Connected Components on Distributed Memory Machines

Arvind Krishnamurthy, Steven Lumetta,
David E. Culler, and Katherine Yelick
Computer Science Division
University of California, Berkeley *

Abstract

In this paper, we describe an implementation of the connected components algorithm on a distributed memory machine. A direct implementation of the PRAM algorithm results in an inefficient implementation due to the huge number of remote accesses generated by the algorithm. Instead, we use a hybrid algorithm that invokes the sequential algorithm as a local preprocessing phase before entering a global phase, which is a modified version of the PRAM algorithm. We use the Split-C language, which provides the abstraction of a global address space, for building the distributed graph data structure. We obtain speedups in the order of 20 on a 32 processor CM5 for certain kinds of graphs.

1 Introduction

Although the asymptotic running times of numerous PRAM algorithms for connected components have been studied within the theory community, there have been very few attempts to make use of these algorithms on real parallel machines. The implementations that do exist generally appear only on shared memory platforms such as the Cray C90 or on SIMD machines such as the CM-2 where messages between processors require only a single cycle.

The large parallel machines of the future, however, lean towards the distributed memory MIMD model, with the machines built using standard fast processors that are loosely coupled using a network (e.g., TMC CM-5, Meiko CS-2, Cray T3D, Intel Paragon, IBM SP-1). A straightforward implementation of a PRAM algorithm on any of these machines 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, hybrid, approach is to employ a PRAM algorithm in conjunction with a

*This material is based upon work supported under a National Science Foundation Graduate Research Fellowship and National Science Foundation Infrastructure Grant number CDA-8722788. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

standard sequential algorithm, using the latter to manage operations local to each processor and the former to manage the interaction between processors.

In this paper, we explore a set of PRAM algorithms for finding the connected components of a graph using this hybrid approach. Our selection of algorithms and graphs is based on a study by Greiner [3] of the pragmatic aspects of connected component PRAM algorithms on shared memory and SIMD platforms. We extend the results to MIMD machines, namely the CM-5, using the Split-C language developed at Berkeley [2]. We discuss the standard PRAM algorithm in section 2, and a simple implementation of the algorithm is presented in section 3. In section 4, we describe the different optimizations used to improve the running time. In section 5, we present the results from running the program on a CM5, and section 6 compares our results with other implementations of the algorithm.

2 Basic Algorithm

Our implementation is based on the connected components algorithm presented by Shiloach and Vishkin [5]. In this section, we will briefly describe the different stages of the algorithm. The basic 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. We will denote the vertex set by V and the edge set by E in the following discussion.

The algorithm starts with the set of n vertices, and repeatedly groups vertices that have edges between them. The algorithm uses two basic operations: *pointer jumping* and *hooking operation*. 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 *pointer jumping* operation takes each vertex in the forest, and makes the current grandparent the new parent of the vertex. 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 depth 1. The parent of the root of a tree is the root itself, which simplifies the pointer jumping phase, allowing it to take the form:

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

The *hooking operation* hooks a star in the forest to another tree in the forest if the star contains a vertex that is adjacent to some vertex in the target tree. This operation appears in two flavors: conditional hooking and unconditional hooking. Let u be a vertex that belongs to a star, and let (u, v) be an edge in the graph. Assume that there is a numbering of the vertices given by a $Value$ attribute. Then, the conditional hooking operation sets $Parent(Parent(u))$ to v if

$Value(Parent(u)) < Value(v)$. The unconditional hooking operation sets $Parent(Parent(u))$ to v irrespective of the values of the vertices being linked.

An invariant is that the parenthood relationship should not contain cycles. The conditional hooking operation ensures that there are no cycles formed as a result of applying the operation; 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 on those stars that were not hooked in the conditional hook phase. This prevents two stars from linking up with each other since at least one of the stars has had an opportunity to be attached to the other star during the conditional hooking operation. The unconditional hooking operation is necessary to obtain $log(n)$ bound on the running time, but is not necessary for correctness[5].

We can now provide the pseudo-code for the algorithm. We assume there is one processor for every vertex and for every edge in the graph. The processors can therefore be classified into vertex processors and edge processors. The algorithm requires concurrent write ability from the PRAM processors to execute each step described below in constant time.

1. Parent(v) $\leftarrow$ v
2. Repeat until there is no change:
   - (a) If u belongs to a star, pick v such that $(u, v) \in E$ and $Value(Parent(u)) < Value(v)$, then $Parent(Parent(u)) \leftarrow v$.
   - (b) If u belongs to a star and $(u, v) \in E$, then set $Parent(Parent(u)) \leftarrow v$.
   - (c) Parent(u) $\leftarrow$ Parent(Parent(u))
   - (d) Value(u) $\leftarrow$ Value(Parent(u))

The loop requires $O(log\ n)$ iterations to terminate. The processors are assumed to execute in a lock-step manner. The vertex processors are active during steps **1**, **2c**, and **2d**. The edge processors are utilized for steps **2a** and **2b**. The concurrent write requirement is essential for steps **2a** and **2b** since any of the edge processors associated with the vertex u can find a target tree to hook onto. However, there are no assumptions made about the policy for disambiguating writes to the same location.

3 Implementation

In this section, we describe our initial attempt at implementing the connected components algorithm. Although the first implementation proved to be inefficient, detailing it will facilitate the description of our improvements in later sections.

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 those data which reside within the processor's local memory. In the global phases, the algorithm must address issues that arise when some data resides in the memory of remote processors and must make an effort to handle these remote references as efficiently as possible.

Fortunately, we can make use of the Split-C language [2] to simplify our task. Split-C provides the abstraction of a global address space and simple but powerful data motion primitives, allowing the programmer to optimize his program in a straightforward manner to any degree desired, and one need not second guess or work around the compiler. Since Split-C is based on Active Messages [6][7], we can afford to use fine-grained parallelism instead of a more difficult (and often non-intuitive) coarse-grained approach.

When discussing the algorithm, we may refer to pointer objects as being either local or remote, referring to the location of the data item. Distinguishing between these two types of pointers is trivial in Split-C, but it is not necessary—one can choose to treat either as a pointer into the global address space.

In addition to the natural flavor of the Split-C language, it has the added advantage of providing debugging support via the Split-C Debugger [4]. This kind of support is a key factor in writing any kind of program, but is poor in some parallel programming environments.

Having briefly discussed our tools, we can now introduce the algorithm. Although this version looks very different from the final version, it will aid in understanding the optimizations we made and introduce the general style of the program. The algorithm follows:

1. Each processor performs a sequential connected components algorithm using a Breadth First Search (BFS) on its nodes, ignoring remote edges for the purposes of the search, to produce a forest.

2. Unlike the sequential version, the structure of the forest on each processor must be maintained in the parallel version to allow propagation of values in the global phase. To this end, a representative node is chosen for each local tree, all remote edges from nodes in the tree are moved to this representative, and the tree is collapsed into a star.

3. Beginning with a list of components on each processor, all of which are stars and are marked with unique values, we use Shiloach and Vishkin's algorithm to handle the global phase of the algorithm. We iterate over the following until all nodes are marked as done:

   (a) If all components on all processors are done, quit.

   (b) Mark star components with no remaining outside edges as done.

   (c) Attach star components to other components if the value of the other component is larger. Remove components that have been attached from the component list.

   (d) Double parent pointers one or more times for all nodes and propagate *Value* and *done* attributes.

   (e) Mark components as stars or non-stars. This is done by first marking all components as stars, then marking the grandparent of each node as a non-star if it is distinct from the parent of the node.

   (f) If remote edges exist for a node, and the parent of the node is a star, move the edges to the parent node.

   (g) Remove the graph edges of star components that point to nodes with the same value.

Note that in the first implementation, we ignored the unconditional hooking phase. We added it later, but whether or not it is needed depends on the structure of the graphs to be used.

4 Improvements

In this section, we describe techniques to improve the performance of our implementation. The optimizations fall into two classes: improving constant factors in the total amount of computation and decreasing the number of non-local references. Our implementation is tuned for distributed memory machines, showing a factor of 20 improvement over the basic version described in Section 3.

4.1 Vertex Pruning

The first optimization is based on the observation that only the roots of stars need to be considered in the global phase, since all remote edges in the star have been moved to the root. We can therefore ignore the leaves of the stars throughout the global phase, if we make two changes to the algorithm. First, we modify the hooking operation, which links u to v if $(u, v) \in E$ and $Value(u) < Value(v)$. Instead, we link u to $Parent(v)$. The values at leaf nodes are ignored and therefore need not be updated during the global phase. This allows the values of the leaves of the trees to be inconsistent since we always examine nodes that are parents. Note that this change results in extra dereferences during the hook operation. However, this change allows us to avoid updating the leaves of the stars found during the local BFS operation. The second modification is the addition of a final pointer doubling operation for the parent and value fields of these leaf, once the global phase is complete.

4.2 Edge Expansion

We can avoid the extra dereferences introduced by vertex pruning in the hooking phase by *edge expansion*. If $(u, v) \in E$, after the local BFS is complete, we replace the edge by $(u, Parent(v))$. This improves the running time of the algorithm since we pay for the cost of the extra dereference only once, instead of repeatedly following parent pointer links over the many iterations of the global phase.

4.3 Postponing Edge List Concatenation

Our initial implementation concatenated the edges of the leaves to the edge list of its parent at the end of every pointer doubling phase. There were two problems with this approach. First, the code required for this operation was complex since it had to handle race conditions where both the parent and the child might be involved in edge-list concatenation operations. The second concern is that to concatenate two edge lists, we need to traverse one of the lists and find its last element. Since the edge list concatenation operation occurred at the end of every pointer doubling operation, the algorithm was traversing edge-lists multiple times once for every doubling operation. We can alleviate both these problems by postponing this concatenation operation till the pointer doubling phase of a tree is complete, and the tree gets a star structure. This avoids complex race conditions as well as repeated traversal of edge-lists.

4.4 Checking for Duplicates before Edge-List Concatenation

Duplicate checking is the operation in which we remove edges from vertices that point to other vertices in the same tree. The original algorithm checked for duplicate edges just before the hook operation. At this stage of the algorithm, all the edges originating from the star are attached to the root of the star. However, since the edge-list concatenation process results in edge-lists that are spread across processors, a traversal of an edge-list might result in a huge number of remote references. Instead, if we execute the duplicate checking phase ahead of the edge-list concatenation phase, each node in the tree checks for duplicates in its own edge-lists. This improves the likelihood that edge-lists are local when checking for duplicates. Also, since the duplicate checking operation is done in a distributed manner by all the nodes in the tree, there is more parallelism.

4.5 Unconditional Hooking

The unconditional hook operation attaches any star that did not get attached in the earlier conditional hook phase. Incorporating the unconditional hook operation in our algorithm, decreases the number of iterations the algorithm needs to run, since it increases the number of hooking operations per iteration. However, since unconditional hooking can be applied only to those stars that did not get attached in the earlier conditional hook phase, we maintain a `stagnant` field for each root, which gets set after a successful conditional hook.

4.6 Synchronous and Asynchronous Pointer Doubling

We experimented with the manner in which the pointer doubling operations are performed. The standard approach is for each processor to iterate over the set of vertices that it owns and double the parent links for each vertex. The processors synchronize and repeat this process a specified number of times. Since the processors synchronize at the end of each pointer doubling operation, the depth of the trees decrease by a factor of two for every such operation. The other approach is to reverse the inner and outer loops. Each processor picks a vertex it owns and follows its parent links for some specified number of times before it picks the next vertex it owns. There is no synchronization in this approach, and each processor traverses its local list of nodes only once, so the cost of each iteration is lower. However, there are no guarantees that the depth of the tree decreases exponentially. We implemented both of these schemes, and studied their performance. Surprisingly, there was little difference between the running times of the two approaches. The extra overhead of synchronizing and traversing the local lists multiple times compensates for the benefits obtained from ensuring that the size of the tree decreases by a factor of two every iteration.

4.7 Aggressive Pointer Doubling

We also studied the effect of varying the number of pointer doublings between two successive hooking phases. The optimal number varied from 3 to 7, depending on the structure of the graph. We then decided to do what we call *maximal pointer doubling*. The idea is to reduce trees to stars by running the pointer doubling operation as many times as necessary for every iteration of the global phase. The resulting execution times were slightly worse, but now the global phase left only stars and no general trees, which enabled other optimizations. After eliminating the phase in which trees are marked as stars, and also eliminating certain conditionals in our code that checked for stars, the resulting program ran faster than the program that did an optimal number of pointer doublings.

5 Performance Measurements

We made a number of measurements of the optimized algorithm running on a 32 processor CM5.

5.1 Graph Construction

Before we begin to discuss the actual measurements, we shall spend some time discussing the types of graphs used. We drew four of the five types of graphs used directly from Greiner [3], but the fifth type given in that work we found to be fairly pointless (the graphs have one connected component with high probability), and instead used a modified form of the graph.

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 2 to be fairly disconnected and a graph with average degree above 2 to be fairly connected, these two percentages outline the boundary region for the two dimensional grid. We shall call these graphs 2D40 and 2D60, following the notation given by Greiner. Both graphs are best divided among the processors by using the underlying mesh, splitting the mesh into equal chunks, and placing one chunk on each processor.

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 at 33% in the three dimensional case, so our measurements use edge presence probabilities of 20% and 40%. These graphs will be known as 3D20 and 3D40. The underlying mesh again provides the best method of partitioning the graph among processors.

The last graph is a derivative of the Tertiary graph given in Greiner. In the Tertiary graph, each node picks three neighbors at random using a uniform distribution across all nodes. The average degree of the graph is therefore 6, and the minimum degree is 3 (counting self-loops). If one thinks about the probability that some subset of nodes will be isolated from the rest of the nodes, one quickly sees that the probability of more than one connected component is quite low. This type of graph did not interest us, since it is better solved by simpler search and mark methods. We therefore changed the definition to create graphs of average degree 3, denoted AD3, by having each node select between 0 and 3 neighbors from a uniform distribution. This results in a less connected graph, with about 95,000 components in a typical 1,600,000 node graph.

Unfortunately, because of the high probability in the AD3 graph that any edge will be remote ($\frac{P-1}{P}$, where P is the number of processors), it is very hard to solve these graphs efficiently on a distributed memory machine. This will be apparent in the results.

Graph	Nodes	Edges	Average Time (sec)	Stan. 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 1: Variation in Execution Time

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 2: Variation in Time per Node as a Function of Graph Size (on 32 Processors)

5.2 Variation Between Graphs of a Single Type

Except where noted, all of the measurements were averaged over several runs of the algorithm using distinct random seeds. Although timings also vary slightly because of non-deterministic behavior of the algorithm, the variation between graphs in the same class is much more significant.

An unforeseen side effect of our random number generation led to one of the processors creating a graph independent of the random seed. Although this practically irrelevant for large numbers of processors, the effect 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 on both the high and low end of the execution time spectrum, but the single processor runs found only a single graph.

Table 1 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.

5.3 Speedup and Efficiency

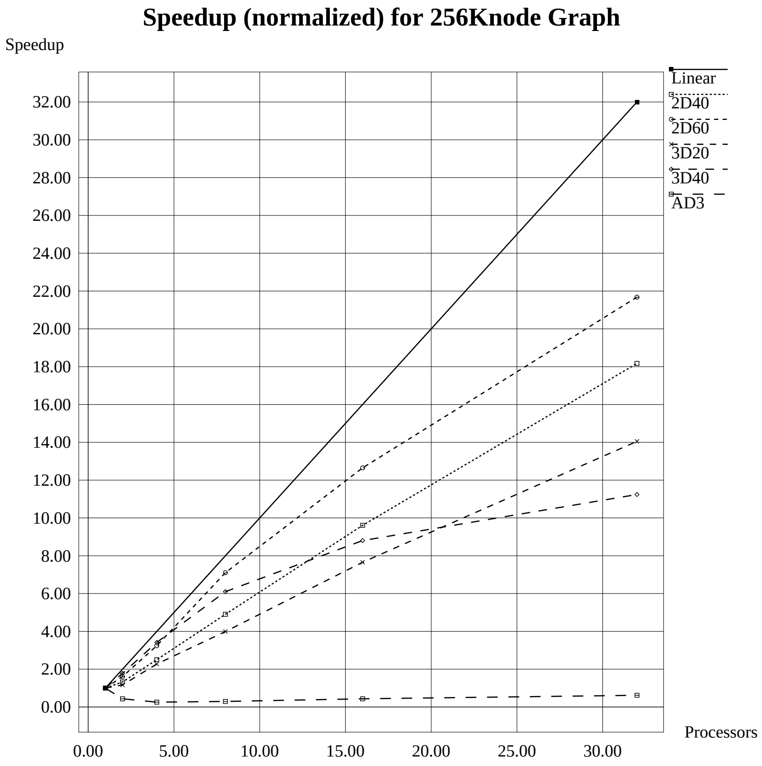

Figure 1: Speedup (problem size fixed at 256K nodes) for all graphs with processors varying from 1 to 32

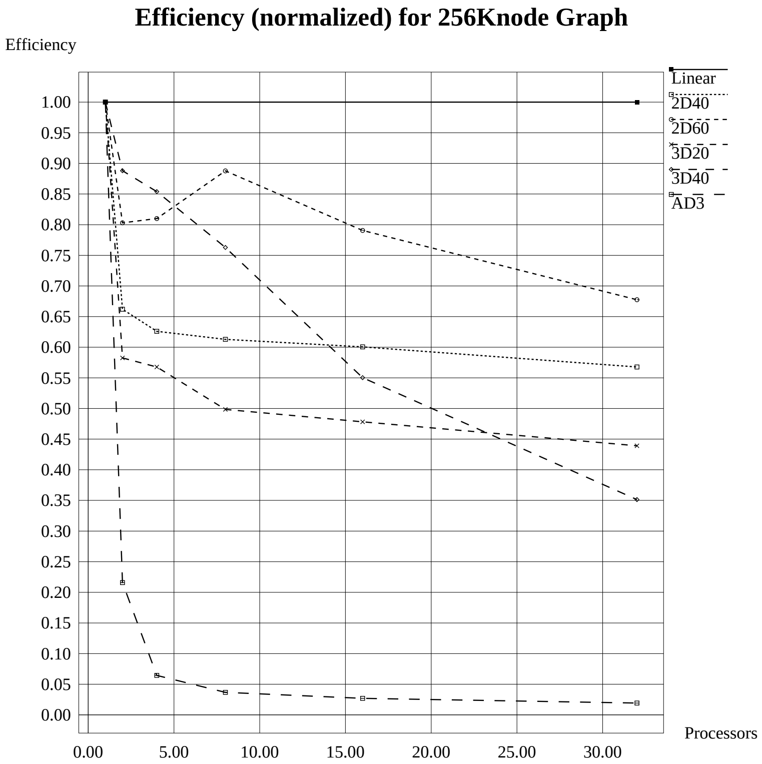

Figure 2: Efficiency (problem size fixed at 256K nodes) for all graphs with processors varying from 1 to 32

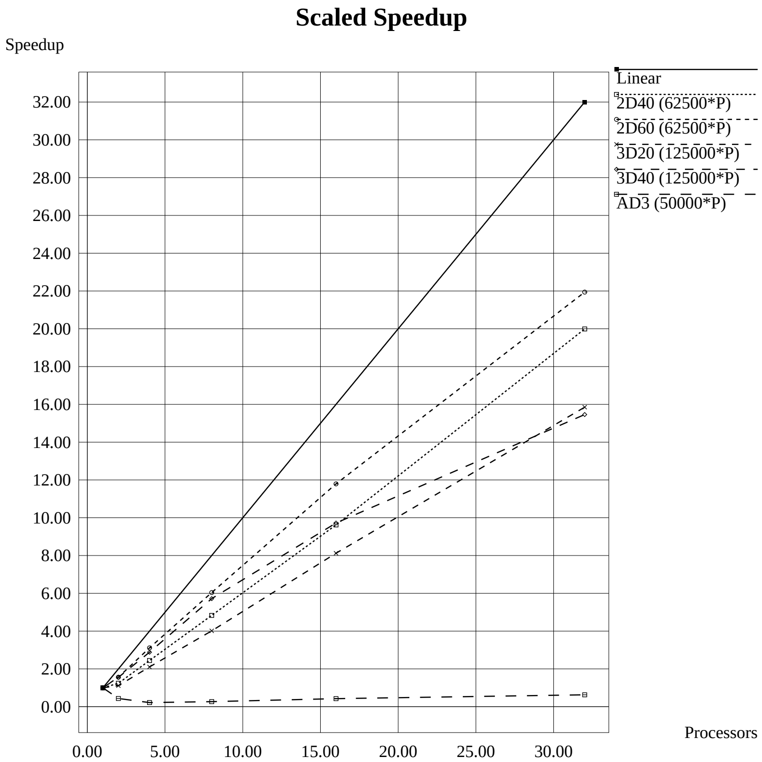

Figure 3: Scaled speedup (problem size per node fixed) for all graphs with processors varying from 1 to 32

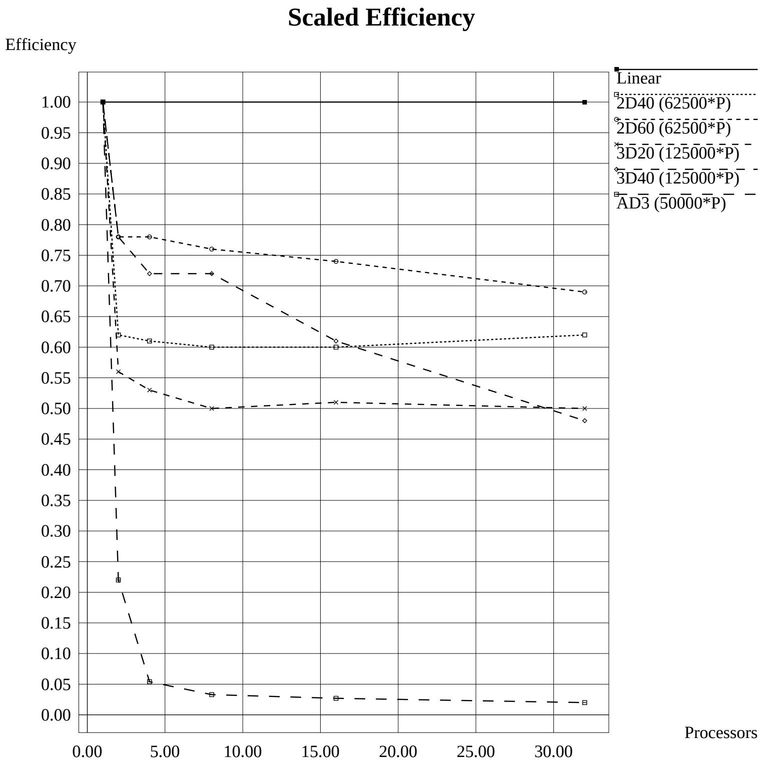

Figure 4: Scaled efficiency (problem size per node fixed) for all graphs with processors varying from 1 to 32

Before looking at the basic results of the algorithm, we must mention a few details of the measurement process. The graph creation section of the program allowed only for square sections and cubic sections on each processor for the 2D and 3D graphs, respectively. Since we wanted to compare execution times for any power of 2 processors, we decided to investigate the cost of processing each node and decide if we could take a close value and simply scale it linearly. The data in Table 2 show the results for 3D20 graphs. The cost does not vary more than about 4% over the range shown, and there is no clear trend, so we decided that scaling would be reasonable.

Further note that the single processor runs against which speedup is measured include only the time for the local phase of the algorithm. Although the global phase takes a significant amount of time even when using only one processor, that time is not relevant to the sequential execution time of the program.

Without further delay, we can examine the speedup for a fixed problem size (262,144 nodes) show in Figure 1. Ignoring AD3 for the moment, we see that 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 brings down the speedup. AD3 never regains a speedup of 1, but seems to be headed in that direction, arriving by extrapolation with around 64 processors.

Figure 2 shows the efficiency for the same data set. Again, we see that 2D40, 2D60, and 3D20 fall rapidly to a fairly level plateau, while 3D40 continues to decrease. AD3 falls precipitously from 1 processor to 4, then levels out.

A second definition of speedup is for a problem proportional to the number of processors. In Figure 3, 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 4, 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 (with the exception of 1 processor).

5.4 Results by Graph Type

Another interesting feature we have explored is how the connectivity of the graph affects the execution time and efficiency of the algorithm. In the 2D and 3D graphs, connectivity can be parametrized by the edge presence fraction. In Figure 5, we see the execution time varying as edge presence varies. The 2D graph runs had 264,992 nodes, but were scaled linearly down to 256K. The 3D graphs had 296,352 nodes and were scaled similarly.

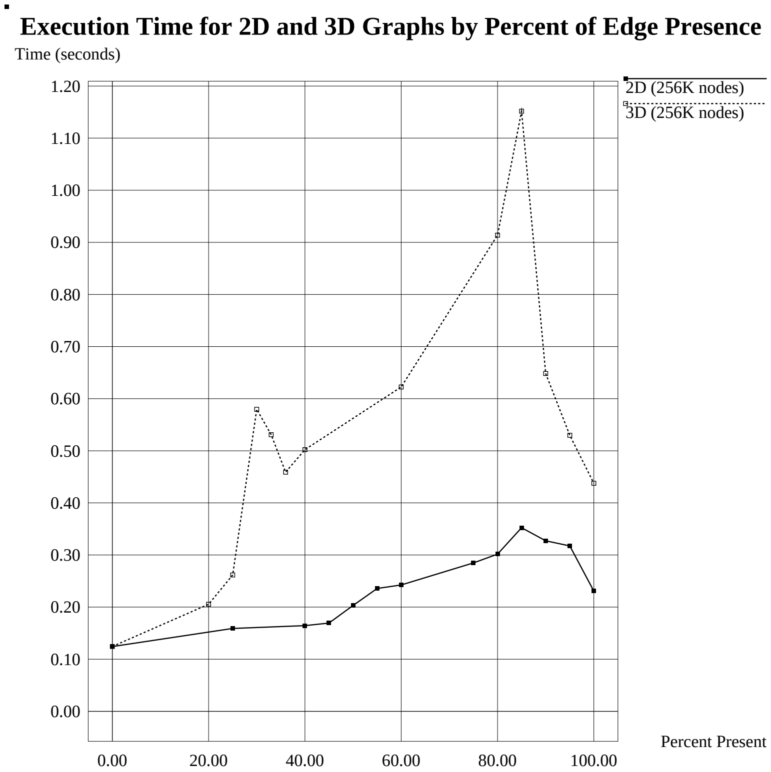

Figure 5: Execution time as a function of graph connectivity

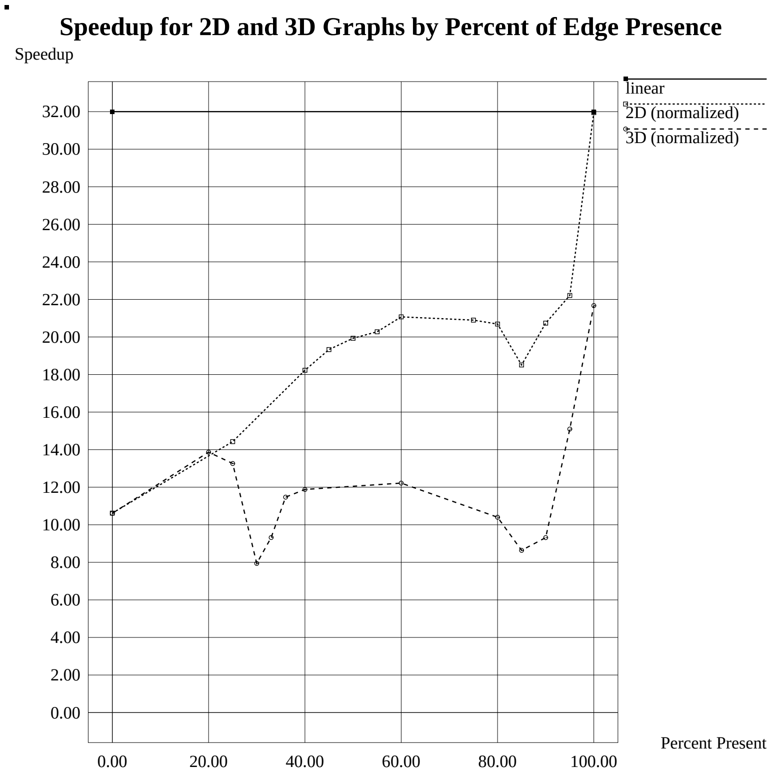

Figure 6: Speedup as a function of graph connectivity

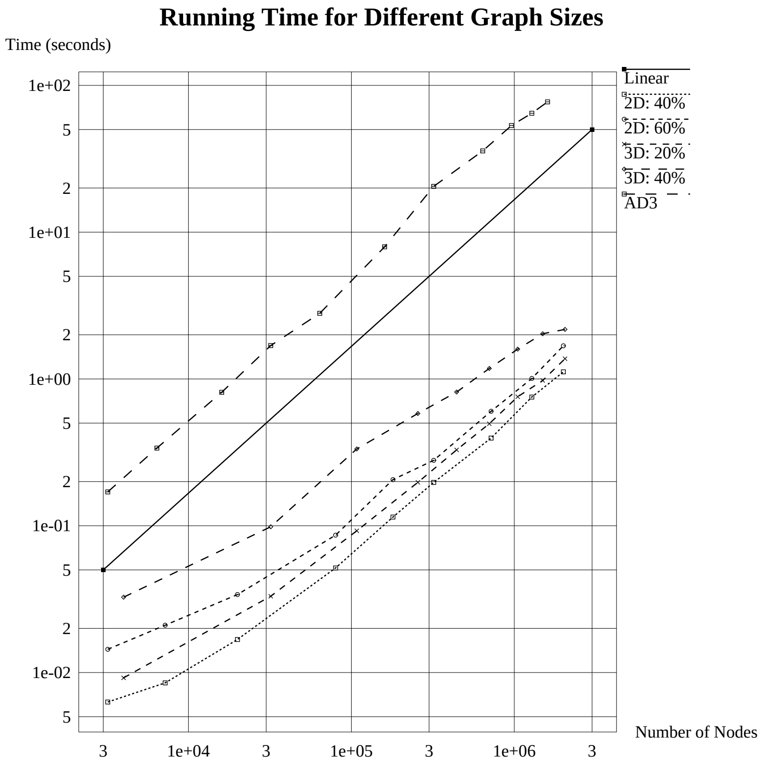

Figure 7: Execution time as a function of graph size

Both types of graphs exhibit a sharp increase in execution time near the boundary region where graphs change from mostly disconnected to mostly connected. They both peak around 85% edge presence and then fall off as more edges are added (probably duplicates). As expected, the graphs take nearly identical times when no edges are present.

Figure 6 shows the speedup attained by the algorithm across the range of graph connectivity for the 2D and 3D graphs. Again, the graphs have been normalized linearly to allow direct comparison with the 256K node graphs run on a single processor. We blame the random seed problem for the small but sharp fluctuations in the graphs. From the graphs we can see that speedup rises slowly up to the boundary region in each graph, where it begins a plateau until the edge percent grows quite high. At some point, the ability to spread the local work across the processors overcomes the diminishing cost of duplicate remote edges, and the speedup rises sharply at around 90 to 95% edge presence. Since a single processor could not fit a 3D, 256K node graph with more than 80% of the edges present, the higher values were obtained using a smaller graph (108K nodes).

5.5 Results by Graph Size

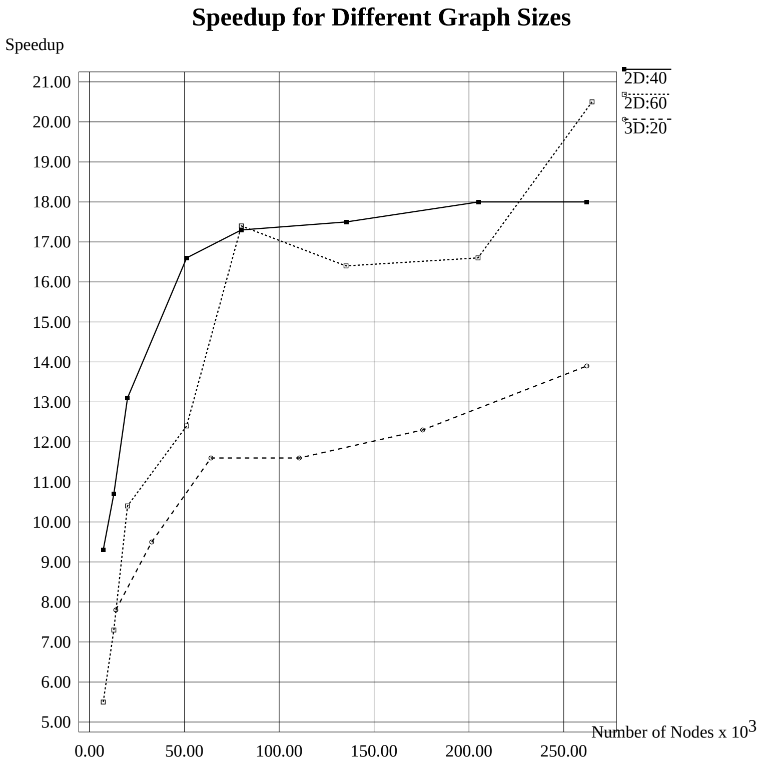

Figure 8: Speedup as a function of graph size

We now examine the characteristics of the algorithm as the problem size varies. In Figure 7, we find the execution time for each graph type as the number of nodes runs from 3,000 up to 2,000,000.

The time for the AD3 graphs is roughly linear and about an order of magnitude larger than any of the other graphs. The linear time is expected, since remote references dominate the cost of the algorithm, and the number of remote edges is proportional to the number of nodes in the AD3 graph.

The 2D and 3D graphs have a slope slightly below the linear slope, again as we expect. The fraction of remote edges decreases in each case as the number of nodes increases (the surface-to-volume ratio of the section of the graph owned by a processor increases with the number of nodes owned by that processor).

The speedup for the 2D graphs and 3D20 as we vary the number of nodes is shown in Figure 8. Unfortunately, the remaining time proved insufficient to gather the remaining data for this aspect of the algorithm. As expected, the speedup for very small graphs is quite low, but grows rapidly to a slowly increasing plateau for all of the graph types. The kinks and bends can be attributed to the problem with the random seed during single processor runs.

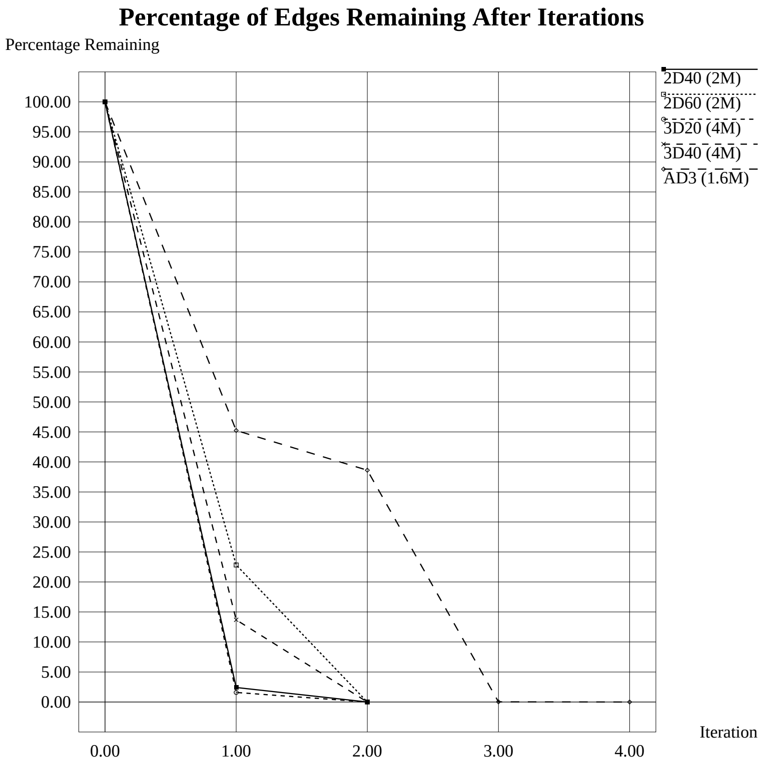

Figure 9: Edges remaining after each iteration (0 corresponds to completion of the local phase)

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 3: Number of Components after Each Iteration on an AD3 Graph

5.6 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 9, 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

Graph	Nodes	Iteration	Components	Unhooked	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 4: Fraction of Components Left Stagnant after an Iteration

the final number, and decreases rapidly in the first two iterations. For AD3 (with 1,600,000 nodes), the data appears in Table 3.

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.

5.7 Usefulness of Unconditional Hooking

Before moving on to compare our results with previous work, we will delve briefly into an investigation of whether or not unconditional hooking is worthwhile.

Table 4 shows the results for a large run of each graph type. The fraction of components left stagnant—those with edges remaining which are neither attached to another component nor become the parent of another component—during an iteration never amounts to even one percent with the 2D and 3D graphs, but can be quite large in AD3 graphs. Regardless, not identifying and hooking these stagnant components can lead to an increased number of iterations, thus costing more time in the end. We agree that unconditional hooking is worthwhile on our random graphs.

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.

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.

[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] 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

[7] Thinking Machines Corporation, "CMMD Reference Manual," Version 3.0, May 1993.