DIMACS Series in Discrete Mathematics
and Theoretical Computer Science
Volume 00, 0000

Connected Components on Distributed Memory Machines

ARVIND KRISHNAMURTHY, STEVEN LUMETTA,

DAVID E. CULLER, AND KATHERINE YELICK

February 1, 1995

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 in 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 portition 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.

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.

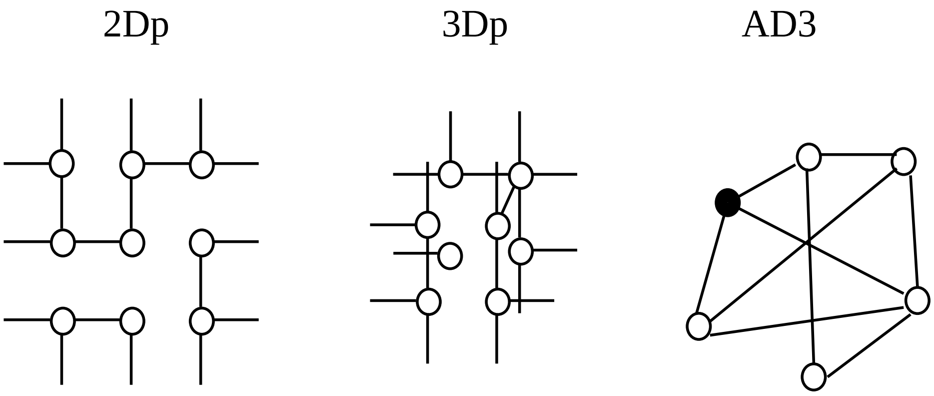

FIGURE 1. Graph types. The left and middle graphs correspond to real problems; p is the probability of presence for each edge on the underlying grid. The right graph is artificial; each node selects between zero and three other nodes as neighbors.

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 distributed memory MIMD model: 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 the Swendsen-Wang[**6**]. The algorithm simulates large systems near criticality by creating bonds between particles with similar spin and selecting a new spin for each particle cluster.

The graphs used for these applications have an underlying grid topology in either two or three dimensions. Because of the underlying topology, the graphs decompose easily into smaller fragments with relatively few edges between frag-

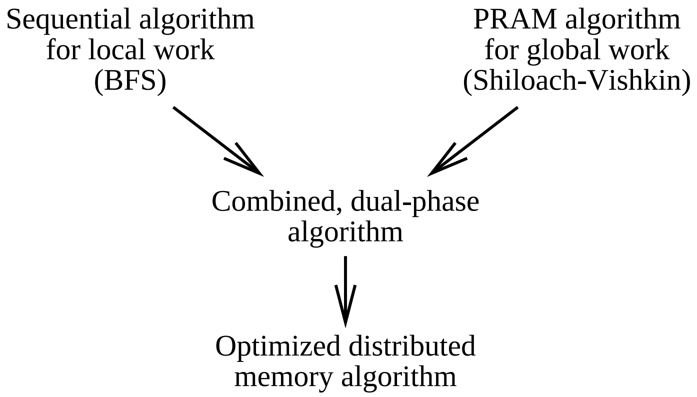

FIGURE 2. Hybrid algorithm strategy. We combine a PRAM algorithm with a sequential algorithm and then optimize the result to create an efficient algorithm for distributed memory machines.

ments, allowing much of the work in finding connected components to be performed locally. For our results, we use the random graphs typical of physics problems. For problems in vision and image recognition, the presence of each edge are not independent, but for the physics problems, the graphs are generated randomly, using a fixed probability for the presence of any particular edge. Figure 1 shows sample graph fragments.

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[**3**] 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 it therefore could be viewed as an extreme input to our connected components algorithm.

Previous parallel implementations of connected components algorithms have focused primarily on shared-memory machines[**3**]. 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 2 for the connected components problem. The two algorithms are merged, then the result is optimized.

[1]The tertiary graphs used by Greiner have an average degree of six. This almost always results in a graph that has only one connected component. We therefore resorted to lower degree graphs for our study.

In this paper, we present a hybrid algorithm for finding the connected components of a graph on a MIMD machine. We implemented and optimized the algorithm on a CM-5 using the Split-C language developed at Berkeley[**2**]. 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 optimization through which we developed the final version of our algorithm. In section 5, we present results from the CM-5 for the graph types of Figure 1. Section 6 compares our results with other implementations of the algorithm and Section 7.

2. The PRAM Algorithm

The sequential algorithm for labeling connected components requires just a simple search (either breadth-first or depth-first) over the graph. However the PRAM algorithm presented by Shiloach and Vishkin[**5**], which forms the basis for the global phase of our algorithm, is more complicated. In this section, we give a brief description of this algorithm. The algorithm requires $O(log\ n)$ parallel steps and a total of $O(m\ log\ n)$ work, where n is the number of vertices of the graph and m is the number of edges of the graph. In the following discussion, we denote the vertex set 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.

Beginning with V, the algorithm alternately performs a *hooking* operation to group connected vertices into trees and a *pointer doubling* operation to reduce the height of the trees. Progress occurs as long as the algorithm can decrease the number of trees in the forest or can decrease 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.

The *pointer jumping* operation replaces the parent of each vertex with the grandparent of the vertex, cutting the height of the each tree in half. Along with the pointer doubling operation, the value of the parent is also propogated to the child. A tree of height one, known as a *star*, is unaffected by pointer jumping. By making the parent of the root of a tree the root itself, we can 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 with and edge to a vertex in the target tree. Hooking occurs in two flavors: conditional and unconditional. The algorithm cannot tolerate cycles in the *Parent* relationship, so correctness requires the application of two conditions to hooking. Conditional hooking allows nodes of lesser value to hook to nodes of greater value, but not vice-versa. Unconditional hooking ignores this condition. A second requirement applies to both forms: since the value of a node might not be consistent with the value of the root of the tree until the tree has collapsed into a star, hooking is only performed on stars.

Clearly, unconditional hooking must be applied with care to prevent the formation of cycles. The algorithm first applies the conditional hooking operation, then applies unconditional hooking only to those stars neither hooked from nor hooked to during conditional hooking. Two stars cannot hook to each other during unconditional hooking since neither hooked during conditional hooking. The unconditional hooking operation is necessary to obtain $log(n)$ bound on the running time.

The algorithm follows:

(i) Parent(v) $\leftarrow$ v

(ii) Repeat until no change occurs in an iteration:

  a. If u belongs to a star, pick v such that $(u, v) \in E$ and $Value(Parent(u)) < Value(v)$ and set $Parent(Parent(u)) \leftarrow v$.
  b. If u belongs to a star that neither hooked to another node nor had another node hooked to it, pick v such that $(u, v) \in E$ and set $Parent(Parent(u)) \leftarrow v$.
  c. $Parent(u) \leftarrow Parent(Parent(u))$
  d. $Value(u) \leftarrow Value(Parent(u))$

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**, **2c**, and **2d**, 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 node 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 resides in the processor's local memory.

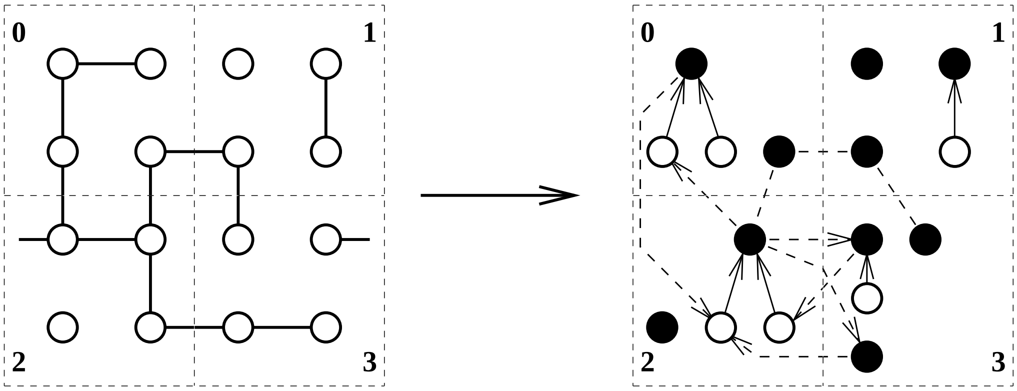

FIGURE 3. Local Phase. This phase processes all local edges to find local connected components.

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 [**2**][**4**] to simplify the task. Split-C provides the abstraction of a global address space on a distributed memory machine. Any processor may 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 can be used just like a local pointer, but it 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 thus allows us to represent a graph whose vertices are spread across processors and whose edges are represented using global pointers. Another aspect of global pointers that is useful for implementing hybrid algorithms 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. Also, the resulting distinction between local and global objects provides a clear cost model for introducing optimizations that we will examine later.

Having briefly discussed the language that we use to code in our implementation, we now introduce the algorithm:

(i) **Local Phase.** Perform purely local computations to decrease the size of the graph processed during the global phase. Figure 3 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.

(ii) **Global Phase.** Beginning with a list of components on each processor, all of which are stars and are marked with unique values, apply the (modified) Shiloach and Vishkin algorithm. Iterate over the following steps until done:

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. Figure 4 illustrates the process.
c. *Pointer Doubling Step.* Double parent pointers one or more times for each node and update the node's value from the new parent. Figure 5 illustrates the process.
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 will introduce this phase as an optimization and study its effect.

4. Optimization

In this section, we describe the sequence of optimizations used to tune our algorithm for distributed memory machines. On a CM-5, the optimized code runs about twenty times faster than the basic version described in Section 3. The optimizations, which we present in decreasing order of importance, all make use of three simple concepts in parallel optimization: reducing the amount of computation, reducing the number of remote references, and balancing the workload across processors.

4.1. Vertex Pruning. The first optimization, *vertex pruning*, takes advantage of the fact that the local phase marks a relatively small fraction of the graph nodes as star roots. Recall that all remote edges from nodes in a local star are moved to the root during the local phase. In the basic algorithm, the value of

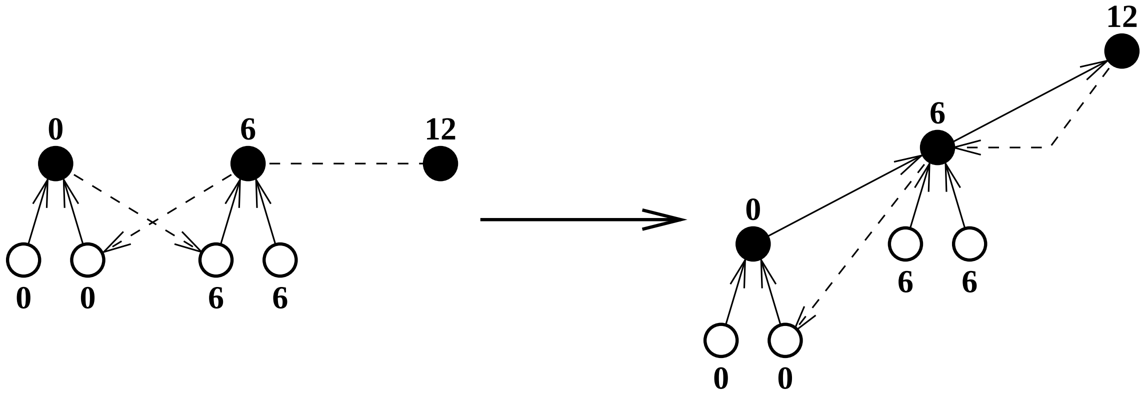

FIGURE 4. Conditional Hooking. The star numbered 0 hooks onto 6, which in turn hooks onto the star numbered 12.

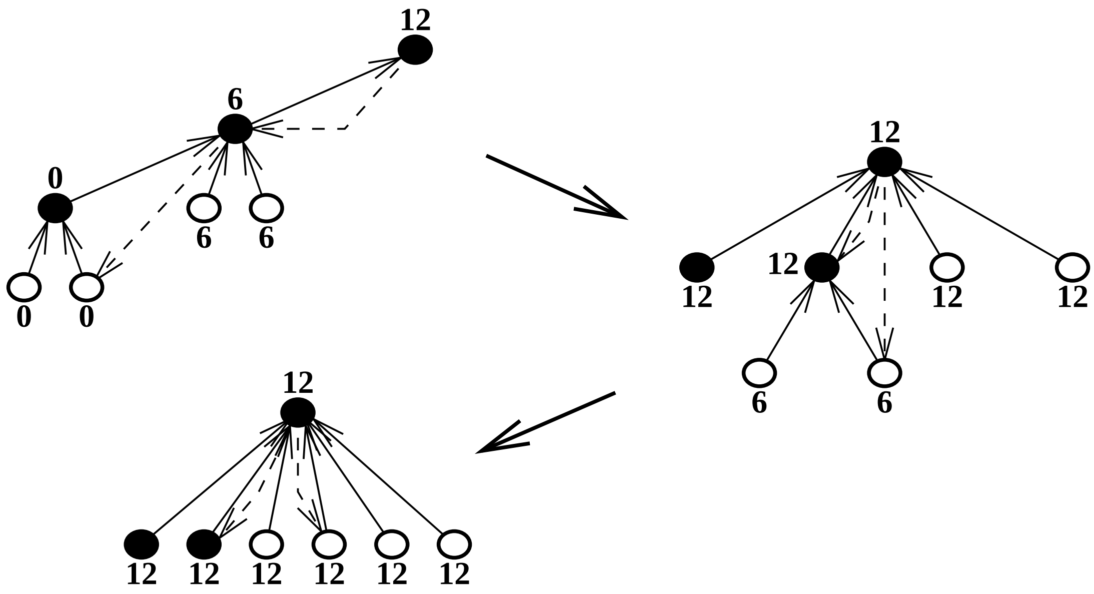

FIGURE 5. Pointer Doubling. Follow parent links and attach each node to its grand-parent.

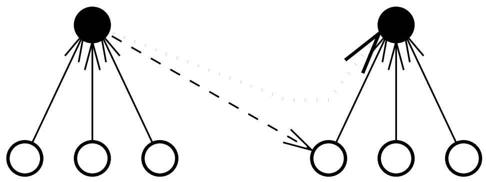

FIGURE 6. Vertex pruning optimization. Instead of comparing with the parent value before hooking, compare with the grandparent.

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; artifical graphs do not.

each node must be kept consistent to avoid the creation of a cyclic graph during the conditional hooking step.

We relax the consistency requirement through the following observation: if u is a star root and (u, v) is a remote edge from u to v, then $Parent(v)$ is also a star root. By comparing with the value of $Parent(v)$ rather than the value of v during conditional hooking, we eliminate all references to leaf node values during the global phase. Figure 6 illustrates the change: the dashed line represents the remote edge used in the basic algorithm to determine whether to hook the left star to the right star, and the dotted line represents the virtual edge used with the vertex pruning optimization. The leaf node values can then be left inconsistent during the global phase, so we prune the leaf vertices from the list after the local phase and add a step at the end of the algorithm to update the values:

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

Note that the update step is completely local—any node with a remote parent is a star root and already has a consistent value.

By reducing the work in the value propagation step, vertex pruning greatly reduces the running time of the algorithm for many important graphs. The graphs for many real-world problems divide easily between processors and result in large stars, as shown in Table 1. The average star size depends strongly on graph connectivity, but vertex pruning results in a significant speedup even with the less connected graphs. Artificially generated graphs such as AD3, on the other hand, do not partition well between processors and generally create

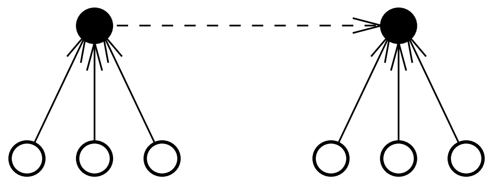

FIGURE 7. Edge expansion optimization. Prior to the global phase, we expand remote edges to point only to star roots.

single-node stars. For these types of graphs, the cost of the extra update step outweighs the reduced work benefits of vertex pruning. The main disadvantage of vertex pruning, however, is the extra dereference required for each check during conditional hooking. Since the parent of a remote node is generally remote, retrieving the value of the parent usually requires a second remote reference, doubling the number of remote references during the conditional hooking step.

4.2. Edge Expansion. Fortunately, we can reduce the extra references introduced by vertex pruning by applying a second optimization, *edge expansion*. Instead of performing two remote references for each check, we expand remote edges across a parent link before entering the global phase:[2]

1.c. *Edge Expansion Step.* Replace every remote edge from u to v with an edge from u to $Parent(v)$.

The virtual edge (the dotted line) of Figure 6 becomes the actual edge, as shown in Figure 7. Edge expansion speeds up the algorithm by replacing potentially many remote references for each edge (possibly one for each iteration in the global phase) with a single remote reference. Note also that the references in the new step can be pipelined to further reduce communication cost, whereas the references added by vertex pruning cannot be, since the dereferencing processor needs the parent's value to continue.

4.3. Postponing Edge-List Concatenation. In our initial implementation, we chose to migrate edges to a node's parent after the pointer doubling step in each iteration of the global phase rather than waiting until the new structure had collapsed into a star. Although our choice clearly results in more messages and more work, we felt that avoiding network contention at the new star root would outweigh the extra cost.

We failed to foresee two other problems, however. First, the operation required very complex code: a node might attempt to receive new edges from several children and send old edges to its parent at the same time, and the code had to ensure atomicity for all remote operations to prevent conflict. Second, although the edge-list for a node is at first completely local, the process of concatenation adds remote sections to the list. The extra list traversals become much more

[2]The new step accesses remote data and should technically not be added into the local phase, but the numbering we chose (1.c.) indicates both the appropriate insertion point for the step and fact that the step is not performed as part of the global phase iteration.

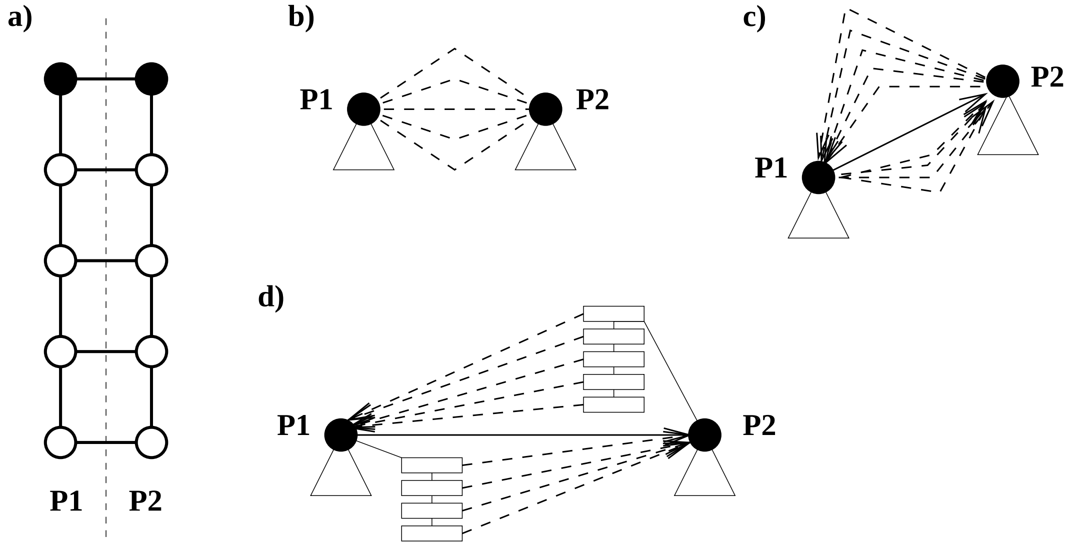

FIGURE 8. Edge list reordering optimization. a) Graph fragment distributed across two processors; b) Fragment after the local phase; c) Fragment after conditional hooking; d) Data structure view of fragment after hooking.

expensive when the data must be fetched from remote processors.

In light of these two difficulties, we decided to postpone edge-list concatenation on a node until the node belongs to a star. We add a condition to the edge-list concatenation step of the algorithm:

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

4.4. Removing Self-Loops before Edge-List Concatenation. Multiple links between two stars can be eliminated quickly and easily once the stars have been hooked and collapsed into a new star: all of the remaining edges point to nodes with the same value as the star root. In the basic algorithm, we check for self-loops after the edge-list concatenation step. Alternatively, we might swap the order of these two steps and perform self-loop removal first.

Consider the graph fragment shown in Figure 8a. Five nodes on each of two processors are joined by five remote edges. The local phase collapses the fragment into two stars with five edges in each direction, as we see in Figure 8b. A conditional hooking step uses a single edge in one direction as a new parent link, leaving nine remote edges that the self-loop step must eventually discard, as we see in Figure 8c. Moving edges from $P1$ to $P2$ before checking for self-loops may at first seem the best option since the new edges point to the star root on P2 and can be discarded with negligible effort. However, by looking more carefully at the data structures used for the graph, we discover that the opposite is true.

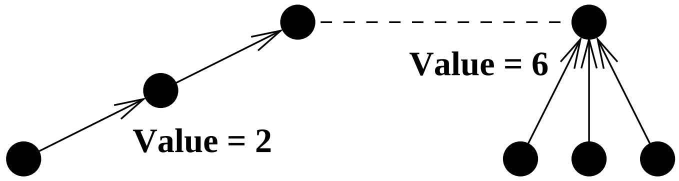

FIGURE 9. Unconditional hooking optimization. Hook stars that remain stagnant during conditional hooking.

Figure 8d shows the two node groups along with the remaining edge-lists. Note that the edge-lists do not move between processors—when we concatenate two edge-lists, we do so using a remote pointer. Hence, if we concatenate the lists before checking for self-loops, $P2$ must perform nine remote references: five to read the value of the black node on $P1$ and four more to read the portion of the edge-list that resides on $P1$, even though all four of those edges merely point back to the black node on $P2$. If we check for self-loops before concatenating the edge-lists, $P2$ need only perform five of the remote references, while $P1$ handles the other four. Neither option is strictly advantageous in terms of work or number of remote references, but the latter balances the load more evenly, so 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.

4.5. Unconditional Hooking. Our initial implementation does not fully implement the Shiloach-Vishkin algorithm. We deliberately ignored a step designed to handle pathological cases that for practical purposes never appear. We added the step later to determine the effect on performance and were surprised to find that the algorithm actually runs faster with the extra check.

The extra check is the unconditional hooking operation. During the conditional hooking step, we mark any star that neither hooks to another star nor has another star hooked to it as *stagnant.* After the conditional hooking is finished, we unconditionally hook all stagnant stars using the first edge on their edge-lists. Although the pathological cases do not occur on an asymptotically significant scale, they do appear with small sets of nodes and can increase the number of iterations. Consider, for example, the situation in Figure 9. The left group of nodes has yet to collapse into a star, and is not considered during conditional hooking. The right group has fully collapsed, but has a value higher than that of the left group, and cannot hook either. Without unconditional hooking, the algorithm must perform an extra iteration while it waits for the left group to finish collapsing. Unconditional hooking allows the right group to hook immediately to the left.

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

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 in the global phase.

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.

The costs are obvious. Generating information on stagnation means more work and more remote references. Cycling through the stars and hooking the stagnant ones adds even more overhead per iteration. The benefits are perhaps not so obvious. The data in Table 2 demonstrate the usefulness of unconditional hooking. 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 time lost in the new steps.

4.6. Synchronous and Asynchronous Pointer Doubling. The pointer doubling operation appears often in parallel algorithms, primarily as a means of operating on a linked structure in time logarithmic rather than linear in the size of the structure. As shown in Figure 10, two pointer doubling steps collapse a tree of height four into a star. The standard pointer doubling implementation is not without disadvantages, however. For each pointer doubling, the processors loop over all nodes and must synchronize before beginning the next doubling. Without synchronization, the number of pointer doublings required to collapse the tree might be as many as the height of the tree. On the other hand, the number might also drop to one, depending on timing and node orderings on each

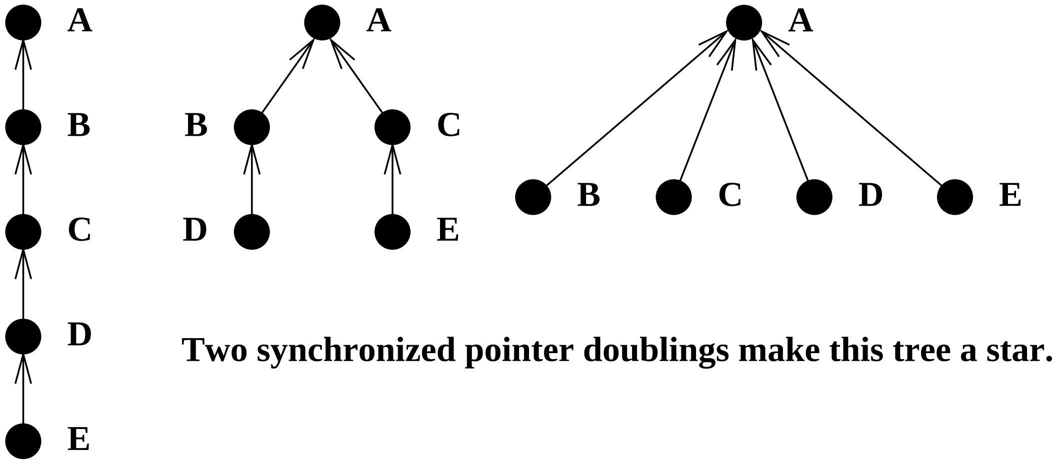

FIGURE 10. Asynchronous pointer doubling optimization. Do not synchronize processors between pointer doublings.

processor.

By reversing the loop order and removing the barrier synchronizations from the pointer doubling step of the basic algorithm, we explored the idea further. With the new code, we loop only once over the nodes owned by a processor. Within the node loop, we perform a fixed number of pointer doublings on a node. We call this form *asynchronous pointer doubling*. Surprisingly, the algorithm runs in almost exactly the same time. Elimination of the synchronization and extra loop overhead compensates for the lack of strictly logarithmic collapse time. Barrier synchronizations are relatively inexpensive on the CM-5, so asynchronous pointer doubling is likely to be more beneficial on machines with more heavyweight barriers.

4.7. Maximal Pointer Doubling. The basic algorithm does not specify the number of pointer doublings performed for each node during the pointer doubling step. We found that for the graphs we studied, three to seven doublings were fastest. As an alternative to a fixed number of doublings, we investigated *maximal pointer doubling*, in which each iteration of the global phase completely collapses every structure into a star. Although maximal pointer doubling can increase the amount of work and can create load imbalance by forcing all processors to wait while one collapses a structure, there are also advantages. The star marking step, for example, is no longer necessary since all structures are stars. Further, conditionals dealing with stars can be removed. The end result is a slightly faster algorithm; the advantages of speeding up the other steps outweigh the cost of the extra doublings.

4.8. Final Algorithm. The optimized algorithm follows:

(i) **Local Phase.** Perform purely local computations to decrease the size of the graph processed during the global phase. Figure 3 illustrates the effect of the local 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. *Edge Expansion Step.* Replace every remote edge from u to v with an edge from u to $Parent(v)$.

(ii) **Global Phase.** Beginning with a list of components on each processor, all of which are stars and are marked with unique values, apply the (modified) Shiloach and 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 star 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 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) *Unconditional Hooking Step.* Attach stagnant star components to other components. Remove each newly attached component 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. *Self-Loop Removal Step.* For each star, remove edges that point to nodes with the same value (nodes within the star).

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

(iii) **Update Phase.** For each node with a local parent, update the value of the node from the value of the parent.

5. Performance Measurements

In this section, we discuss our measurements of the optimized algorithm running on a 32 processor CM-5 and 512 processor CM-5 (we could only obtain one set of data on the latter). We begin with a short discussion of the graph types; performance of the algorithm depends strongly on the graph structure. After explaining our methodology, we 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.

5.1. Choice of Benchmark Graphs. The first two types of graphs used in our measurements are built on a two dimensional toroidal mesh. Each edge in the mesh is present with some fixed probability, either 40% or 60%. Since one expects a graph with average degree below two to be mostly disconnected and a graph with average degree above two to be mostly connected, these two percentages outline the region of interesting problems for two dimensional grids. We call these graphs 2D40 and 2D60, following the notation given by Greiner[**3**]. We divide the underlying rectangular mesh into P square chunks, where P is the number of processors.

The second two types of graphs are built in the same fashion on a three dimensional toroidal mesh. The boundary between mostly connected and mostly disconnected graphs falls at 33% in the three dimensional case, so our measurements use edge presence probabilities of 20% and 40%. These graphs are named 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 from zero to three other nodes as neighbors, resulting in an undirected 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 for AD3 and the generation method lead to extremely poor locality.

5.2. Effect of Randomization. For each type of graph, we generate several instances that meet the given connectivity specification using a different random number seed. Except where noted, the timings represent an average over the graphs in that class.

The variations in timings introduced by random graph generation is notice-

Graph	Nodes	Edges	Average Time (seconds)	Stan. Dev. (seconds)	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

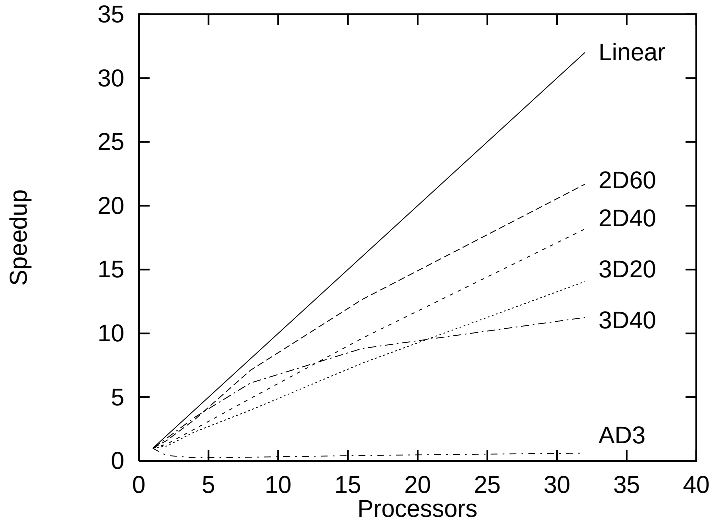

FIGURE 11. Speedup a problem size of 256K nodes.

able. Table 3 gives the variation in execution time for large samples of all graph types running on 32 processors. Twenty seeds were chosen at random and then 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 tend to be larger for more strongly connected graphs, but also tend to rise around the boundary regions (where some graphs are mostly connected and others are mostly disconnected). The largest variations occur in the AD3 graphs, where the cost of remote references must be factored into the variation in number of references.

There is nondetermism in the parallel algorithm due to the ordering of events during execution. However, the variation on overall performance is slight compare to that of different random graphs in the same class.

5.3. Speedup. Speedup of the parallel version is measured by comparing to a sequential implementation that contains only the local phase of of the algorithm. Although the global phase requires no communication on a single processor, it takes a significant amount of time that is not relevant to the sequential

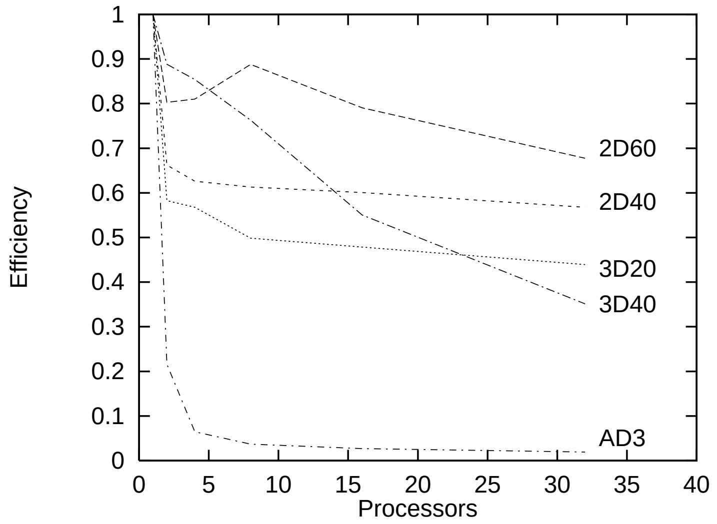

FIGURE 12. Efficiency for a problem size of 256K nodes.

execution time of the program.

Figure 11 shows the speedup for a fixed problem size (262,144 nodes) on between 1 and 32 processors of a CM5. 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 graph leads to unavoidable communication. Figure 12 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 13 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 node on 32 processors.

A second definition of speedup, *scaled speedup*, uses a problem size proportional to the number of processors. In Figure 14, 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 15, 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.

5.4. 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

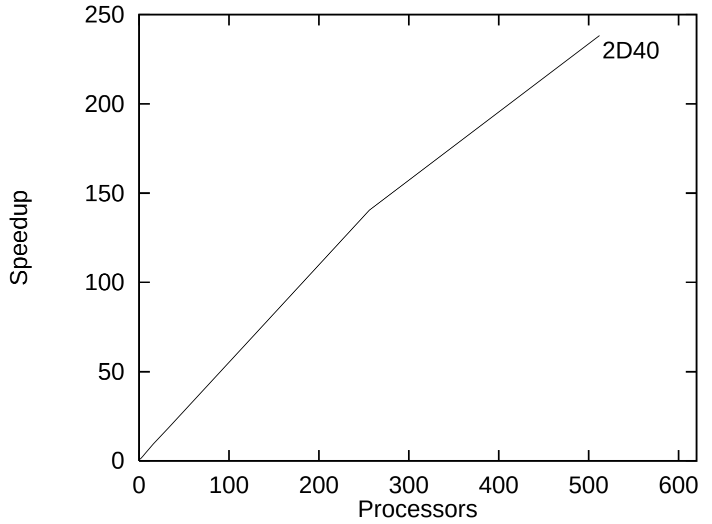

FIGURE 13. Speedup for a 2D40 graph up to 512 processors. The problem size is 256K nodes between 1 and 32 and 8M nodes bewteen 32 and 512.

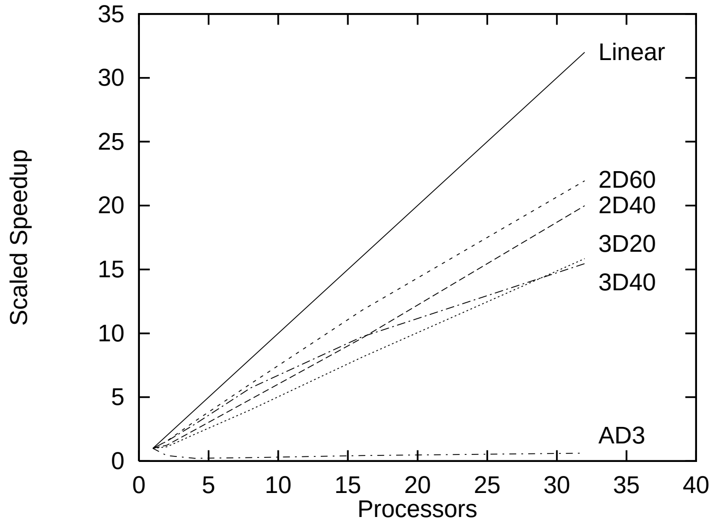

FIGURE 14. Scaled speedup. The problem size per processor is fixed: 62500 nodes for 2D40 and 2D60, 125000 for 3D20 and 3D40, and 50000 for AD3.

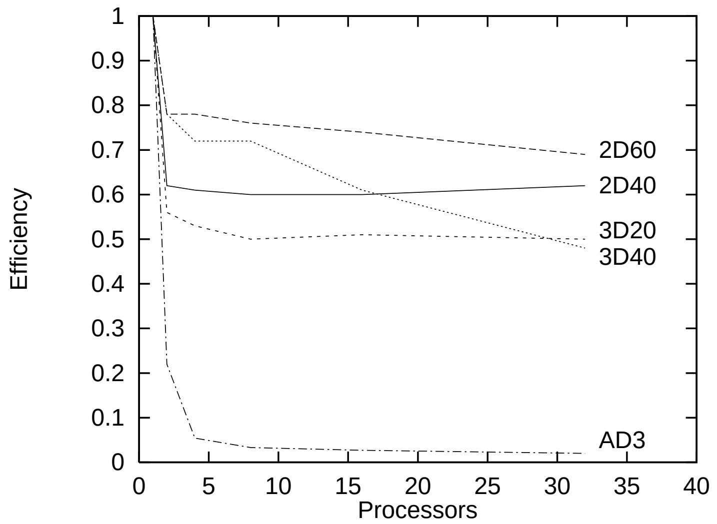

Figure 15. Scaled efficiency. The problem size per processor is fixed: 62500 nodes for 2D40 and 2D60, 125000 for 3D20 and 3D40, and 50000 for AD3.

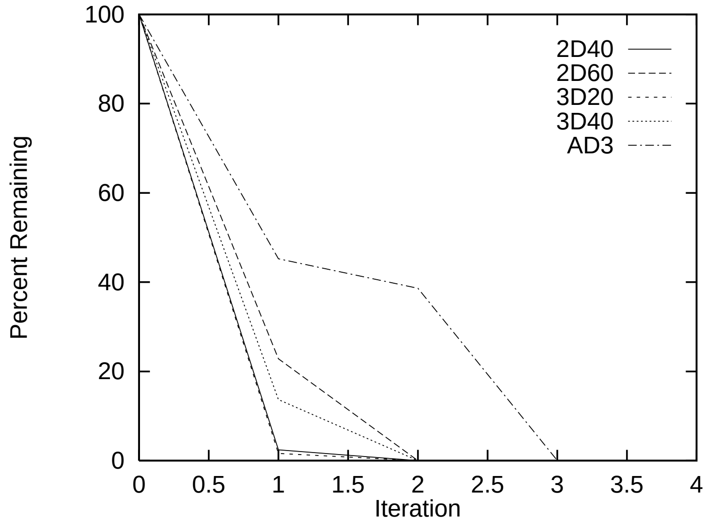

Figure 16. Edges remaining after each iteration (0 corresponds to completion of the local phase)

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 4. Number of Components after Each Iteration on an AD3 Graph

a few more iterations, averaging about 3 or 4. In Figure 16, 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 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 4.

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 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.

6. 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[**3**] 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 CM2 is a SIMD machine. These machines are easier to program than distributed memory MIMD machines, which are however more scalable. Therefore, our work on implementing the connected components algorithm on the CM5 exposes a new set of concerns and optimizations that were non-issues on the C-90 and the CM2. By introducing the optimizations described in Section 4, we have an highly efficient implementation of the connected components algorithm. For some of the graphs that we studied (2D graphs with 40%), 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-node CM5 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[**7**] on a CM5 using CMMD, which is a message passing library provided by Thinking Machines Corporation. As mentioned earlier, the clustering algorithm requires a connected components labelling on a 2D mesh. Their implementation achieves a peak performance of finding connected components for about 12M vertices in a second on a 256 processor CM5, which is comparable to our results. However, their implementation is optimized to labelling a 2D graph, which allows a compact representation of the edge list. As a result, the storage requirements of their algorithm is lesser. However, our algorithm is more general-purpose, and we still obtain similar performance results. Also, our implementation is more portable since Split-C runs on a variety of parallel machines including the Paragon, the SP1, the T3D, the Meiko, and a network of workstations. In fact, we were able to run our connected components program on a network of workstations without any changes[3].

7. 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 in the order of 20 on 32 processor machines.

References

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

[3] The performance numbers on a network of workstations is currently not impressive due to the lack of a specialized low-latency network.

2. 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.
3. J. Greiner, "A Comparison of Parallel Algorithms for Connected Components," to appear in the Symposium on Parallel Algorithms and Architectures 1994.
4. S. Lumetta, "A Debugger for the Split-C Language," available from author.
5. Y. Shiloach, U. Vishkin, "An *O(log n)* Parallel Connectivity Algorithm," Journal of Algorithms, No. 3, 1982, pp. 57-67.
6. J.S. Wang, R. H. Swendsen, "Cluster Monte Carlo Algorithms," Physica A, No. 167, 1990, pp. 565-579.
7. 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.
8. T. von Eicken, D. E. Culler, S. C. Goldstein, K. E. Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation," Proceedings of the International Symposium on Computer Architecture, 1992
9. Thinking Machines Corporation, "CMMD Reference Manual," Version 3.0, May 1993.

Computer Science Division, University of California, Berkeley, California 94720
E-mail address: arvindk@cs.berkeley.edu

Computer Science Division, University of California, Berkeley, California 94720
E-mail address: stevel@cs.berkeley.edu

Computer Science Division, University of California, Berkeley, California 94720
E-mail address: culler@cs.berkeley.edu

Computer Science Division, University of California, Berkeley, California 94720
E-mail address: yelick@cs.berkeley.edu