DIMACS Series in Discrete Mathematics
and Theoretical Computer Science
Volume 00, 19xx

Connected Components on Distributed Memory Machines

Arvind Krishnamurthy, Steven S. Lumetta, David E. Culler,
and Katherine Yelick

Abstract. The efforts of the theory community to develop efficient PRAM algorithms often receive little attention from application programmers. Although there are PRAM algorithm implementations that perform reasonably on shared memory machines, they often perform poorly on distributed memory machines, where the cost of remote memory accesses is relatively high. We present a hybrid approach to solving the connected components problem, whereby a PRAM algorithm is merged with a sequential algorithm and then optimized to create an efficient distributed memory implementation. The sequential algorithm handles local work on each processor, and the PRAM algorithm handles interactions between processors.

Our hybrid algorithm uses the Shiloach-Vishkin CRCW PRAM algorithm on a partition of the graph distributed over the processors and sequential breadth-first search within each local subgraph. The implementation uses the Split-C language developed at Berkeley, which provides a global address space and allows us to easily manipulate the distributed graph data structure. We present our first version, then provide a detailed account of the optimizations used to create the final version. For graphs from real-world problems, we obtain speedups on the order of 20 on a 32-processor CM-5 and 238 on a 512-processor CM-5.

1. Introduction

Although the theory community has studied the asymptotic running times of numerous PRAM algorithms, comparatively few of these algorithms are used in practice. The implementations that do exist generally appear on small shared memory platforms such as the Cray C90 or on SIMD machines such as the CM-2 or Maspar, where messages between processors require only a single, very long cycle. Large parallel machines, however, typically have a distributed memory model:

1991 *Mathematics Subject Classification.* Primary 68Q22, 68R10; Secondary 68P05.

This material is based in part upon work supported by a National Science Foundation Graduate Research Fellowship, by the National Science Foundation Infrastructure Grant numbers CDA-8722788 and CDA-9401156, by the Lawrence Livermore National Laboratory Grant LLL-B28 3537, by the National Science Foundation award number CCR-9210260, by the Advanced Research Projects Agency of the Department of Defense monitored by the Office of Naval Research under contract DABT63-92-C-0026, by the Department of Energy under grant number DE-FG03-94ER25206. The content of the information does not necessarily reflect the position or the policy of these organizations.

1052-1798/00 $1.00 + $.25 per page

off-the-shelf processors loosely coupled via a fast network (e.g., TMC CM-5, Meiko CS-2, Cray T3D, Intel Paragon, IBM SP-1). In this paper, we demonstrate the process of adapting a PRAM algorithm to execute efficiently on a distributed memory machine. We start with the PRAM algorithm for finding the connected components of a graph, and, through a gradual process of refinement, we develop an efficient hybrid parallel algorithm.

Labeling the connected components of a graph has a wide range of uses, including applications in computer vision and condensed matter physics. Grouping adjacent pixels of similar intensity to identify edges and planes, for example, helps to analyze images for object recognition. By creating a graph in which adjacent pixels of equal intensity are connected and then finding the connected components of the graph, we find the homogeneous regions of the image. Connected component labeling is used in Physics to implement clustering in Monte Carlo algorithms such as that of Swendsen and Wang [**13**], which simulates physical systems near critical temperatures by repeatedly grouping particles into clusters (connected components) and choosing a new state for each cluster.

The graphs used for these applications have underlying grid topologies in either two or three dimensions. Because of the underlying topology, the graphs decompose easily into smaller fragments with only a small fraction of edges crossing between fragments, allowing much of the work in finding connected components to be performed locally. For our results, we use the graphs typical of Physics problems. These graphs are generated randomly, using a fixed probability for the presence of each edge from an underlying lattice graph. For problems in vision and image recognition, the presence of an edge from the underlying grid is not independent of the presence of other edges.

Although we are primarily interested in graphs from actual problems, we also consider an artificial graph type. The graph, denoted AD3 for "average degree three," is generated by having each node pick zero to three other random nodes as neighbors. AD3 is a variant of the Tertiary graph used by Greiner [**7**] for benchmarking connected components algorithms.[1] Graphs corresponding to physical systems usually exhibit locality in their structure. However, the AD3 graph exhibits almost no locality, and could therefore be viewed as an extreme input to our algorithm.

Previous parallel implementations of connected components algorithms have focused primarily on shared-memory machines [**7**]. For distributed memory machines, a straightforward implementation of a PRAM algorithm is generally of little use because of the high cost of remote accesses and the frequency of such accesses in most PRAM algorithms. A more sophisticated approach employs a PRAM algorithm in conjunction with a standard sequential algorithm, using the latter to manage operations local to each processor and the former to manage the interaction between processors. This hybrid approach is illustrated in Figure 1 for the connected components problem. The two algorithms are merged, then the result is optimized.

In this paper, we present a hybrid algorithm for finding the connected components of a graph on a distributed memory machine. We implemented and optimized the algorithm on a CM-5 using the Split-C language developed at Berkeley [**4**].

[1] Each node in an instance of Greiner's Tertiary graph randomly selects three other nodes as neighbors, resulting in an average degree of six and a graph that has only one connected component with high probability.

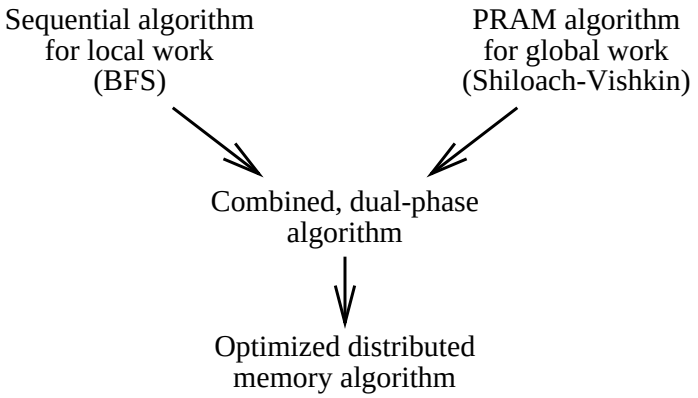

FIGURE 1. Hybrid algorithm strategy. We combine a PRAM algorithm with a sequential algorithm and optimize the result to create an efficient algorithm for distributed memory machines.

We discuss the PRAM algorithm in Section 2 and give details of the first hybrid implementation in Section 3. In Section 4, we describe the sequence of optimizations through which we developed the final version of our algorithm. Section 5 discusses the graphs used and our methodology for measuring performance. In Section 6, we present our results.[2] Section 7 compares the results with other implementations of the algorithm, and Section 8 concludes.

2. The PRAM Algorithm

Sequential solutions for identifying the connected components of a graph are generally based on variants of depth-first search, breadth-first search, or union-find. The solutions have running times linear in the number of edges and vertices in the graph and are easy to implement. Many efficient parallel solutions [**2, 6, 12**] have been devised, but these solutions are often complex and difficult to implement. Our implementation is a hybrid of a sequential search on the subgraph local to each processor and a variant of the Shiloach-Vishkin PRAM algorithm [**12**] on the global collection of subgraphs. In this section, we briefly describe the key components of the PRAM algorithm.

In the following discussion, we denote the vertex set of the input graph by V and the edge set by E. Each vertex has an associated $Value$ attribute that is a unique number at the beginning of the algorithm. When the computation terminates, all vertices within the same connected component share the same value. We use the notation (u, v) to denote an edge between the vertices u and v.

Given a graph with n vertices and m edges, the Shiloach-Vishkin algorithm requires *O(log n)* parallel steps and a total of *O(m log n)* work. The algorithm repeatedly groups vertices that have edges between them using two basic operations: *pointer doubling* and *hooking*. The algorithm maintains a *forest* of trees, and makes progress either by decreasing the number of trees in the forest or by decreasing the height of the trees. The algorithm terminates when no two trees in the forest share an edge and all trees in the forest are of height one.

[2] A separate paper [9] presents more detailed results for the algorithm on several different platforms and demonstrates the best connected components performance seen to date.

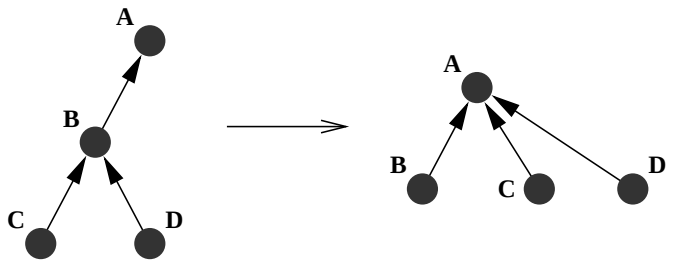

FIGURE 2. Pointer doubling operation. The parent of each vertex is replaced with the vertex' grandparent. The root of a tree is assumed to be its own parent.

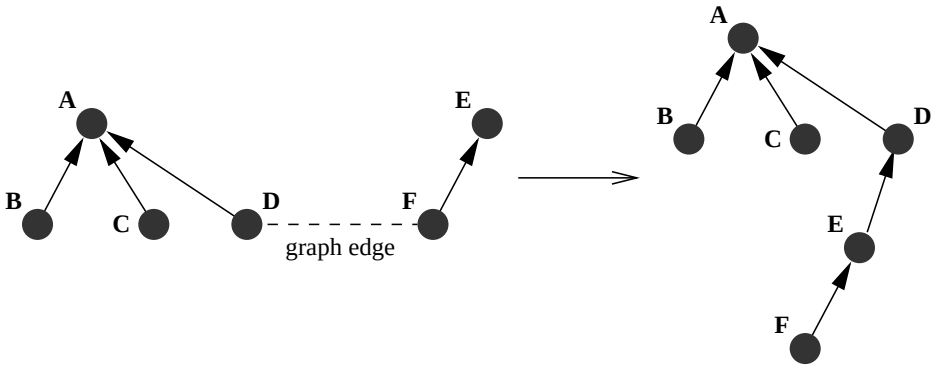

FIGURE 3. Hooking operation. A graph edge between two trees in the forest is replaced with a parent link, merging the two trees into a single tree.

The *pointer doubling* operation replaces the parent of each vertex with the vertex' grandparent, as shown in Figure 2. This operation decreases the distance from the root of the tree to the leaves, and terminates when the tree becomes a *star*, which is a tree of height 1. During the pointer doubling operation, the algorithm also propagates the value of the new parent to the child. By making the parent of the root of a tree the root itself, we simplify the operation to the form:

$$Parent(v) \leftarrow Parent(Parent(v))$$
$$Value(v) \leftarrow Value(Parent(v))$$

The *hooking operation* hooks a star in the forest to another tree in the forest if the star contains a vertex adjacent to a vertex in the target tree, as shown in Figure 3. The operation comes in two flavors: conditional and unconditional. A conditional hooking operation is permitted only when the $Value$ attribute of the first vertex[3] is less than that of the adjacent vertex. An unconditional hooking operation links the two trees irrespective of their values.

In order to guarantee termination, the algorithm must ensure that the parent relationship remains acyclic. The conditional hooking operation prevents the

[3]Note that the $Value$ of every vertex in a star is the same, as the $Value$ propagates from parent to child during pointer doubling.

formation of cycles by requiring that the $Value$ attribute monotonically increases from the leaves to the root of a tree. The same is not true of the unconditional hooking operation, however. The algorithm prevents the creation of cycles by first applying the conditional hooking operation, and then applying the unconditional hooking operation only to those stars that were not hooked in the conditional hooking phase. This scheme prevents two stars from linking to one another since at least one of the stars has had an opportunity to link to the other star during the conditional hooking phase. Unconditional hooking is necessary to obtain $log(n)$ bound on the running time, but is not necessary for correctness [**12**].

The algorithm follows:

1. For each vertex u, set
   Parent(u) $\leftarrow$ u
2. Repeat until no change occurs in an iteration:
   a. For each vertex u such that u is part of a star, pick v such that $(u, v) \in E$ and $Value(u) < Value(v)$ and set
   $Parent(Parent(u)) \leftarrow v$.
   b. For each vertex u such that u is part of a star that neither hooked to another vertex nor had another vertex hooked to it, pick v such that $(u, v) \in E$ and set
   $Parent(Parent(u)) \leftarrow v$.
   c. For each vertex u, set
   $Parent(u) \leftarrow Parent(Parent(u))$ and
   $Value(u) \leftarrow Value(Parent(u))$

Given one processor for each vertex and each edge in the graph, the loop requires $O(log\ n)$ iterations to terminate. The vertex processors perform during steps 1 and 2c, while the edge processors perform during steps 2a and 2b. The processors execute in a lock-step manner and must be able to read and write a single memory location concurrently for each step to execute in unit time. Steps 2a and 2b, for example, require the concurrent write ability, since multiple children of a vertex may attempt to change the parent. Note, however, that the algorithm makes no assumptions about the policy for disambiguating writes to the same location.

3. Implementation

In this section, we describe our initial implementation of the connected components algorithm. We start with a hybrid algorithm, which composes the local sequential breadth-first search with a global PRAM-based algorithm. Although this implementation proved to be inefficient, the description introduces the general style of the program and facilitates understanding of the optimizations discussed later.

The natural implementation of many algorithms on distributed memory machines involves a combination of local and global phases. During the local phases, the algorithm deals only with data that reside in the processor's local memory. In the global phases, the algorithm must address the issues of efficient remote data access and synchronization between processors. The global phases are hence more difficult to program.

Fortunately, we can make use of the Split-C language [**4**] to simplify the task. Split-C provides the abstraction of a global address space on a distributed memory

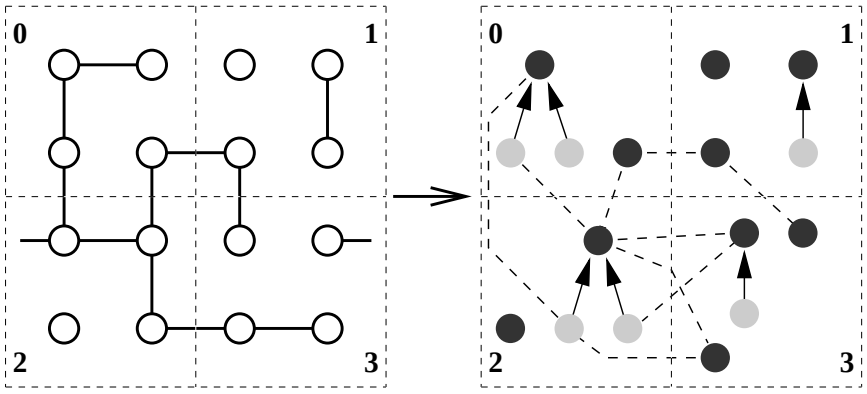

FIGURE 4. Local phase. In this phase, the algorithm processes all local edges to find local connected components, then passes the reduced graph into the global phase.

machine.[4] Any processor can access any location in the global address space using *global pointers*, and each processor owns a specific region of the global space, its local region. A global pointer is used just like a local pointer, but can reference the entire global address space, while standard pointers reference only the portion local to the accessing processor. The notion of a global pointer allows us to represent a graph whose vertices are spread across processors and whose edges are represented using global pointers. Another useful aspect of global pointers is the ability to determine the processor that owns the object pointed to by a global pointer without actually dereferencing the pointer. In our implementation, we use this ability in the local phase to explore only those edges that point to local vertices. The distinction between local and global objects provides a clear cost model for introducing optimizations that we examine later.

Having briefly discussed the language used to code our implementation, we now introduce the algorithm:

1. **Local Phase.** Perform purely local computations to decrease the size of the graph processed during the global phase. Figure 4 illustrates the effect of the local phase.
   a. *Search Step.* On each processor, find local connected components among local nodes and edges using Breadth First Search (BFS). Ignore remote edges.
   b. *Star Formation Step.* Assign a unique value to each node. Choose a representative node for each local connected component. Move all remote edges from nodes in the component to the representative and collapse the component into a star. The representative node becomes the root of the star, and the value of the representative node becomes the value of the star.
2. **Global Phase.** Beginning with a list of components on each processor, all of which are stars and are marked with unique values, apply a modified Shiloach-Vishkin algorithm. Iterate over the following steps until done:

[4] Implementations of the language exist on a variety of machines including the IBM SP-2, the Intel Paragon, the Cray T3D, and the Meiko CS-2 [**1, 8, 10, 11**].

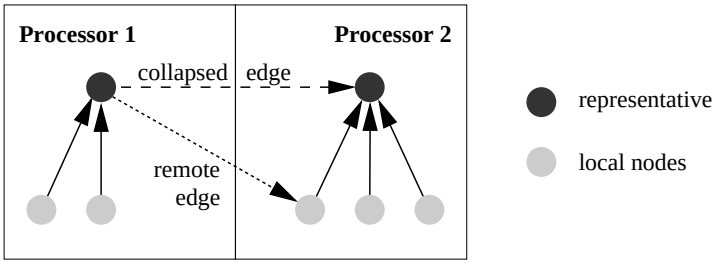

FIGURE 5. Collapsing remote edges. By collapsing remote edges before entering the global phase, we reduce the amount of work required for each iteration of that phase.

a. *Termination Check.* Move star components with no remaining remote edges to a finished component list. If all components on all processors are finished, quit.
b. *Conditional Hooking Step.* Attach star components to other components if the value of the other component is larger. Remove newly attached components from the component list.
c. *Pointer Doubling Step.* Double parent pointers one or more times for each node and update the node's value from the new parent.
d. *Star Marking Step.* Determine whether each component is a star: first mark all components as stars, then mark the grandparent of each node as a non-star if it is distinct from the parent of the node.
e. *Edge-List Concatenation Step.* For each node, pass remote edges to the parent of the node.
f. *Self-Loop Removal Step.* For each star, remove edges that point to nodes with the same value (nodes within the star).

The initial implementation does not include an unconditional hooking phase. We introduce this phase and study its effect in the next section.

4. Optimization

In this section, we describe the sequence of optimizations that we used to improve the performance of our implementation on distributed memory machines. On a CM-5, the optimized code runs roughly twenty times faster than does the basic version described in the last section. The optimizations make use of three simple concepts in parallel optimization: reducing the amount of computation, reducing the number of remote references, and balancing the workload between processors.

4.1. Collapsing remote edges. The local phase leaves all local components in star form. We first consider the role of the leaf nodes of these stars, which we call the *local nodes*. As the local nodes make up the bulk of the graph in most cases, we want to eliminate any reference to them within the global phase. Although we have chosen representative nodes (which are the roots of the stars) during the local phase of the algorithm and have moved one end of each edge to the representative nodes, the other end of each edge remains unchanged, and often refers to a local node. To avoid creating a cycle in the graph, we must keep the unique component values for the local nodes consistent in each iteration of the global phase. By extending the

Graph	Nodes	Avg. Nodes in Star
2D40	10,000	4.2
2D60	10,000	27
3D20	8,000	2.3
3D40	8,000	14
AD3	10,000	1.05

TABLE 1. Average star size after the local phase. Graphs used in real-world problems form large stars; artificial graphs might not. All measurements used a 32-processor CM-5, and the "Nodes" column shows the number of graph nodes per processor.

algorithm slightly, we remove edge references to the local nodes and greatly reduce the amount of work done in the global phase.

The first extension involves collapsing the remote edges just before beginning the first iteration of the global phase. For each remote edge, we replace the remote node with the parent of the remote node, as shown in Figure 5. Since each local component has the form of a star after the local phase, the parent of any node is that node's representative. The extension adds the following step just after the local phase:[5]

1.c. *Remote Edge Collapse Step.* Replace each remote edge (u, v) with the collapsed edge $(u, Parent(v))$.

The local nodes can then be safely ignored during the global phase, and their component values can become inconsistent without affecting the correctness of the algorithm.

The second extension involves updating the unique component values of the local nodes after the global phase completes, bringing them back into consistency with the reduced graph. Since all representative nodes are updated in the global phase, we need merely copy the component value of each local node from its parent, an operation requiring no remote references as the parent of a local node is always local. We add an update phase after the global phase:

3. **Update Phase.** For each local node, update the value of the node from the value of its parent.

How these two extensions affect the execution time of the algorithm depends upon the balance between the computation and communication architectures and upon the structure of the graph. The first extension potentially adds an additional remote reference for each remote edge, but allows us to forgo updating the values of local nodes during the global phase. We must eventually update these values at least once, and do so in the second extension after the global phase completes. In a graph where the number of local nodes is small compared to the number of remote edges, or on a machine on which the cost of a remote reference is large compared to the cost of a cache miss, the changes described in this section can increase the

[5]The new step accesses remote data and should technically not be a part of the local phase, but the numbering used indicates both the appropriate insertion point for the step and fact that the step is not part of the global phase iteration.

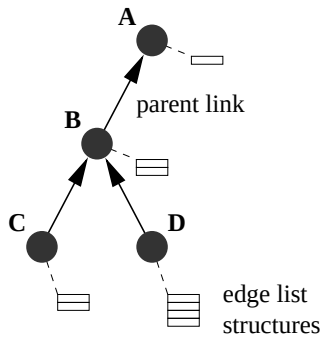

FIGURE 6. Postponing edge list concatenation. Race conditions in edge motion make efficient design difficult; waiting until the tree has collapsed into a star eliminates races and allows further optimization.

execution time. In our case, however, they greatly reduced that time. Table 1 shows the average star size for our graphs.

4.2. Postponing edge list concatenation. The initial implementation moved remote edges from leaf nodes to their parents after every pointer doubling, concatenating each leaf's list to that of its parent. Unless we are careful in designing the interactions for this step, races between the parent and children of a node make the code very unstable. To understand the problem, consider the tree shown in Figure 6. The edge list structures exist in the global address space, so that accessing or modifying an edge might involve a remote access. In one correct solution, each node maintains pointers to the first and last edge list structures in its list of edges. Before sending edges to its parent, a node saves copies of these two pointers and zeroes the originals in an atomic fashion, protecting the edge list from corruption by incoming edges. The node then sends both the start and end pointers to its parent. When the parent node receives these pointers from a child, it replaces its own first edge with the child's first edge and, if the parent already had edges, modifies the child's last edge to point to the parent's previous first edge. The latter operation requires a message to the processor on which the child's last edge list structure resides, but since the parent is the only node that has a pointer to this edge list structure, no races exist.

Unfortunately, the method outlined above requires too much information to take advantage of the short messages available on the CM-5, and using a slightly longer message adds a significant amount of overhead and complexity. The solution we chose is to delay the motion of edge lists. By postponing the edge list concatenation on a tree until pointer doubling has collapsed the tree into a star, we sidestep the difficulties of designing a correct and efficient method for this step. The cost is moderate—the root of each tree must handle all of the link messages instead of handling one or more rounds from each immediate child (a very small cost when the height of the trees is small)—and the change allows us to take advantage of a more significant optimization, as we discuss in the next section. We modify step 2e to affect only stars:

2.e. *Edge-List Concatenation Step.* For all leaf nodes of star components, pass remote edges to the star root.

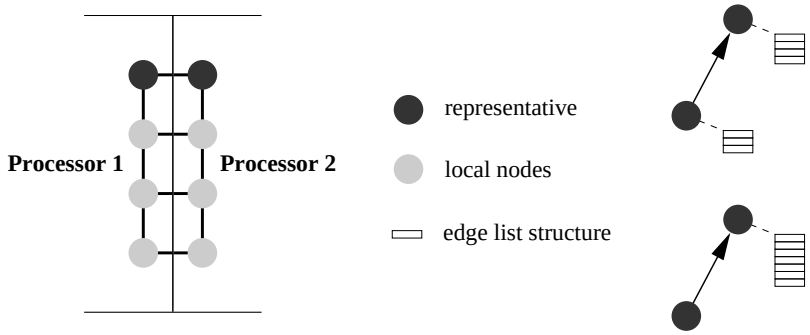

FIGURE 7. Removing duplicate edges before concatenation. Pushing duplicate edges to the root of a star forces the root to handle the destruction of all duplicates. Eliminating the duplicates first improves load balance.

4.3. Removing duplicate edges before concatenation. After the local phase completes, many components are left with *duplicate edges* in their edge lists. The graph segment depicted in Figure 7, for example, gives rise to a set of four duplicate edges in both local components. Detecting duplicate edges at this point requires sorting the edge lists and fails to catch duplicate edges created in the global phase (when two local components are collapsed into a single component). For this reason, the original algorithm checked for duplicate edges at the end of each global phase iteration, when they appear as edges in a star that point to other nodes within the star.

The problem with such an approach is that the root of the star must perform all of the remote references needed to detect duplicate edges. The upper righthand section of the figure shows the two representative nodes and their edge list structures after one node has been hooked to the other (eliminating one edge). In this figure, all of the edge list structures reside on the same processor as the associated node, and all of the edges point to remote nodes. The lower righthand section shows the nodes after edge list concatenation; in this case, three of the edge list structures are remote, and the complementary structures (the other four) point to remote nodes. In both cases, each edge list structure requires one remote access to determine its duplicate nature, but in the upper figure, the cost of these accesses is shared between two processors, while a single processor must perform all remote accesses in the lower figure. By checking for duplicate edges before concatenating the edge lists, we create a better load balance between the processors. We swap the two steps:

2.e. *Self-Loop Removal Step.* For each star, remove edges that point to nodes with the same value (nodes within the star).
2.f. *Edge-List Concatenation Step.* For all leaf nodes of star components, pass remote edges to the star root.

Since the root of a star must perform a remote reference for each edge when scanning the list for duplicates, the benefits of this optimization outweigh the small costs incurred by delaying concatenation in the earlier section, where the root of a star need make only one remote reference per edge list.

Graph	Nodes	Iteration	Components	Stagnant	Percentage
2D40	2,000,000	1	472,538	425	0.090%
2D60	2,000,000	1	70,141	157	0.22%
3D20	4,000,000	1	1,673,284	1,915	0.11%
3D20	4,000,000	2	1,627,463	15	0.00092%
3D40	4,000,000	1	251,734	823	0.33%
3D40	4,000,000	2	228,101	2	0.00088%
AD3	1,600,000	1	1,525,032	47,560	3.1%
AD3	1,600,000	2	252,240	6,624	2.6%
AD3	1,600,000	3	100,671	25	0.025%

TABLE 2. Usefulness of unconditional hooking. The stagnant components make up a small fraction of the total, but lead to additional iterations of the global phase. All measurements in the table were made on a 32-processor CM-5.

4.4. Unconditional hooking. Certain pathological graphs require unconditional hooking to prevent the possibility of requiring one iteration per node to find connected components. But the stagnation information needed for unconditional hooking requires extra work and extra remote references, and the unconditional hooking phase itself adds still more overhead. As we are not concerned with these pathological cases, we chose not to implement unconditional hooking in our initial implementation.

We found, however, that unconditional hooking serves a practical purpose, as demonstrated by the data in Table 2. For each graph type, the table shows the number of components left stagnant during each iteration of the global phase. Although the stagnant fraction is generally small, the number is large enough to increase the number of iterations required in the global phase. Adding unconditional hooking results in fewer iterations, and the time gained by eliminating iterations outweighs the overhead costs.

We modify the conditional hooking step and add the new step as follows:

2.b.1) *Conditional Hooking Step.* Mark all stars as stagnant. Attach star components to other components if the value of the other component is larger. Remove each newly attached component from the component list and remove the stagnant marker from the component to which it attached.

2.b.2) *Unconditional Hooking Step.* Attach stagnant star components to other components. Remove each newly attached component from the component list.

4.5. Asynchronous pointer doubling. The standard approach to pointer doubling, as shown in Figure 8, requires that each processor iterate over its local set of nodes and double the parent link for each vertex (replace the parent with the grandparent). The processors then synchronize and repeat the doubling procedure some number of times. The synchronization guarantees that each doubling decreases the height of all trees by a factor of two.

An alternative approach reverses the loop structure and eliminates the synchronization; each processor iterates over vertices and replaces the parent of each

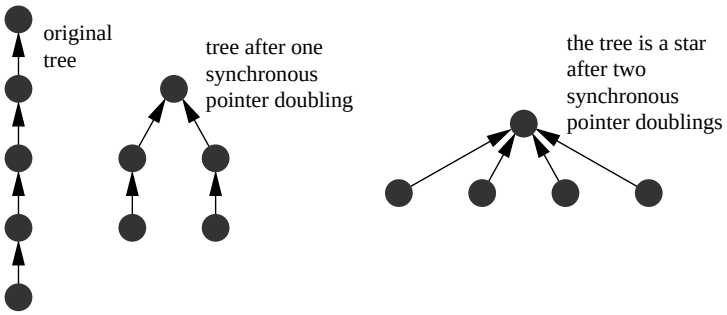

FIGURE 8. Asynchronous pointer doubling. Synchronized pointer doubling guarantees fully collapsed trees in time logarithmic in the height of the trees. Asynchronous doubling offers no such guarantee, but requires much less overhead.

vertex with the node found by following the parent link some number of times. This second approach has much less overhead than the first: not only does it lack multiple synchronizations between processors, but it requires only one loop through a processor's vertices. However, the approach does not guarantee that the depth of the tree decreases exponentially.

We implemented both schemes and compared the results. Surprisingly, execution times were very close. The extra overhead of synchronization and multiple list traversals compensated for the benefits of guaranteed exponential decrease. Keep in mind, however, that barrier synchronization is relatively inexpensive on the CM-5. On machines with more heavyweight barriers, asynchronous pointer doubling is more beneficial.

4.6. Maximal pointer doubling. The last optimization we discuss involves tuning the number of doublings performed in each pointer doubling phase. We studied the effect of the number on execution time and found the the optimal number varied from three to seven, depending on the structure of the graph. We next tried maximal pointer doubling, in which pointer doubling continues until every tree is reduced to a star. Although execution times were slightly worse, maximal pointer doubling enabled numerous other optimizations. Since every tree entering an iteration of the global phase was a star, we removed the star-marking phase entirely and eliminated all conditionals that checked a tree's star property. The resulting program ran faster than the one that performs an optimal number of pointer doublings. The algorithm outlined below reflects the numerous minor changes made with maximal pointer doubling.

4.7. Final Algorithm. The optimized algorithm follows:

1. **Local Phase.** Perform purely local computations to decrease the size of the graph processed during the global phase.
   a. *Search Step.* Each processor finds local connected components among its nodes using Breadth First Search (BFS). The search ignores remote edges.
   b. *Star Formation Step.* Assign a unique value to each node. Choose a representative node for each local connected component. Move all remote edges from nodes in the component to the representative and

collapse the component into a star. The representative node becomes the root of the star, and the value of the representative node becomes the value of the star. Make a list of star roots for each processor.

c. *Remote Edge Collapse Step.* Replace each remote edge (u, v) with the collapsed edge $(u, Parent(v))$.

2. **Global Phase.** Beginning with a list of components on each processor, all of which are stars and are marked with unique values, apply a modified Shiloach-Vishkin algorithm. During this phase, ignore any nodes not on the local star root list. Iterate over the following steps until done:
   a. *Termination Check.* Move components with no remaining remote edges to a finished component list. If no components need still be processed on any processor, quit.
   b. *Hooking Steps.* Merge components into larger components.
      1) *Conditional Hooking Step.* Mark all components as stagnant. Attach components to other components if the value of the other component is larger. Remove each newly attached component from the component list and remove the stagnant marker from the component to which it attached.
      2) *Unconditional Hooking Step.* Attach stagnant components to other components. Remove each newly attached component from the component list.
   c. *Pointer Doubling Step.* Double parent pointers for each node until the parent and the grandparent are the same; that is, collapse all components into stars. Update the node's value from the new parent.
   d. *Self-Loop Removal Step.* For each component, remove edges that point to nodes with the same value (nodes within the component).
   e. *Edge-List Concatenation Step.* For all leaf nodes of components, pass remote edges to the component root.
3. **Update Phase.** For each node with a local parent, update the value of the node from the value of its parent.

5. Graphs and Methodology

We begin this section with a description of the graphs used to measure the performance of our optimized implementation. We then discuss our measurement methodology.

5.1. Graph construction. As our results depend fairly heavily upon the types of graphs studied, we first describe those graphs. We used five separate types of graphs; four are drawn directly from the work of Greiner [7], and the fifth is a modified form of another graph used in that work.

The first two graphs are built on a two-dimensional toroidal mesh. Each edge in the mesh is present with some fixed probability, either 40% or 60% in our measurements. Since one expects a graph with average degree below two to be fairly disconnected and a graph with average degree above two to be fairly connected, these two percentages outline the boundary region for the two dimensional grid. We call these graphs 2D40 and 2D60, following the notation given by Greiner. We divide the underlying rectangular mesh into P square chunks, where P is the number of processors.

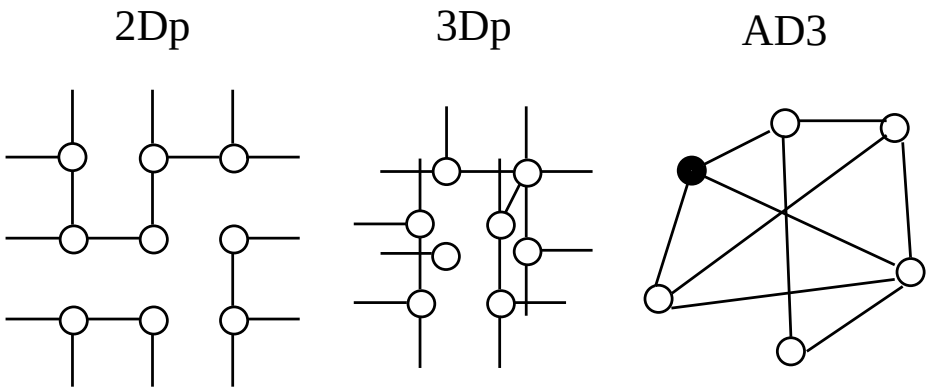

FIGURE 9. Graph types. The left and middle graphs correspond to real problems; p is the probability of presence for each edge on the underlying grid. The right graph is artificial; each node selects zero to three other nodes as neighbors.

The second two graphs are built in the same fashion on a three dimensional toroidal mesh. The boundary between fairly connected and fairly disconnected graphs falls near 33% in the three dimensional case, so our measurements use edge presence probabilities of 20% and 40%. We call these graphs 3D20 and 3D40. The underlying mesh again provides the best method of partitioning the graph among processors.

The last type of graph, AD3, is a randomly generated graph of average degree three. Each node randomly selects zero to three other nodes as neighbors, resulting in a graph with average degree three (although the degree of a particular node may vary from zero to $3n + 3$). The lack of an underlying coordinate space and the nature of the generation method for AD3 lead to extremely poor locality.

5.2. Methodology. Except where noted, all measurements were averaged over several runs of the algorithm using distinct random seeds. Variation in execution time between multiple runs arises from three sources. In decreasing order of significance, these sources are the random nature of the graph, the non-deterministic behavior of the algorithm, and irreproducible timing fluctuations on the CM-5.

An unforeseen side effect of our random number generation led to a graph independent of the random seed on one processor, introducing some amount of systematic error into our measurements. Although the error is practically irrelevant for large numbers of processors, the advantage of averaging is nullified in the single processor data. We blame this problem in several cases for bumps in our data, where the runs with many processors found graphs from both the high and low end of the execution time distribution, but the single processor runs found only a single graph.

Table 3 shows variation in execution time for large samples of all graph types running on 32 processors. Twenty seeds were chosen at random and fed into the algorithm for each graph type. The table shows the size of the graphs, the average time, and the standard deviation in seconds and as a percentage of the average. The variations are larger for more strongly connected graphs and tend to rise around the boundary regions between mostly connected and mostly disconnected graphs.

Graph	Nodes	Edges	Avg. Time (sec)	Std. Dev. (sec)	SD/Avg.
2D40	2,000,000	1,600,000	1.13	0.064	5.69%
2D60	2,000,000	2,400,000	1.63	0.12	7.59%
3D20	4,000,000	2,400,000	2.59	0.14	5.52%
3D40	4,000,000	4,800,000	4.83	0.97	20.1%
AD3	1,600,000	2,400,000	9.29	2.8	30.3%

TABLE 3. Variation in execution time. The random nature of the graph proved to be the most significant factor in execution time variance.

Size	Nodes	Time (sec)	Per Node (usec)
18x18x18	5,832	0.1451	24.88
19x19x19	6,859	0.1711	24.95
20x20x20	8,000	0.1960	24.50
21x21x21	9,261	0.2319	25.04
22x22x22	10,648	0.2559	24.03
23x23x23	12,167	0.2900	23.83
24x24x24	13,824	0.3308	23.93

TABLE 4. Variation in execution time per node as a function of graph size. Measurements of a 3D20 graph on a 32-processor CM-5 show only minor variations in cost per node, validating our scaling of measurements.

The largest variation occurs in the AD3 graphs, where the variation in number of references is amplified by the higher average cost of each reference.

In addition to averaging, some of the results are scaled linearly from graphs close to the same size. As the graph creation section of the program allowed only for square (cubic) sections on each processor for the 2D (3D) graphs, we were unable to obtain graphs with exactly the same number of nodes when doubling the number of processors. To justify our choice of linear scaling, we investigated the effect of graph size on execution time for various graphs. Since the number of nodes in the actual graphs measured are quite close to the desired number of nodes, we require only that the effect be reasonably approximated by a line. The data in Table 4 show the results for 3D20 graphs. The cost does not vary by more than 5% over the range shown, and demonstrates no clear trend.

6. Performance Measurements

In this section, we discuss our measurements of the optimized algorithm running on a 32-processor CM-5 and on a 512-processor CM-5 (we were able to obtain only one set of data on the latter). We first explore the efficiency of the parallel algorithm by comparing execution times for graphs of fixed size running on a variable number of processors. Next, we scale the graph size with the number of processors to demonstrate scalability for large graphs. Finally, we look at two performance factors in detail: convergence and load balance.

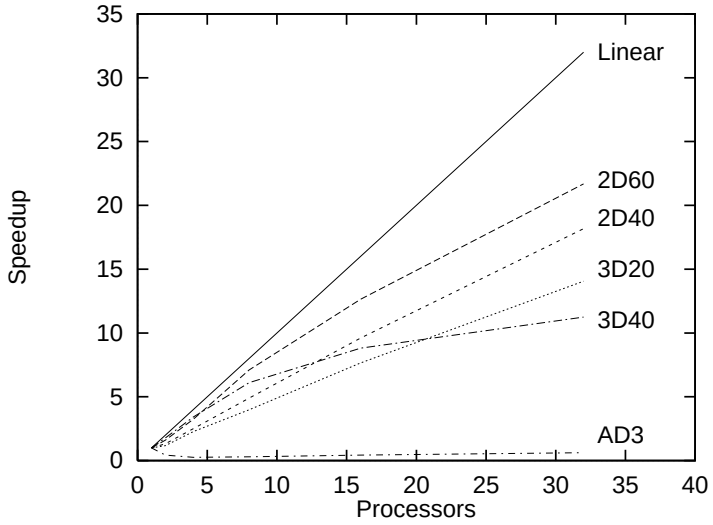

FIGURE 10. Speedup a problem size of 256K nodes.

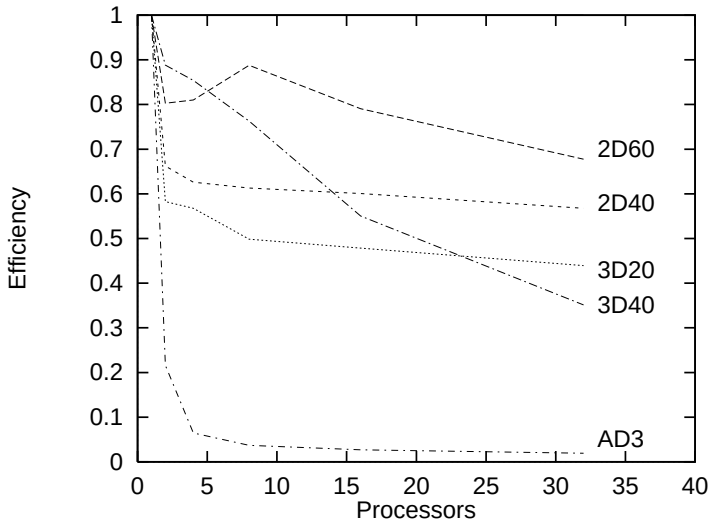

FIGURE 11. Efficiency for a problem size of 256K nodes.

6.1. Speedup. Speedup of the parallel version is measured by comparing to a sequential implementation that contains only step 1a of the algorithm. Although the algorithm requires no communication on a single processor, it takes a significant amount of time that is not relevant to the sequential execution time of the program.

Figure 10 shows the speedup for a fixed problem size (262,144 nodes) on between 1 and 32 processors of a CM-5. Ignoring AD3 for the moment, the speedups are roughly linear after discounting the overhead in moving from one processor to two. The exception is 3D40, for which the chunk owned by each processor has become small enough that the fraction of remote edges rises significantly and limits the speedup. AD3 never regains a speedup of 1, because the nonlocal structure of the

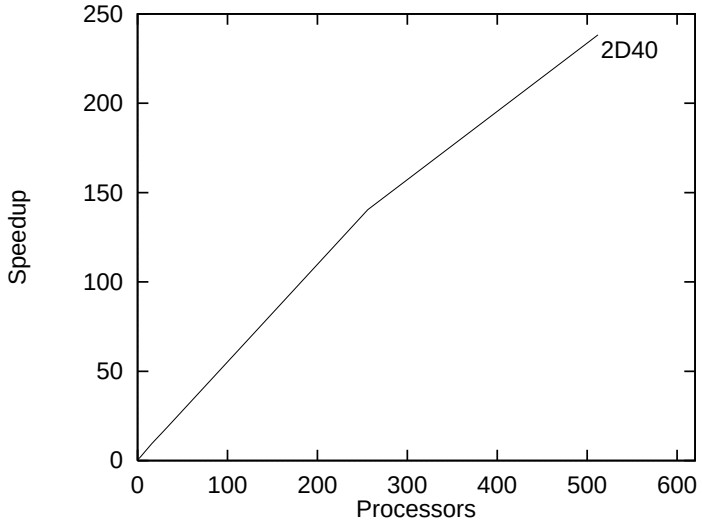

FIGURE 12. Speedup for a 2D40 graph up to 512 processors. The problem size is 256K nodes between 1 and 32 and 8M nodes between 32 and 512.

graph leads to unavoidable communication. Figure 11 shows the efficiency for the same data set. Again, we see that 2D40, 2D60, and 3D20 fall rapidly to a fairly level plateau, indicating good scalability on these problems.

Figure 12 extends the speedup for 2D40 graphs on machine sizes up to 512 processors using an 8 million node graph. Because this graph does not fit on a small number of processors, the speedups from 1 to 32 are computed using the smaller, 256 thousand node, graph and setting the speedup for 8 million nodes on 32 processors equal to the speedup for 256 thousand nodes on 32 processors.[6]

A second definition of speedup, *scaled speedup*, uses a problem size proportional to the number of processors. In Figure 13, we see the results for the graphs using this definition and varying numbers of nodes per processor (dependent upon graph type). They follow the same pattern as did the previous set, with slightly better values. Finally, in Figure 14, we see the scaled efficiency for the algorithm on all graph types. The plateaus in this case are flatter because the fraction of remote edges remains roughly constant across the graph, except between 1 and 2 processors.

6.2. Convergence and load imbalance. Most runs of the algorithm converged in about 2 or 3 iterations for the 2D and 3D graphs. AD3 graphs took a few more iterations, averaging about 3 or 4. In Figure 15, we see the number of remote edges remaining after each iteration normalized by the number of remote edges existing immediately after the local phase (iteration 0). The rate at which edges are removed depends on the degree of graph connectivity: the mostly unconnected graphs, 2D40 and 3D20, lose over 95% of their edges in the first iteration; the mostly connected graphs, 2D60 and 3D40, lose between 75% and 90% of their edges in the first iteration; and the most strongly connected graph, AD3, loses just

[6]Note that this method underestimates the speedup on large graphs—the larger chunks have relatively fewer remote edges and should therefore have better speedup than the smaller chunks.

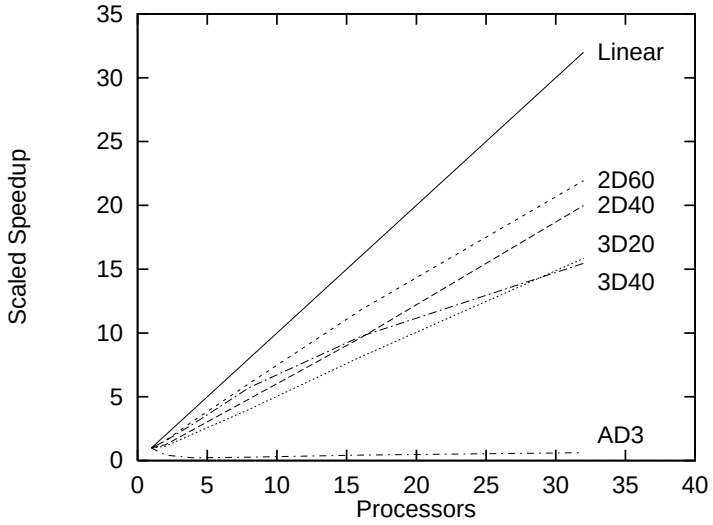

FIGURE 13. Scaled speedup. The problem size per processor is fixed: 62,500 nodes for 2D40 and 2D60, 125,000 for 3D20 and 3D40, and 50,000 for AD3.

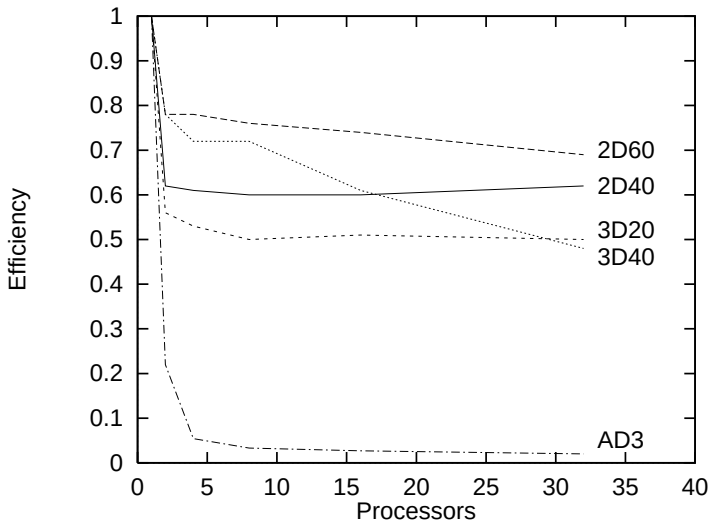

FIGURE 14. Scaled efficiency. The problem size per processor is fixed: 62,500 nodes for 2D40 and 2D60, 125,000 for 3D20 and 3D40, and 50,000 for AD3.

over half of its edges in the first iteration, retaining nearly 40% after the second as well.

For all but AD3, the number of components after completion of the local phase is within 10% of the final number, and decreases rapidly in the first two iterations. For AD3 (with 1,600,000 nodes), the data appears in Table 5.

One of the biggest problems with most graph algorithms on distributed memory machines lies in managing to partition the graph across processors in such a way

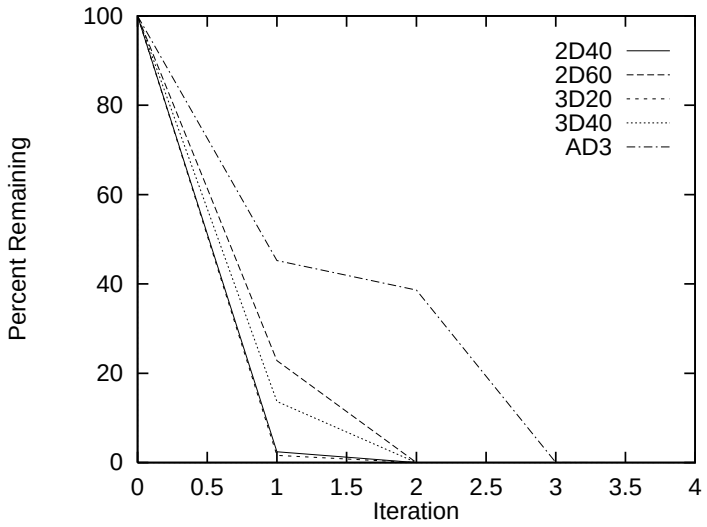

FIGURE 15. Percentage of remote edges remaining after each iteration. 100% corresponds to the state at the completion of the local phase.

Iteration	Components	Percent of Final Value
0	1,525,032	1602%
1	252,240	265%
2	100,671	106%
3	95,191	100%
4	95,190	100%

TABLE 5. Number of components remaining. For an AD3 graph with 1.6 million nodes, the table lists the number of components remaining after each iteration of the global phase.

that each processor has an approximately equal amount of work at each stage. For the 2D and 3D graphs, the natural partitioning provided by the underlying mesh performs quite well, keeping the variance across processors small except during the very last iteration (for which the time spent is much smaller anyway).

For AD3 graphs, however, there is no underlying topology, but the random nature of the graph helps to create a fair load balance. Unfortunately, the methods used for hooking in this algorithm tend to cause load imbalance fairly early in the AD3 processing, with a factors as high as 2.25 between some processors and the average arising while a significant fraction of edges and components remain. We plan to further optimize the solution of this type of graph as time permits.

7. Comparison with Earlier Work

Though a lot of research has been done in proposing theoretically optimal algorithms for finding connected components of a graph, not much work has been done in implementing these algorithms efficiently on parallel machines. Greiner [7] implemented the connected components algorithm on the Cray C-90 and on the

Connection Machine 2. However, the C-90 is a shared bus multiprocessor system, and the CM-2 is a SIMD machine. These machines are easier to program than distributed memory machines, which are however more scalable. Therefore, our work on implementing the connected components algorithm on the CM-5 exposes a new set of concerns and optimizations that were non-issues on the C-90 and the CM-2. By introducing the optimizations described in Section 4, we have created a highly efficient implementation of the connected components algorithm. For some of the graphs that we studied (*e.g.*, the 2D40 graphs), the execution time of our implementation is comparable to the results obtained by Greiner on the C-90. This is highly encouraging considering that our results were obtained on a 32-processor CM-5 without vector units, which is a much cheaper machine.

Another piece of related work is the implementation of the parallel clustering algorithm by Flanigan and Tamayo [**5**] on a CM-5 using CMMD, which is a message passing library provided by Thinking Machines Corporation. As mentioned earlier, the clustering algorithm requires a connected components labeling on a 2D mesh. Their implementation achieves a peak performance of finding connected components for about 12M nodes in a second on a 256-processor CM-5, a result comparable to our own. However, their implementation is optimized to labeling a 2D graph, which allows a compact representation of the edge list. As a result, they are able to greatly reduce the storage requirements for their algorithm. Bader and Jájá [**3**] adopt a similar strategy for identifying components of 2D images such that each component is a maximal collection of adjacent pixels with the same intensity. Our algorithm is more general-purpose since the input graph can have arbitrary connectivity, and we still obtain similar performance results. Also, our implementation is more portable than Flanigan and Tamayo's implementation since Split-C runs on a variety of parallel machines including the Paragon, the IBM SP-1 and SP-2, the Cray T3D, the Meiko CS-2, and a network of workstations. In fact, we were able to run our connected components program on a network of workstations without any change.

8. Conclusions

We have implemented the connected components algorithms on a distributed memory machine. We used a hybrid algorithm that combines the important aspects of the sequential and the PRAM algorithms. By using the Split-C language, which exposes the underlying machine to the programmer, we were able to enhance the performance of our implementation by treating local and global subgraphs separately, by paying attention to locality, and by tolerating remote memory access latencies. The resulting implementation is very efficient and obtains speedups on the order of 20 on a 32-processor CM-5 and 238 on a 256-processor CM-5. In related work [**9**], we demonstrate that our algorithm is the fastest in the world.

References

[1] R. H. Arpaci, D. E. Culler, A. Krishnamurthy, S. Steinberg, K. Yelick, "Empirical Evaluation of the Cray T3D: A Compiler Perspective," *Proceedings of the International Symposium on Computer Architecture*, 1995.

[2] B. Awerbuch, Y. Shiloach, "New connectivity and MSF algorithms for Ultracomputer and PRAM," International Conference on Parallel Processing, 1983, pp. 175-179.

[3] D. Bader and J. Jájá, "Parallel Algorithms for Image Histogramming and Connected Components with an Experimental Study," Journal of Parallel and Distributed Computing, June 1996.

[4] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, K. Yelick, "Parallel Programming in Split-C," Proceedings of Supercomputing '93, Portland, Oregon, November 1993, pp. 262-273.

[5] M. Flanigan, P. Tamayo, "A Parallel Cluster Labelling Method for Monte Carlo Dynamics," International Journal of Modern Physics C, Vol. 3, No. 6, 1992, 1235-1249.

[6] H. Gazit, "An Optimal Randomized Parallel Algorithm for Finding Connected Components in a Graph," *SIAM Journal of Computing* **20(6)**, December 1991.

[7] J. Greiner, "A Comparison of Parallel Algorithms for Connected Components," to appear in the Symposium on Parallel Algorithms and Architectures 1994.

[8] L. T. Liu, D. E. Culler, "Evaluation of the Intel Paragon on Active Message Communication," Proceedings of the Intel Supercomputer Users Group Conference, 1995.

[9] S. S. Lumetta, A. Krishnamurthy, D. E. Culler, "Towards Modeling the Performance of a Fast Connected Components Algorithm on Parallel Machines," Proceedings of Supercomputing '95, San Diego, California, December 1995, available at *http://www.supercomp.org.sc95/proceedings/465_SLUM/SC95.HTM.*

[10] S. Luna, "Implementing an Efficient Portable Global Memory Layer on Distributed Memory Multiprocessors," U. C. Berkeley Technical Report #CSD-94-810, May 1994.

[11] K. E. Schauser, C. J. Scheiman, "Experience with Active Messages on the Meiko CS-2," *Proceedings of the International Parallel Processing Symposium*, 1995.

[12] Y. Shiloach, U. Vishkin, "An *O(log n)* Parallel Connectivity Algorithm," Journal of Algorithms, No. 3, 1982, pp. 57-67.

[13] J. S. Wang, R. H. Swendsen, "Cluster Monte Carlo Algorithms," Physica A, No. 167, 1990, pp. 565-579.

Computer Science Division, University of California at Berkeley, Berkeley, California 94720

E-mail address: {arvindk,stevel,culler,yelick}@CS.Berkeley.EDU