphs. Section 4.3 addresses how graphs are read into the MasPar and distributed. Sections 4.4, 4.5, and 4.6 address the performance of algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ on random, tertiary, and grid graphs respectively. The timing tests reported in this paper (except the sequential algorithm) were performed on a 8192 processor MasPar MP-1.

4.1. Definitions.

A graph in which each pair of distinct vertices is joined by an edge is called a *complete graph*. The number of edges in a graph as a percentage of complete cover is called its *density*: e.g., a p% dense graph has $\frac{p}{100} \cdot \frac{n(n-1)}{2}$ edges, n being the number of vertices in the graph. The number of edges incident on a vertex is called its *vertex degree* and the degree of the graph is its maximum vertex degree. A *2D* graph is a subset of a two-dimensional toroidal grid. The neighbors of a vertex in a 2D graph form a subset of the four neighbors on such a grid [Gre93]. Similarly, a *3D graph* is a subset of a three-dimensional toroidal grid [Gre93]. The vertices of a *random graph* are joined at random, and unless otherwise noted, the number of components is not constrained; it is a function of the random edge generation. Each vertex of a *tertiary* graph has degree 3. When no duplicate edges are allowed, a tertiary graph has $1.5n$ edges.

4.2. Generating Benchmark Graphs.

Initially, we created graphs 'on the fly' as other research projects had done [KLCY94, HRD94]. However, we found that this method presented two problems. First, duplicate edges were created which inflated the 'actual' number of edges, resulting in better performance for our algorithms. The second problem was that creating graphs on the fly precluded the possibility of accurately comparing algorithms implemented on different machines (or by other groups).

We have created a tool, *mkgraph*, that was used to generate a suite of benchmark graphs for finding connected components. This program creates a binary graph file

consisting of a list of unique, undirected edges that conform to the options provided on the command line.

The program can create graphs with a specific number of components that conform to one of four component structures: star, chain, mesh or random. The structure defines the minimal connections between the nodes when forming the component. After the initial component is created, the rest of the edges for that component are added at random.

The total number of edges in an n node graph is defined by the vertex degree or graph density. If a vertex degree is specified, all vertices of the resulting graph have the requested degree. Otherwise, the density parameter is used to define the total number of edges in the graph.

4.3. **Reading and Distributing Graphs.** All of our performance results were measured with graphs created by the *mkgraph* tool. The graphs were read into the MasPar and then distributed to the proper PEs in parallel. Although we measured the time taken to read and distribute the graphs, this time is *not* included in our performance results. Only the actual time to find the connected component is presented in this paper. It is interesting to note that the time to read and distribute the graphs ranged from 200 milliseconds to 2 seconds depending on the number of edges. While reading the file in parallel rather than sequentially significantly reduced the time it took to load the graph (by a few orders of magnitude), it still dominated the time it took to actually find the components for most graphs. We have determined that the limiting factor in loading the graph is the disk network itself and not the use of the router, which indicates that using the MasPar to find connected components must be part of a larger problem as opposed to a stand-alone program.

Each PE reads $\frac{m}{p}$ edges from the file where we have m edges and p processors. The edges are then distributed to the proper PE via parallel sends using the router. We studied several different decompositions and found a cyclic decomposition based on the vertex values provides the best performance on average.

Nodes, Edges	$\mathcal{A}_1$ time (ms)	$\mathcal{A}_2$ time (ms)	Comments
1000,9990	43.7	22.6	2% complete
2000,39980	125.4	47.2	2% complete
3000,89970	237.8	71.6	2% complete
4000,159960	387.7	103.8	2% complete
5000,249950	571.4	133.1	2% complete
6000,359940	766.1	164.1	2% complete
7000,489930	1014.2	197.6	2% complete
8000,639920	1294.4	335.1	2% complete

TABLE 1. **Random Graphs**: Other than density no other property is prescribed for this experiment. The times increase linearly with the size of the graph.

4.4. **Random Graphs.** Algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ perform well on random graphs, which are more dense than tertiary or grid graphs. We tested a variety of random

graphs and compare results from both algorithms. Algorithm $\mathcal{A}_1$ has clear performance advantages in dense graphs, but when the density is under 20% $\mathcal{A}_2$ is the faster.

Table 1 shows our results when we varied the number of nodes from 1000 to 8000 while maintaining a constant density of 2%. The times for both algorithms increase almost linearly as the number of nodes is increased.

Our next suite of graphs all have 4096 nodes, but their density varies from 1% to 10%. Table 2 shows the timing results for both $\mathcal{A}_1$ and $\mathcal{A}_2$ in finding the connected components of these graphs.

Nodes, Edges	$\mathcal{A}_1$ time (ms)	$\mathcal{A}_2$ time (ms)	Density
4096,83865	407.2	133.3	1% complete
4096,167731	408.1	145.7	2% complete
4096,251596	408.9	158.5	3% complete
4096,335462	407.7	173.1	4% complete
4096,419328	300.2	136.4	5% complete
4096,503193	299.4	145.2	6% complete
4096,587059	298.7	154.9	7% complete
4096,670924	298.1	165.9	8% complete
4096,754790	297.3	176.1	9% complete
4096,838656	296.4	186.7	10% complete

TABLE 2. **Varying Density of 4096 Node Random Graphs**: On sparse graphs, $\mathcal{A}_2$ always outperforms $\mathcal{A}_1$. The times increase nearly linearly with the density of the graph.

Table 3 amplifies the efficiency of $\mathcal{A}_1$ for very dense graphs. When the number of nodes is held constant, as in Table 2, and the density approaches 100%, $\mathcal{A}_1$ gets faster. The inverse is true for $\mathcal{A}_2$ whose time to find the connected components increases almost linearly with the increased number of edges. Table 3 also shows what happens when the density is kept constant at 50%, but the number of nodes is increased (as in Table 1 with 2% dense graphs).

Increasing Density				Increasing size,50% dense		
Nodes, Edges	Density	$\mathcal{A}_1$ ms	$\mathcal{A}_2$ ms	Nodes, Edges	$\mathcal{A}_1$ ms	$\mathcal{A}_2$ ms
4096,4193280	50%	268.4	547.8	1024,261888	35.5	55.7
4096,5031936	60%	264.6	631.7	2048,1048064	92.4	169.7
4096,5870592	70%	261.8	720.4	4096,4193280	268.4	547.8
4096,6709248	80%	260.2	799.7	8192,16775168	432.7	799.7

TABLE 3. **Dense Graphs**: The time for $\mathcal{A}_1$ goes down as we start approaching the completeness of the graph since the distance of any vertex to the lowest numbered vertex in its component decreases, which is what drives the complexity of algorithm $\mathcal{A}_1$.

4.5. **Tertiary Graphs.** Regular tertiary graphs are included in our benchmark suite of sparse graphs. Tertiary graphs satisfy the property that each vertex has exactly three neighbors. The neighbors are picked at random by the *mkgraph* program such that all vertices have degree 3. Thus, each tertiary graph has *1.5n* edges. As with all graphs generated by *mkgraph*, no self-loops or duplicate edges are allowed.

We employed a number of other definitions for tertiary graphs, but the performance of our algorithms was not significantly affected. In particular, we created AD3 [KLCY94] graphs. Each vertex in an AD3 graph selects between 0 and 3 neighbors so that one vertex may end up being directly connected to many different nodes. Such graphs tend to have more components [KLCY94]. We also generated graphs in which the degree of each vertex lies between 0 and 6 (uniformly distributed). The performance of $\mathcal{A}_1$ and $\mathcal{A}_2$ on these graphs mirrored the results shown for tertiary graphs.

Nodes	$\mathcal{A}_2$ time (ms)	Comments
50,000	2,095.6	75,000 Edges
100,000	4,160.0	150,000 Edges
150,000	6,208.6	225,000 Edges
200000	8,246.1	300,000 Edges
250000	10,288.8	375,000 Edges

TABLE 4. **Tertiary Graphs**: These are highly sparse graphs. The degree of a node is fixed to 3, but the neighbors are selected randomly. If we let the degree vary from 0 to 6, the performance of the algorithms does not change noticeably.

We only present performance results from $\mathcal{A}_2$ on sparse graphs. There are two reasons for this. First, the data structures required by $\mathcal{A}_1$ become too large to fit in memory when the number of vertices gets beyond 9,000. Second, $\mathcal{A}_1$ is designed for dense graphs and doesn't perform well on these sparse graphs.

The times shown in Table 4 are considerably higher than those in the previous tables. The time taken by $\mathcal{A}_2$ to find the connected components of the tertiary graphs ranges from approximately 2.01 seconds to 10.29 seconds when the number of nodes varies from 50,000 to 250,000.

4.6. **Grid Graphs.** The other class of sparse graphs in our suite are the grids. We generated two dimensional (2D) and three dimensional (3D) grid graphs. For each possible edge of the grid, the probability that it exists in the graph was varied. The probabilities were 0.4, 0.6, 0.2, and 0.4 for graphs of classes 2D40, 2D60, 3D20 and 3D40 respectively.

Grids are highly sparse graphs. As Tables 5 and 6 indicate, $\mathcal{A}_2$ is not affected by the structure of these graphs as much as it is by their density. Most of these graphs have long chains in comparison with random or tertiary graphs. As the density of the graph increases, the average number of components and average length of a chain decreases. The impact of the longer chains is clearly reflected in the longer execution times of $\mathcal{A}_2$ in Tables 5 and 6.

Nodes	$\mathcal{A}_2$ 2D40 time(ms)	Edges	$\mathcal{A}_2$ 2D60 time(ms)	Edges
65536	4,218.6	52496	4,426.1	78517
262144	14,460.2	208893	17,557.2	313593
300000	17,916.7	239239	16,018.3	359047
400000	25,325.1	319397	27,274.1	479365

TABLE 5. **2D Grids**: Highly sparse two dimensional grids. The 2D40 and 2D60 graphs were constructed such that the probability of a grid edge's existence in the graph is 40% and 60% respectively.

	$\mathcal{A}_2$ 3D40		$\mathcal{A}_2$ 3D60	
65536	2,507.8	39174	2,807.2	78403
262144	12,110.6	156825	11,080.5	313934
300000	15,547.3	179656	14,865.2	359321
400000	20,432.3	239755	25,290.9	479458

TABLE 6. **3D Grids**: 3D20 and 3D40 are highly sparse three dimensional grids. The corresponding probability of the existence of a grid edge in the graph is 20% and 40% respectively.

5. Slower Algorithms

While working with early versions of $\mathcal{A}_1$ and $\mathcal{A}_2$ we evaluated the possibility of performance gains using random mating techniques from the RM algorithm of [Ble90], a variation of which was presented as $cc\_RM2()$ in [Gre93]. Table 7 shows timing results of the NESL program $cc\_RM2()$ compared with the MPL implementations[1] of Algorithm 5.2 of [JÓ92] and algorithm $\mathcal{A}_2$. We acknowledge that the comparison is not entirely fair since MPL programs are in general faster than NESL versions.

Greiner claims the RM algorithm has $O(\log n)$ time complexity and $O(m \log n)$ work complexity in the worst case. However, the RM algorithm relies on CRCW capabilities, and the MasPar doesn't provide such support in hardware. CRCW can be simulated using library routines as was done in [HRD92], but then the *constant* communication costs assumed in the PRAM analysis isn't constant in the implementation. Moreover, the NESL implementation of the RM algorithm uses calls to the router for the *mating*, which can take up to 100 times longer than a mesh oriented communication mechanism. Our attempts to remove calls to the router led to algorithms similar to $\mathcal{A}_2$, but less efficient. Next we attempted to use random mating techniques at selected points in the algorithm. However, the performance cost of simulating the CRCW requirements of RM outweighed the potential benefits. We have concluded that random mating will not improve the performance of $\mathcal{A}_1$ or $\mathcal{A}_2$. Algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ perform well on the MasPar

[1] All algorithms, except the NESL $cc\_RM2()$ program, were implemented using the language MPL.

because great care has been taken to eliminate calls to the router and they don't require CRCW capabilities.

Nodes, Edges	NESL RM seconds	Jájá's 5.2 seconds	$\mathcal{A}_2$ seconds	Density
4096,419328	229.380	13.72	0.1364	5% complete
4096,167731	52.650	14.23	0.1457	2% complete
4096,4096	3.73	4.1	0.12	0.05% complete
8192,81920	15.46	4.7	0.54	0.24% complete
16384,163840	32.34	68.0	16.97	0.12% complete
409600,409600	40.95	273.77	42.37	0.00005% complete

TABLE 7. **Random Graphs**: Comparisons of a NESL implementation of Random Mating to MPL implementations of sparse graph connected component algorithms.

We also implemented a simple sequential algorithm based on depth first search of the graph. This was implemented on an HP 9000-712/80, an 80MIPS machine. We found that for small[2] 2% random graphs the sequential implementation beat our MasPar algorithms. As the graph grows beyond 4K nodes, the sequential implementation starts getting slower. We observed that the sequential implementation exhibited about 2-7 times improvement in performance for sparse random graphs of up to 8K nodes over the MasPar routines. However, the sequential machine did not have sufficient memory to store larger graphs. Therefore, performance dropped considerably when the graphs were paged in and out of memory. The performance of the sequential machine was limited by memory even though it had a significantly faster processor than the type of processors used in the MasPar.

6. Faster Algorithms

The success of a parallel algorithm lies in how well it keeps the processors busy and how well the communication pattern can be mapped onto the structure of the machine. We used fast communication mechanisms, but load balancing remains an issue.

We found that cyclic (cut-and-stack) decomposition provides better all around performance than hierarchical (block) decomposition for the sparse graphs handled by $\mathcal{A}_2$, but both of these simple virtualization techniques can result in load imbalances among the processors. While first developing our graph creation tool (see Section 4.2) to build the benchmark graphs, we employed a bad random number generator and our *random graphs* were not very random. The resulting graphs produced a load imbalance that was worse than 50:1 (and we may find such graphs in practice). This type of load imbalance creates more work than our $O(m \log n)$ goal, but the overall execution times were still not far from the numbers reported in this paper — they were about 10–20% slower. With a more uniform random number generator in place, we see processor load is balanced quite well with a load imbalance on the order of 1.3:1 for 2% complete 8,000 node graphs. However, we have seen an imbalance as high as 5.4:1 for some graphs.

[2] 4000 vertices

We have identified three distinct methods of improving the performance of our connected components algorithms. The next three sections outline these ideas.

6.1. **Load Balancing.** Virtualizations based on random sampling may improve the load balance between processors when an imbalance exists. To come close to the best known PRAM complexity of $O(m \log n)$, we need to get the work distributed evenly. One way to achieve this goal would be to execute a fast graph *pre-conditioner* that determines load balance and sparsity of the graph before it is distributed. Our plan is to have each PE randomly select edges from the block it reads and execute the pre-conditioner. This sampling is sorted and then segmented scans count the degree of each node to determine the connectivity.

Load balancing can also be performed by creating new edges in under-loaded PEs and then mapping some of the edges from the overloaded PEs to these new edges. One can think of it as splitting graph G that contains a node u into two graphs G_1 and G_2 that contain nodes u_1 and u_2 respectively. For each edge $(u, v) \in G$ there exists either $(u_1, v) \in G_1$ or $(u_2, v) \in G_2$. In addition, the edge (u_1, u_2) is created. The resulting graph $G' = G_1 \cup G_2$ has the same connected components as G.

6.2. **Graph Contraction.** Graph contraction, which substitutes a smaller, simpler graph problem for the original graph, is another promising method for reducing the work complexity of our algorithm. In practice, after a couple of iterations, we get several trees that form rooted stars. Some of these rooted stars are actually stagnant trees while others are connected components that have already collapsed to a star. Building a new graph with single nodes representing the rooted stars and keeping only the vertices and edges necessary to continue the algorithm has the potential to greatly reduce the work complexity. However, graph contraction introduces its own set of problems. It can cause load imbalance, especially with vertices that have dense connections in the new graph. Another problem in implementing graph contraction is recognizing the duplicate edges that are no longer needed in the smaller graph. Duplicate edges arise when one node in the contracted graph represents a stagnant rooted star that has edges connecting multiple leaf vertices to a node in a chain of another tree. While both of these problems have been solved before, the solution employed must use only row or column communication primitives if we are to reduce the execution time of the algorithm as well as its work complexity.

6.3. **Type of Graph.** There exist a variety of algorithms that perform differently on different classes of graphs. If we could identify the best possible algorithm for a given class of graphs and given a graph, identify its class efficiently, we may be able to find components of any given graph efficiently.

In this enhancement, we choose one of the connected components algorithms based on the graph density in the sampled data. The hard part is finding the correct density thresholds for the sampled data. We hope to spend a small amount of time up front to select the proper algorithm and to create a balanced work load. For a large class of graphs, this overhead will be more than offset by the efficiency gained by selecting the correct algorithm and having the work evenly distributed.

We feel that this type of sophisticated virtualization is the key to finding connected components quickly. It provides the opportunity to select the best algorithm for the graph density and to distribute the work evenly.

7. Conclusion

We have shown how to implement pointer jumping and related CRCW PRAM operations using simple row and column operations to minimize communication time and thus speed up the total execution. We have encouraging performance results, and have shown the feasibility of efficient implementations on modest sized machines.

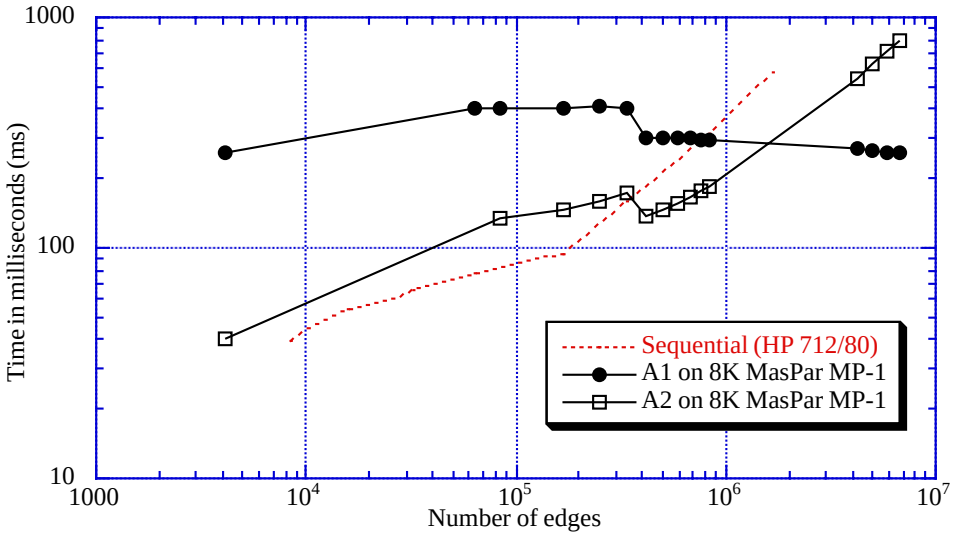

Figure 3. Performance Analysis: This graph plots the performance of algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ against a depth-first sequential algorithm for a 4096 vertex graph as we increase the graph density (i.e. the number of edges). Algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ were executed on a 8196 processor MasPar MP-1. The sequential algorithm was executed on a HP 712/80 workstation.

The graph in Figure 3 shows the effectiveness of our algorithms. The number of edges were varied while keeping the number of nodes constant at 4096. The performance of algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ have been compared with a sequential implementation on an HP 712/80 workstation. The size was kept small to accommodate the sequential implementation, but the general characteristics of $\mathcal{A}_1$ and $\mathcal{A}_2$ is reflected in this graph. (For larger graphs, the sequential implementation becomes less competitive at lower densities.) The time of $\mathcal{A}_1$, which uses the adjacency matrix explicitly, remains relatively constant, while the time of $\mathcal{A}_2$ increases with the number of edges. The shape of the graph for $\mathcal{A}_2$ tracks the sequential implementation much more closely since it uses an adjacency list like representation.

Not surprisingly, the key to getting good execution times for sparse graphs is the careful implementation of the row and column operations on the sparse adjacency matrix. We believe we can get results on highly sparse graphs that rival our random graph times by additional efforts in this area. With more data structure changes, we think we can increase the size of the graphs that we will be able to process and reduce the total time. We also believe that such changes will make it possible to do work at each step strictly proportional to the largest number of edges present on

any processor. To this end, we believe there is a potential benefit in using sampling techniques to reduce the work balance problem to a manageable task.

Further work in this area includes the analysis and implementation of the row and column operations on other parallel machines to examine their performance relative to CRCW operations. If the results of that effort look promising, there are many other CREW and CRCW PRAM graph algorithms that employ pointer jumping and similar operations that could be implemented using the techniques we have described to make them more practical on current parallel machines.

References

[AS87] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for Ultracomputer and PRAM. *IEEE Transactions on Computers*, 36(10):1258–1263, 1987.

[Ble90] G. Blelloch. Unpublished CVL Code, 1990.

[CLC82] F. Chin, J. Lam, and I. Chen. Efficient parallel algorithms for some graph problems. *Communications of the ACM*, 25(9):659–665, 1982.

[CV91] R. Cole and U. Vishkin. Approximate parallel scheduling. Part II: Application to optimal parallel graph algorithms in logarithmic time. *Information and Computation*, 92(1):1–47, 1991.

[Gre93] J. Greiner. A comparison of data-parallel algorithms for connected components. Technical Report CMU-CS-93-191, CMU, 1993.

[HCS79] D. Hirschberg, A. Chandra, and D. Saraswate. Computing connected components on parallel computers. *Communications of the ACM*, 22(8):461–464, 1979.

[Hir76] D. Hirschberg. Parallel algorithms for the transitive closure and the connected component problems. In *Eighth Annual ACM Symposium on theory of Computing*, pages 55–57, Hershey, Pennsylvania, 1976.

[HRD92] T. Hsu, V. Ramachandran, and N. Dean. Implementation of parallel graph algorithms on the MasPar. Technical Report TR-92-38, University of Texas at Austin, 1992.

[HRD94] T. Hsu, V. Ramachandran, and N. Dean. Parallel implementation of algorithms for finding connected components. In *DIMACS implementation challenge*, 1994.

[HW90] Y. Han and A. Wagner. An efficient and fast parallel-connected component algorithm. *JACM*, 37(3):626–642, 1990.

[JÝ92] J. Jájá. *An Introduction to Parallel Algorithms.* Addison Wesley, NewYork, 1992.

[KLCY94] A. Krishnamurthy, S. Lumetta, D. Culler, and K. Yelick. Connected components on distributed memory machines. In *DIMACS implementation challenge*, 1994.

[KRS86] C. Kruskal, L. Rudolph, and M. Snir. Efficient parallel algoritms for graph problems. In *1986 International Conference on Parallel Processing*, pages 278–284, St. Charles, Illinois, 1986.

[SV82] Y. Shiloach and U. Vishkin. An O($\log n$) parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982.

[Vis84] U. Vishkin. An optimal parallel connectivity algorithm. *Discrete Applied Mathematics*, 9(2):197–207, 1984.

[Wyl79] J. Wyllie. *The Complexity of Parallel Computation.* PhD thesis, Cornell University, Department of Computer Science, Ithaca, NewYork, 1979.

Department of Computer Science, University of North Carolina, Chapel Hill NC 27599-3175, USA,

E-mail address: goddard@cs.unc.edu