Towards Modeling the Performance of a Fast Connected Components Algorithm on Parallel Machines *

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

April 4, 1995

Abstract: We present and analyze a portable, high-performance algorithm for finding connected components on modern distributed memory multiprocessors. The graphs are probabilistic meshes used in cluster dynamics methods of computational physics. The algorithm is a hybrid of the classic DFS on the subgraph local to each processor and a variant of the Shiloach-Vishkin PRAM algorithm on the collection of subgraphs. On a 256-node Cray T3D, our algorithm demonstrates the fastest solution to date on graphs of practical importance. We analyze the performance characteristics of our solution across a range of graph types and machine performance parameters. The study sheds light on the performance impact of improvements in computational and/or communication performance on this challenging problem.

1 Introduction

The problem of finding the connected components of a graph has broad importance in both computer science and computational science. Computer vision, for example, makes extensive use of connected components algorithms for problems such as edge detection and object recognition. Use of connected components algorithms has also advanced the study of various physical properties of magnetic materials near critical temperatures, among other physical phenomena.

Connected components also appears to offer a unique challenge for parallel computing. Sequential solutions are well understood and commonly used in introductory computer science theory courses as an application of depth-first and breadth-first search. Parallel solutions have received a great deal of attention from both theorists and practical computer scientists, and have proven difficult. Theoretical work shows good results on the CRCW PRAM model[3, 10, 11, 24], which assumes uniform memory access time and arbitrary bandwidth to any memory location.

*This material is based upon work supported under a National Science Foundation Presidential Faculty Fellowship Award, a Graduate Research Fellowship, and Infrastructure Grant number CDA-8722788, as well as Lawrence Livermore National Laboratories Inst. for Scientific Research Grants #UCB-ERL-92/69 and #UCB-ERL-92/172. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of either organization.

The inherent contention in the algorithm has made even EREW solutions much more challenging [5, 14, 15, 17]. Practical application of the theoretical work to parallel machines has been largely restricted to small shared-memory machines and SIMD machines with very slow processors [11]. Many practical solutions have been developed for modern MIMD massively parallel platforms, or MPP's [6, 9, 13, 22, 18], and vector machines [8, 22]. The practical solutions typically emphasize performance over issues of portability, scalability, and generality, but still rarely obtain good performance.

In this paper, we present a portable, high-performance algorithm for finding the connected components of a general graph on distributed memory machines. On a 256-node T3D, our algorithm demonstrates the fastest solution to date on graphs of practical importance. This algorithm is a hybrid of the classic DFS on the subgraph local to each processor and a variant of the Shiloach-Vishkin PRAM algorithm on the collection of subgraphs. We analyze the performance characteristics of our algorithm across a range of graph types and modern parallel machines. This study sheds light on the performance impact of improvements in computational and/or communication performance on this challenging problem. These trade-offs are particularly important to quantify given that with microprocessor performance improving rapidly with time, greater computational performance can easily be obtained at the expense of communication performance by building less tightly integrated systems[1].

For our study, we use a class of graphs motivated by cluster dynamics problems in computational physics. Traditionally, numerical simulations of materials near critical temperatures utilize Monte Carlo algorithms with local methods of information propagation. The quality of statistics returned from these simulations depends heavily on the size of the sample, L. Although a larger sample size leads to better statistics, the number of steps needed to propagate information grows generally as $O(L^2)$ or worse using the traditional methods [4, 16, 21, 27, 28]. More recent methods, such as the Swendsen-Wang algorithm, reduce correlation time for the simulations. For a two-dimensional Ising model, the correlation time using S-W grows as $O(L^{0.35})$ [25, 26], allowing much larger samples to be studied.

At the heart of S-W is a connected components algorithm. The algorithm generates a random graph, finds the connected components of the graph, and applies a simple transformation to each component, then starts over [26]. In the years since the introduction of the S-W algorithm, cluster dynamics, and hence the use of connected components, has also found widespread use in other areas including computational field theory, experimental high energy and particle physics, and statistical mechanics.

The class of graphs arising in cluster dynamics is a random graph built on an underlying lattice structure, i.e., a *probabilistic mesh.* An instance of the graph is produced by using a single probability p to decide whether an edge in the underlying lattice is present in the graph instance. The decision for each edge is made independently of the decision for any other edge. Figure 1 illustrates possible instances of probabilistic meshes: 2Dp is built on a 2-dimensional lattice, and 3Dp is built on a 3-dimensional lattice.

The remainder of the paper is structured as follows: Section 2 describes the parallel algorithm,

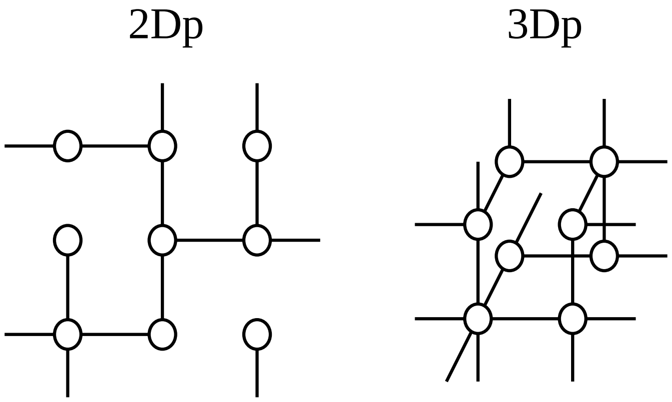


Figure 1: Probabilistic meshes used to study physical phenomena. Each edge in the underlying mesh is present in a given graph instance with probability p, independent of other edges.

Section 3 provides details the MPP's used in our study, Section 4 explains the application scaling rule used in the study, justifies the probabilistic analysis, and identifies the key performance issues inherent in scaling the problem across edge presence probabilities. Section 5 demonstrates and develops a model of the performance characteristics of the hybrid algorithm across the range of machines and graph types. Section 6 draws conclusions.

2 Algorithm

This section outlines the connected components algorithm and discusses the portable parallel language used in our implementation.

Obtaining good performance on a distributed memory machine demands that we pay attention to locality of access in order to minimize communication costs. For many algorithms, this results in a hybrid approach, a combination of local and global phases. During the local phases, an algorithm deals only with data that reside in a processor's local memory. In the global phases, the algorithm addresses the issues of efficient remote data access and synchronization between processors. The phases employ different algorithms and are optimized using different cost models. Performance generally depends on the balance of time spent in the two phases, and is usually good if the local phase dominates the total time. For some algorithms, this balance can imply doing extra local work to reduce the complexity of the global phase. As a global algorithm grows in complexity, understanding and avoiding load imbalance and network contention becomes much more difficult.

We followed a strategy of hybridization to create our algorithm, merging a simple and fast local algorithm (depth-first search, or DFS) with a powerful and efficient global algorithm (Shiloach-Vishkin [24], or S-V). Neither the local nor the global algorithm is sufficient alone. We might try to extend DFS with a task queue, assigning a new tree in the depth-first forest whenever a processor needs more work, but the complete ignorance of data locality inherent in this method proves fatal to performance. S-V, designed with a CRCW PRAM in mind, suffers from nearly the same problem—the PRAM model fails to recognize the non-uniform access times of distributed memory machines, resulting in poor performance, as we demonstrate later. By merging the local DFS with the global S-V and then optimizing the result, we create an algorithm that performs and scales well on an interesting set of graphs. Compared with purely global approaches, the algorithm performs an order of magnitude better.

The optimized algorithm follows. For more detail on the data structures or on the process of optimization, see [19].

1. **Local Phase.** Perform a local depth-first search (DFS) on each processor's portion of the graph, collapsing each local connected component into a representative node. The local phase results in a much smaller graph for the global phase. Each component of this global graph is marked with a unique value for the global phase.
2. **Global Initialization.** Finish the preparation of the global graph by pointing each remote edge at one of the local representative nodes selected in the local phase.
3. **Global Iterations.** Beginning with the reduced global graph of representative nodes and remote edges, iterate over the following steps.
   a. *Termination Check.* If no components remain to be processed on any processor, quit.
   b. *Hooking Step.* Merge components into larger components by attempting to link each component across a shared edge. Conditions on attachment guarantee that no cycles are formed.
   c. *Star Formation Step.* Collapse newly formed components into trees of height one (stars) to ensure that representative nodes in a component have a single, consistent value.
   e. *Self-Loop Removal Step.* For each component, remove edges that point to nodes within the component, *i.e.* nodes with the same value.
   f. *Edge-List Concatenation Step.* For all leaf nodes of components, move edges to the root.
4. **Local Cleanup.** For each node not selected to be a representative node in the global graph, update the value of the node from its representative.

The implementation difficulty of a distributed memory algorithm occurs almost completely in the global phases. Here, we must devise methods to access remote data as efficiently as possible while maintaining correctness and minimizing synchronization costs. Optimization of the global phases is essential for good performance, but requires that the language provide an easy to understand cost model for parallel operations. Also, this model must be stable across a variety of platforms to make the algorithm truly portable—if the model changes dramatically on a new machine, we must optimize independently for that machine.

The Split-C language [7] provides such a cost model, giving us a set of simple yet effective abstractions for programming a parallel machine and allowing us to implement the algorithm in a straightforward fashion and to optimize the global phase once for all platforms. In particular,

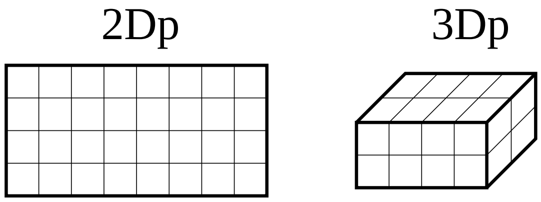


Figure 2: Natural partitioning of probabilistic meshes. The underlying lattice is cut into equal-size blocks for the processors.

Split-C provides a global address space: 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 is used just like a normal pointer, but can reference the entire global address space; standard pointers reference only the portion local to a processor. Just as pointers help to build data abstractions in C, distributed data structures in Split-C are realized by laying the data across the global address space and using global pointers to hook it together. The edges of a distributed graph, for example, are merely global pointers from one distributed vertex to another. Global pointers also allow the programmer to determine the owner of the referenced data without actually dereferencing the pointer. We use this ability in the local phase to differentiate between local edges, which must be collapsed into single nodes of the graph for the global phase, and remote edges, which must be retained as edges in the global graph. The clear distinction between local and global objects helps us to apply the cost model for optimization.

Using Split-C also gives our implementation portability. Versions of Split-C exist on the Cray T3D, the IBM SP-1 and SP-2, the Intel Paragon, the Thinking Machines Corp. CM-5, the Meiko CS-2, and networks of workstations [2, 20, 23, 29].

Although our algorithm accepts arbitrary graphs as input, obtaining optimal performance requires a reasonable partitioning of the graph across processors to enhance locality and load balancing. Partitioning techniques rely on the ability to determine properties of the graph structure. Fortunately, probabilistic meshes offer a natural geometric partitioning via the underlying lattice structures, and we exploit this partitioning for high performance. As shown in Figure 2, we simply cut the underlying lattice into equal-size sections, one for each processor. The effectiveness of this simple partitioning is discussed at length later in the paper.

In summary, we have implemented a portable hybrid algorithm for finding the connected components of a general graph. Our implementation scales reasonably and performs efficiently on a class of graphs of interest to the scientific community.

3 Machines

In our study, we consider three large-scale parallel machines: the Cray T3D, the Meiko CS-2, and the Thinking Machines Corp. CM-5.[1] These machines offer a range of computational and communication performance against which to evaluate the algorithm implementation. In each case, the Split-C compiler is built as an extension of GCC version 2.4.5, which provides the local code generation. The code generated for global access is fully optimized to exploit the specific communication hardware capabilities. The T3D is based on the DEC Alpha 21064, a 64-bit, dual-issue, second generation RISC processor, clocked at 150 MHz, with 8 kB split instruction and data caches. A Split-C global read involves a short instruction sequence to gain addressability to the remote node and to load from remote memory, taking approximately 1 μs[2]. The CS-2 is based on a 90 MHz, dual-issue Sparc microprocessor with a large cache. Communication is supported by a dedicated "ELAN" processor within the network interface, which can access remote memory via word-by-word or DMA network transactions. The Split-C global read issues a command to the communications co-processor via an exchange instruction, which causes a waiting ELAN thread to either access remote memory or to begin a DMA transfer, depending on length. A remote read requires roughly 20 μs[23, 29]. The CM-5 is based on the Cypress Sparc microprocessor, clocked at 33 MHz, with a 64 kB unified instruction and data cache. A Split-C global read involves issuing a CMAML active message to access the remote location and reply with the value, taking approximately 12 μs. Traditional measures of the computational performance of the node, such as MFLOPS and SPECMARKS, offer little indication of the performance on this graph algorithm, which stresses the storage hierachy, so we will calibrate the local node performance empirically in Section 5.

4 Application Scaling

The bottom-line performance of our algorithm is shown in Figure 1 for four important graph types. The 2D40/200 and 2D60/200 graphs are rectangular, two-dimensional probabilistic meshes with 40% and 60% edge probability, respectively, and a 200x200 subgraph per processor. Similarly, 3D20/30 and 3D40/30 are three-dimensional probabilistic meshes with 20% and 40% edge probability, respectively, and a 30x30x30 subgraph per processor. The importance of these particular graphs is discussed below. Fixed problem size per node scaling is used, so the T3D is solving a larger problem than the CM-5 or CS-2. However, the problem size is not extremely large, so that the problems can reasonably be run on a single node. The use of larger graphs would only make obtaining high performance easier. Also shown in the table is the best published uniprocessor performance, obtained on a Cray C-90, for comparable graphs.[2] The C90 performance rate is essentially independent of the problem size as long as it fits in memory. Previously, Flanigan

[1] In the final version of the paper we intend to include the IBM SP-1/2, Intel Paragon with Active Messages, and a high-performance network of workstations. The Split-C language is currently available on these platforms, and the connected components algorithm is running, but at the time of writing this draft, the data collection is not yet complete.

[2] The use of a lower density graph in the 2D/30 case makes the comparison somewhat optimistic in favor of the C90.

Graph Type	T3D/256 (Mn/s)	CS-2/32 (Mn/s)	CM-5/64 (Mn/s)	C90/1 (Greiner) (Mn/s)
2D30/*				4.3
2D40/200	130	3.39	4.66	
2D60/200	70.5	1.57	3.42	2.5
3D20/30	101	1.32	2.47	4.2
3D40/30	16.3	0.335	0.780	2.1

Table 1: Raw performance of the parallel connected components algorithm. The values in the table are in millions of nodes processed per second using the hybrid algorithm described in the text. The rightmost column shows millions of nodes processed per second on a single node of a Cray C90 using an algorithm developed by Greiner.

and Tamayo achieved 12.5 million nodes per second on much larger 2D graphs using a 256-node CM-5 [9]. Also, Hackl et. al. achieved nearly 7 Mn/s on 32 processors of an Intel IPSC/860, again on a very large 2D graph [12]. Neither of the results can be compared directly, since they were applied to a particular problem and do not specifically state the edge presence probability for the graphs.

While the bottom-line performance results are pleasant, our goal is to understand in depth the performance characteristics of our hybrid algorithm, the fundamental bottlenecks it presents, and how its performance is affected by machine characteristics. To this end, we need to examine a range of problem sizes and graph structures across the machines. In addition, we are working with a class of random graphs, not a specific problem instance. In this section, we sketch the key aspects of the connected components problem that influence algorithm performance and validate that probabilistic analysis of a sample of graphs is well-behaved.

4.1 Equal nodes-per-processor scaling

One expects the total work in solving connected components to be a function of the number of nodes V, the number of edges E, and the number of components C in the graph. For the class of graphs we are considering, the expected number of edges is proportional to the number of nodes: $\bar{E} = pV$, where p is the edge presence probability. The sampling of random graphs in Figure 3 indicates that for a given edge probability, the number of components in the graph is also proportional to the graph size. By scaling the number of nodes linearly with the number of processors, we scale up all aspects of the problem uniformly.

Figure 3 also suggests that the ratio of components to graph nodes varies significantly with edge probability, since the slopes of the lines differ. Fortunately, since the number of components per node is largely independent of the graph size, we can examine how this varies with edge probability. Figure 4 shows the expected number of components per node for 2D and 3D graphs over a range of edge presence probability. The curves are normalized from graphs with 1.28 and 0.864 million nodes.

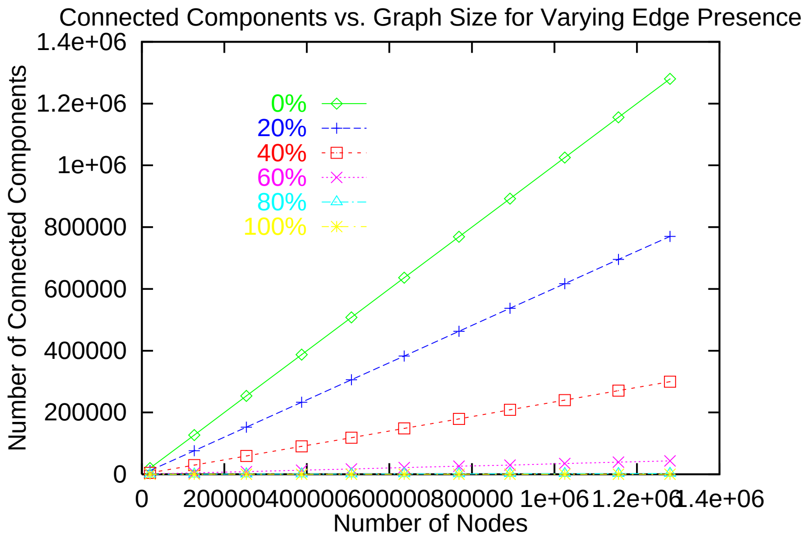


Figure 3: Expected number of connected components for 2D graphs. For a given edge presence probability and underlying lattice, the expected number of components is linear in the number of nodes.

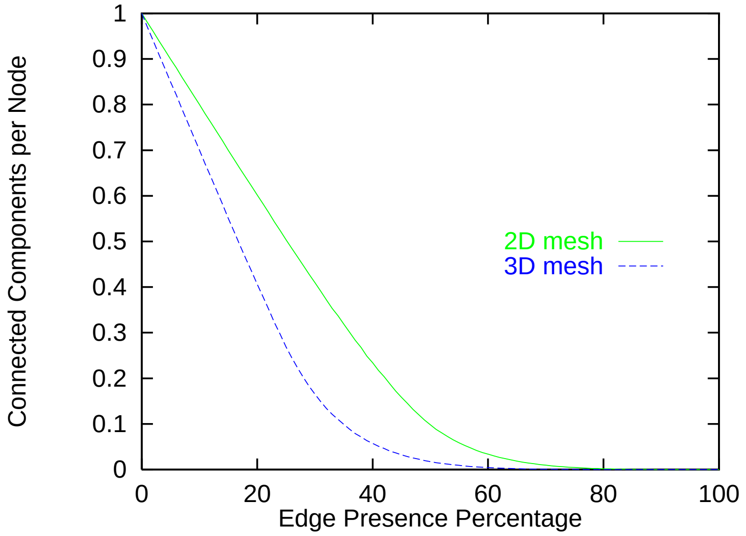


Figure 4: Expected number of connected components per node. As graph size increases the fractional variance decreases, making these functions very accurate for large graphs.

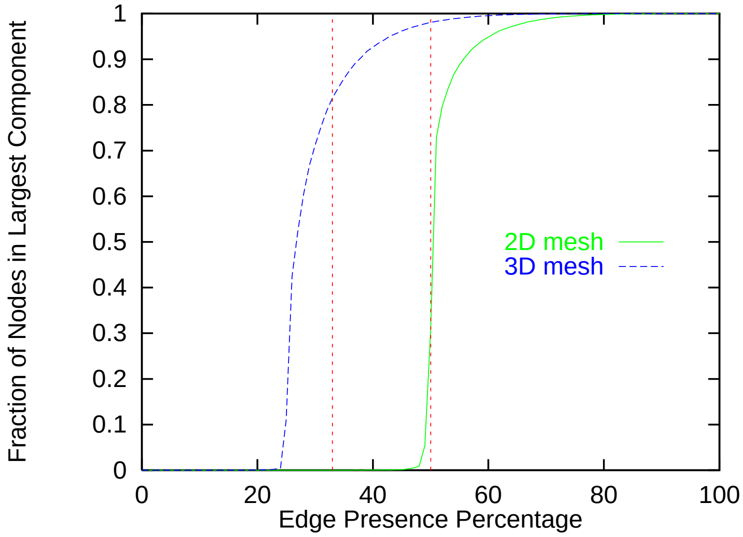


Figure 5: Size of the largest connected component in a graph. Note the rapid transition in both 2 and 3 dimensions.

4.2 Graph phase transition

Observe that components per node shown in Figure 4 is highly non-linear. This behavior is a result of a very interesting phenomenon, which is critical to understanding the performance exhibited by connected component algorithms. As the edge presence probability crosses a boundary where the nodes have an average degree of two, 50% for 2D graphs, and 33% for 3D graphs, a phase transition occurs between mostly unconnected and mostly connected graphs. Below the boundary, the graph consists of a large number of fairly regular sized components. Above the boundary, the graph consists of a single, very large component and a number of tiny components. Figure 5 shows the fraction of nodes in the largest component for 2D and 3D probabilistic meshes as a function of the edge presence probability. The vertical bars mark the theoretical boundaries for the two graphs, the point where each node is expected to connect to two other nodes.

To illustrate the phase transition graphically, Figure 6 shows the largest three components in 2D graphs with edge percentages below, on, and above the boundary. The following table gives the number of nodes in each of the components shown:

Rank	40%	50%	60%
largest	281	38,884	62,174
2nd largest	219	2,064	19
3rd largest	119	177	6

For mostly connected graphs, we must expect that the performance of the algorithm will be strongly influenced by the presence of this single large component. Although the phase transition makes it difficult to compare across graph types, we still have a good scaling rule for any particular graph type of interest. In the next section, we use the nodes-per-second metric to help understand

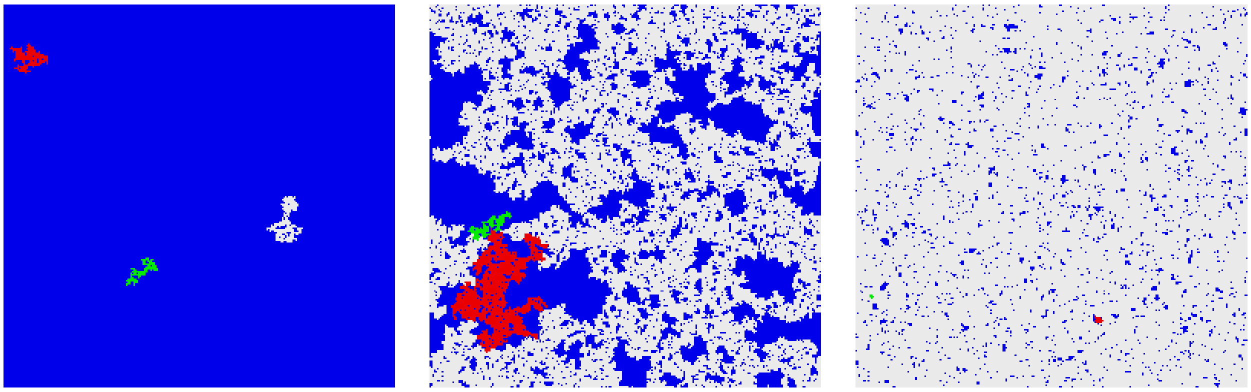


Figure 6: Largest connected components for several 256x256 2D graphs. From left to right, the graphs have edge presences percentages of 40, 50, and 60. The white component is the largest in the graph, followed by the red and then the green. The blue sections are made up of smaller components.

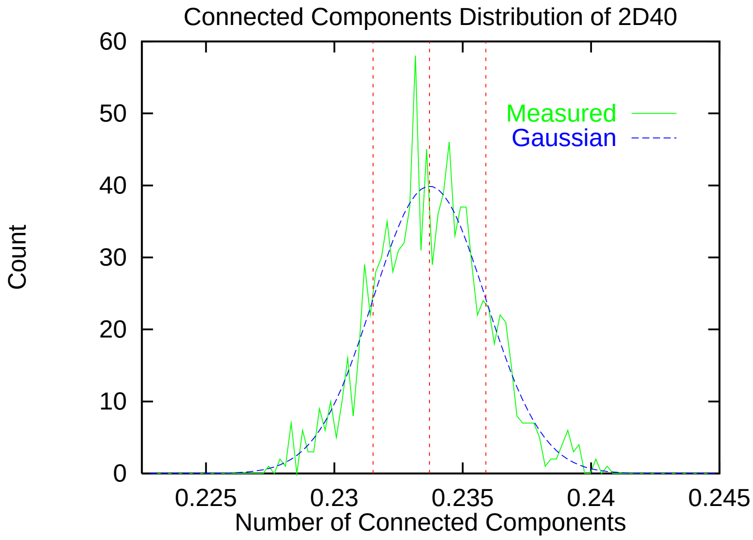


Figure 7: Distribution of number of connected components. Note how closely the distribution of 1,000 samples from a 256x256 2D40 graph matches a Gaussian distribution with the same mean and standard deviation. The vertical lines mark the mean and one standard deviation in either direction.

the performance in the local and global phases of the algorithm for each graph type.

Naturally, we might question whether the expected number of components is statistically well-behaved—it is certainly not obvious that a complex, discrete operation like forming connected components should yield a nice distribution, even on random graphs. We find, however, that the expected number of connected components per node can indeed be treated as a normal random variable. Consider Figure 7, which shows the distribution of number of connected components found in 1,000 2D40 graphs with 65,536 nodes. The average number of components per node is 0.234, just as we saw in Figure 3, and the standard deviation is 0.00220. The samples are binned into tenths of the standard deviation. The figure also shows a Gaussian distribution with the same mean and standard deviation to allow for easy comparison between the two.

The second implication of Figure 7 is that when a mostly unconnected graph is partitioned into a large number of subgraphs of similar structure, a few of the subgraphs are very likely to have a number of components that is much larger than the average, causing a fundamental load balancing problem.

4.3 Graph selection

In order to study the performance of our algorithm, we must select the specific graphs to study. The discussion above suggests that two important characteristics require attention in this process: the phase transition in graph structure and the surface-to-volume ratio. The first because the algorithm will act differently with mostly connected and mostly unconnected graphs, and the second because the global phase of the algorithm is fundamentally influenced by the cost of remote access.

The execution time for the local phase of the algorithm is expected to be proportional to the number of nodes in the local portion of the graph. We can assume that the cost of the global phase is monotonically increasing in the number of nodes on the boundary of the local portion—as the size of the boundary increases, the number of remote edges increases, and processing time will rise. The cost model for distributed memory machines tells us that the work done locally can be done much more rapidly than the work done remotely. Putting these facts together, we recognize that as the surface-to-volume ratio increases, performance decreases. This realization is equivalent to saying that we cannot achieve perfect speedup on this problem. An interesting exploration of the probabilistic meshes requires that we consider at least two distinct surface-to-volume ratios to demonstrate the performance concepts.

Interesting values for the surface-to-volume ratio characteristic can be obtained by simply examining both 2D and 3D graphs—a 3D cube with the same number of nodes as a 2D square has a much larger surface-to-volume ratio. We choose to examine scaled speedup since it allows us to fix the surface-to-volume ratio for a graph and thereby simplifies the analysis. Exploration of the phase transition requires that we examine graphs in both phases for each underlying lattice, giving us a total of four graphs to measure. By the argument given above, we can obtain the best results by using the largest graphs that can fit into memory, but we choose instead to use a modest amount of memory, as shown in the table below describing our four graphs:

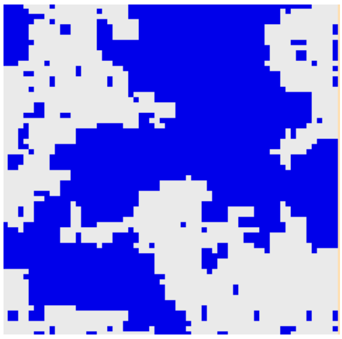


Figure 8: A possible local portion of a 2D50 graph. The white areas are parts of the same component, but only connect at great distance from the processor.

Name	Underlying Lattice	Edge Presence	Nodes per Processor	Surface-to-Volume Ratio	Memory per Processor
2D40/200	2D	40%	200x200 = 40,000	1.99%	3.3 MB
2D60/200	2D	60%	200x200 = 40,000	1.99%	4.3 MB
3D20/30	3D	20%	30x30x30 = 27,000	18.7%	1.9 MB
3D40/30	3D	40%	30x30x30 = 27,000	18.7%	2.9 MB

It turns out that many of the most expensive graphs on which to compute connected components are those very near the edge presence phase transition boundary. The reason is illustrated by Figure 8, which shows a local section of a 2D50 graph. One processor might have several local components which are connected way out in the remote parts of the graph. However, as the behavior the the graphs is fairly chaotic near this boundary, with large variations in the size of the largest component, we do not include these graph types in the study.

5 Performance and Models

In this section, we present results and analyze the algorithm, building on our notions of scaling. We begin with a discussion of the local phase, illustrating how the phase transition affects even the depth-first search. We then briefly discuss alternatives to the hybrid approach and demonstrate their deficiencies. Finally, we examine the balance between the local and global phases of the algorithm for the graph types we selected in the previous section and explain how this balance can be used to understand the speedup results.

5.1 Modeling the Local Phase

As a first step, we compare the computational capabilities of the machines. Table 2 shows the processing power of a single node on each of our platforms. The second column in the table gives

Graph Type	C90 (Greiner) (Mn/s)	T3D (Mn/s)	CS-2 (Mn/s)	CM-5 (Mn/s)
2D30/*	4.3			
2D40/200		0.781	0.354	0.223
2D60/200	2.5	0.574	0.237	0.147
3D20/30	4.2	0.885	0.375	0.239
3D40/30	2.1	0.540	0.216	0.133

Table 2: Single node connected components performance. The values in the table are in millions of nodes processed per second using an optimized depth-first search algorithm. For comparison, the second column shows millions of nodes processed per second on a Cray C90, as reported by Greiner.

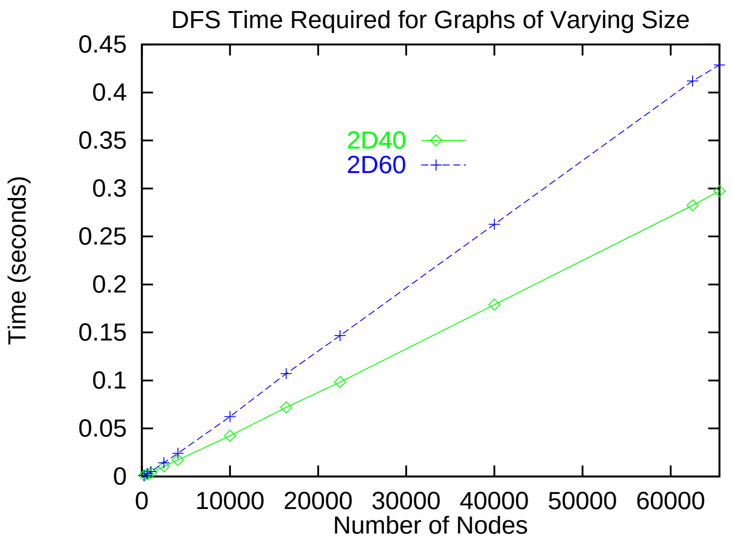


Figure 9: Time required for DFS of a graph on a CM-5 Sparc node. Execution time is linear in the size of the graph.

processing speeds for a single node of one of today's most powerful sequential supercomputers, the Cray C90, as reported by Greiner in [11].

Consider the relative performance of a T3D and a CM-5. The cycle time of the Alpha node in a T3D is about 4.5 times faster than that of the Sparc in a CM-5; on the other hand, the memory access latency is only about 3 times as fast. We see from Table 2 that the performance of a T3D node ranges between 3.5 and 4.1 times better than the performance of a CM-5 node, depending on the type of graph. We begin to understand what characteristics of the machine dominate the behavior of connected components algorithms.

As we saw earlier, the connected components performance for a given graph can be expressed as a function of the number of nodes, since the expected number of edges and connected components in a probabilistic mesh are also functions of the number of nodes. In fact, for the local phase, the function is linear in the size of the graph, as shown in Figure 9. In the figure, we see the execution

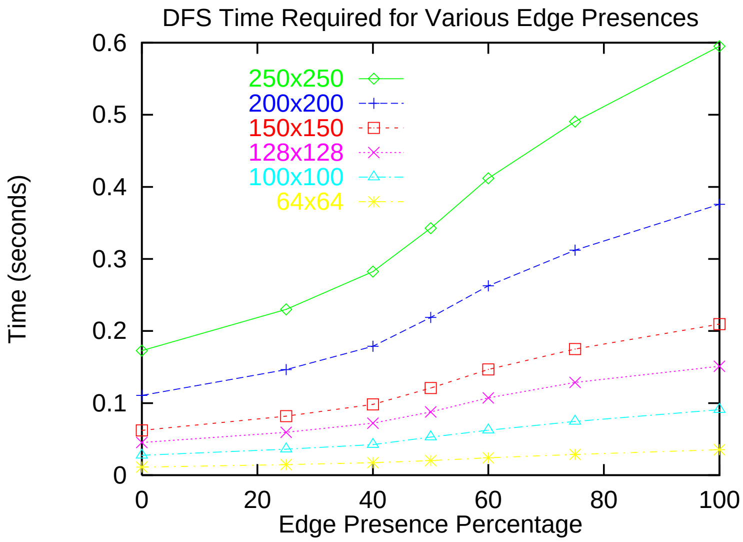


Figure 10: Time required for DFS of a graph on a CM-5 Sparc node. We expect the time to rise linearly in the edge presence probability p, but see some strange behavior around the 50% mark.

time of the local phase for two graph types of various size. The linear behavior makes the nodes-per-second metric meaningful for the local phase regardless of problem size. Clearly, this will not be true for the parallel algorithm, in which the graph size has direct impact on the performance through the surface-to-volume ratio.

Although there is some information in the single-node performance numbers, we cannot use them to make very accurate predictions for other graph types or other machines.[3] We might consider measuring independent processing rates for nodes and edges to allow us to characterize all probabilistic meshes, but the phase transition in graph structure breaks such a model, as shown in Figure 10. As the figure shows, the execution time for 2D graphs rises more rapidly around the phase transition boundary at 50%. The graphs in the figure are large enough that cache behavior is not a problem unless the graph structure changes significantly. The phase transition does exactly that, however, causing the algorithm to behave much differently since a single DFS covers only a small portion of the graph in the less connected phase, which we might think of as the 'liquid' phase, but covers nearly the entire graph in the more connected, or 'solid' phase. The algorithmic differences lead to differing cache behavior, which in turn explains the rise in execution time around the phase transition boundary. Nevertheless, the nodes-per-second metric does provide scalable information on the local phase for any graph type, and we can use this information to understand how the local and global phases interact and to predict speedup.

[3] The CM-5 and CS-2 performance ratings in the table are fairly similar: with the numbers for a CS-2 and one number from a CM-5, we can predict the other CM-5 numbers to within 4% of their actual values. Remember that both machines are based on the SPARC architecture, however, so we expect the numbers to be similar. Predictions about a CM-5 or a CS-2 from a T3D result in errors of up to 16% for either machine.

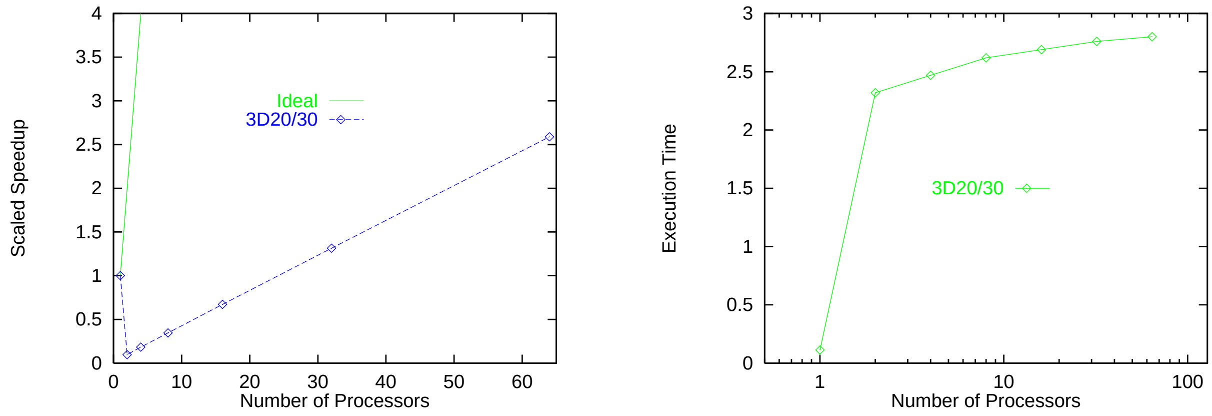


Figure 11: CM-5 execution time for a purely global algorithm on 3D20/30. Each processor has 27,000 nodes. The time for 1 processor is based on the local phase of the algorithm, and illustrates the large overheads incurred by skipping the local phase on parallel executions.

5.2 Comparing Other Solutions

Before continuing with the modeling process, we stop for a moment and ask again: why use a hybrid algorithm? An algorithm that assumes uniform access time to the entire global address space is certainly easier to write, and arguably easier to understand as well. But can such an algorithm achieve good performance? According to the cost model supported by Split-C for distributed memory machines, the answer is that it cannot. Ignoring data locality proves fatal to performance. By skipping the local phase of the algorithm and using only the S-V implementation to find the connected components, we are able to illustrate this concept in Figure 11. The figure shows the CM-5 execution time of the purely global algorithm on a 3D20/30 graph. Although the cost of the global phase asymptotically approaches a constant value, resulting in a constant fraction of ideal speedup, the fact remains that the parallel algorithm requires 24 processors to find components of the same graph as does a single processor using a sequential algorithm. Results for 2D40/200 are slightly better, requiring only 20 processors to achieve the uniprocessor figure of merit. Highly connected graphs suffer from a phenomenon associated with the phase transition, as we discuss in Section 5.3, and can never achieve the speed of a single processor.

A second alternative is for the programmer to use a hybrid algorithm, but to discard the sequential programming abstractions when writing the global phase of the algorithm. Typically, this approach is often taken when the overhead associated with the communication layers used is high, as with [9][12]. The global phases of these algorithms build new programming abstractions to allow for bulk communication between processors. The added complexity of design often results in worse performance, and forces the programmers to rewrite the implementation (if not the algorithm) to accept anything but a specific type of graph. Both [9] and [12], for example, work only with 2D graphs. Conversely, our hybrid implementation uses many of the sequential programming abstractions and can accept arbitrary graphs. Our algorithm performs well provided that the graph can be partitioned well between processors, as can the graphs of interest in many areas of science.

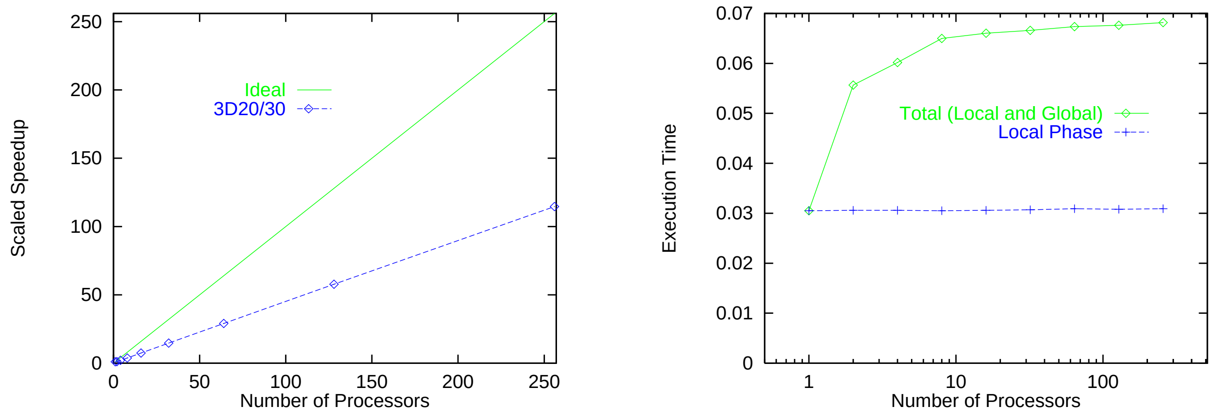


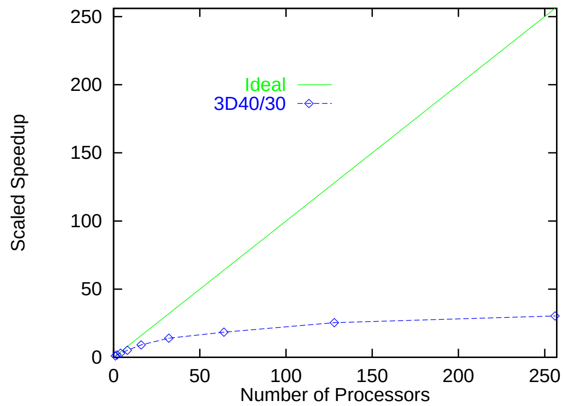
Figure 12: Execution time for 3D20/30 on a T3D. The local cost remains nearly constant as the number of processors increases. The global cost for a graph in the disconnected phase quickly reaches a plateau, indicating that speedup is close to a constant fraction of ideal speedup.

5.3 Modeling the Global Phase

Consider the work done in the global phase of the algorithm. After finding the connected components in its local portion, each processor must examine its remote edges and process them. If we assume that most of the remote interactions are with neighboring processors and are fairly well load-balanced, then the global phase requires a constant amount of time, independent of the number of processors. The balance between the local and global phases is then constant as well, giving us a constant fraction of ideal speedup, with the fractional value depending only on the relative cost of the two phases. For graphs in the disconnected phase, the approximation is reasonable. The number of remote edges is a well-behaved random variable. Most distributed components are split across only two processors; a handful might be spread over four (or even eight in the 3D graphs); but very rarely will any component cross the breadth of a processor's local portion and link together multiple processors. The global cost in this case consists primarily of trading information with neighbors, a cost which should not rise as the number of processors increases. Our hypothesis is verified by the data in Figure 12, which shows the local and global execution times for 3D20/30 for a T3D.

As we expect, the nearly constant execution time for the global phase results in a nearly constant fraction of ideal speedup as we increase the number of processors. The exact fraction we obtain depends on the relative cost of the local and global phases. We expect this in turn to depend on the surface-to-volume ratio, since a relatively smaller number of nodes on the boundary of the local portion should result in a smaller global execution time relative to the local time. This is in fact the case: 2D40/200, with a surface-to-volume ratio of 1.99%, achieves a speedup of 0.508 per processor, while 3D20/30 with a surface-to-volume ratio of 18.7%, achieves somewhat less, 0.395 per processor.

Although the global execution time is nearly constant for large numbers of processors, small numbers of processors exhibit a clear upward trend. This trend can be explained with a straight-



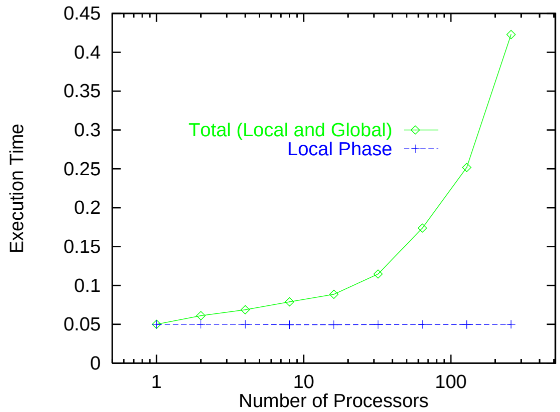


Figure 13: Execution time for 3D40/30 on a T3D. The global cost for a graph in the connected phase rises almost linearly in the number of processors, indicating nearly flat speedup for large numbers of processors.

forward statistical analysis. Consider a normal distribution for a random variable such as that shown in Figure 7 for the distribution of number of connected components in a graph. The value of a single variable from the distribution will most likely end up within a standard deviation of average, and almost surely within two standard deviations. The hybrid algorithm, however, works in bulk synchronous phases: each processor does some work, and then they all synchronize. The barrier synchronization effectively applies a MAX function to the cost of any step. Naturally, as we select an increasing number of independent variables from the normal distribution, we expect an increasing maximum value over the set of variables. In fact, we expect the maximum to rise fairly quickly at first, but then to taper off, just as does the global execution cost in Figure 12.

Having established the foundations of a model for the disconnected phase, we now consider performance on graphs in the connected phase. Figure 13 breaks down the local and global execution times for 3D40/30 on a T3D. Unlike the graphs in the disconnected phase, the global cost for 3D40/30 grows almost linearly in the number of processors. Clearly, the phase transition into the connected phase has broken our intuitive model of the disconnected phase. In the connected phase, the graph is dominated by a single, very large component. Only a single processor can 'own' the big component; each other processor attaches its own single large component to the main component and sends unresolved edges to the owner. The number of remote edge structures explored by the owner of the large component thus grows nearly linearly in the number of processors. Since the algorithm operates in a bulk synchronous fashion, all other processors must sit idle while a single processor handles the bulk of the work, as shown in Figure 14.

In addition to the load imbalance caused by the single component phenomenon in the connected phase, we also note a high degree of contention around the end of the algorithm. After the large component has completely formed, another global iteration is required to remove all of the connected representatives from the processing list. During this iteration, all processors must access the large component to check for remaining edges. On a CM-5, we observe an average time of 100 μs for each remote access during the termination check. Recall that the uncongested remote access time

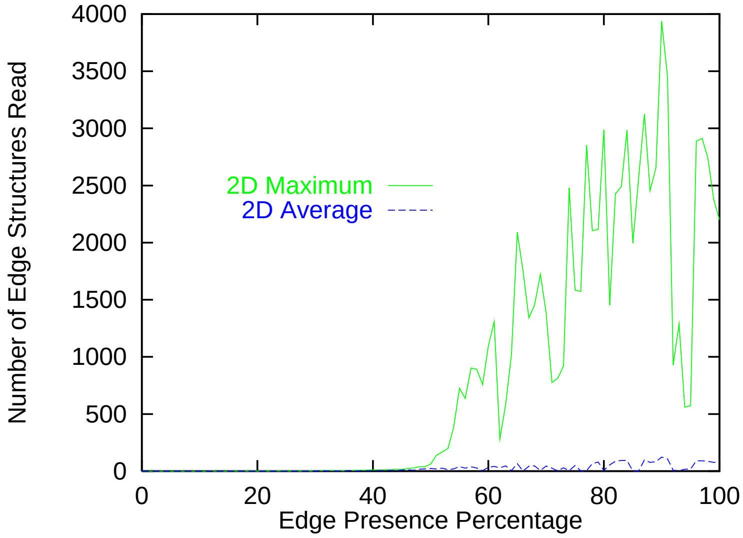


Figure 14: Load imbalance shown for 2D graphs. Once past the phase transition boundary, the maximum number of edge structures read by a processor grows rapidly, though chaotically, while the average number remains much smaller. The same phenomenon occurs, even more dramatically, for 3D graphs.

is only 12 μs, indicating a more than eightfold increase. A policy of caching could greatly improve the algorithm in this regard, but more effort is required to solve the load balancing problem.

Just as we saw with the graphs in the disconnected phase, graphs in the connected phase obey the same relationship between surface-to-volume ratio and performance. The 3D40/30 speedup on a T3D has nearly flattened out at 30.3 for 256 processors. For 2D60/200, however, the speedup is still increasing fairly quickly when it reaches 115 at 256 processors. We will observe the actual plateaus in the CM-5 and CS-2 results; both machines have relatively slow communication compared with a T3D.

Figure 15 shows scaled speedup for our four graphs on up to 256 processors of a T3D. Interestingly, the results indicate better performance on graphs in the connected phase when the number of processors is small. The speedup lines for the same surface-to-volume ratio then cross over as the number of processors increase: the disconnected phase speedup continues to rise linearly, while the connected phase falls off. Why do small numbers of processors perform better on graphs in the connected phase?

The answer lies in the relative cost of the local and global phases. The cost of the local phase is significantly greater for the connected phase than for the disconnected phase, not only because of the greater number of edges, but also because of the different algorithmic behavior, as we discussed when modeling the local phase. For small numbers of processors, this extra cost is not reflected in the global phase: ten remote edges that link the same component might be explored more cheaply than five that link distinct components since the algorithm discards redundant remote edges as soon as it recognizes them. The cost of the global phase is thus relatively less for the

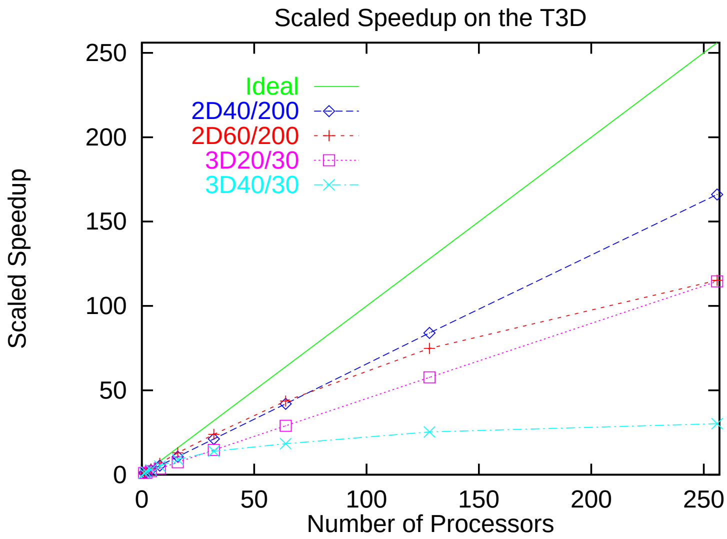


Figure 15: Scaled speedup on a Cray T3D. Note the crossovers between the mostly unconnected and mostly connected graphs in both 2 and 3 dimensions.

graphs in the connected phase, which tend to have many redundant edges, than for the graphs in the disconnected phase, which tend to have few redundant edges. The data in Figures 12 and 13 illustrate this difference: the execution time of the global phase for 3D40/30 only catches up with the time for 3D20/30 at around 8 processors.

These two graphs, one in the disconnected phase and one in the connected phase, have given us a good intuitive grasp of the algorithm's performance. We expect that speedup for disconnected graphs will be linear, and that it will increase as the surface-to-volume ratio decreases. For the connected graphs, we expect to find speedup asymptotically approaching some maximum value, again defined by the surface-to-volume ratio. We have also gained intuition about how we might create a better algorithm for solving connected graphs by identifying the nature of the problems that reduce speedup.

We have yet to investigate one major factor, however: the relative costs of communication and computation on the given machine. The Cray T3D has the best relative communication performance, and so we expect it to give the best speedup results. In the next section, we attempt to use measurements of communication and computational performance to predict speedup on another platform, the CM-5.

5.4 Speedup Prediction

Recall for a moment the simple model of the algorithm as some amount of computation and some amount of communication. The local phase of the algorithm is pure computation and can be measured in terms of the number of nodes processed per second, as we saw when modeling that phase. Let us for now assume that the global phase is dominated by computation, or more

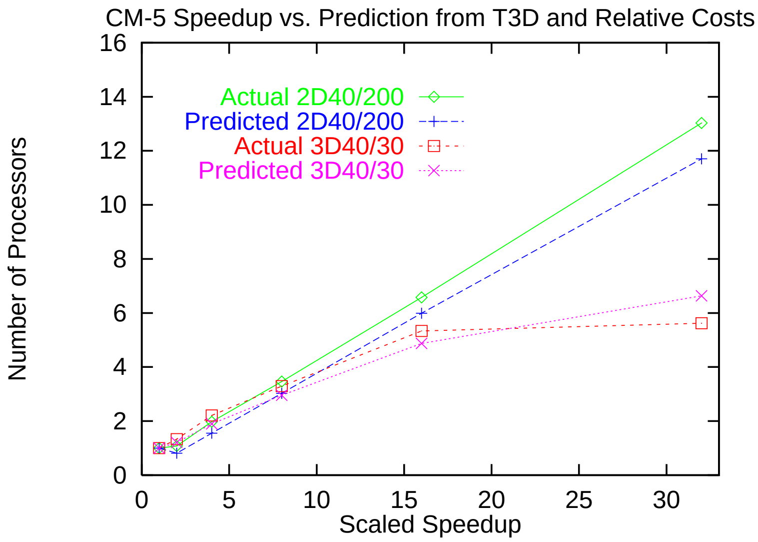


Figure 16: Predicted CM-5 performance. The scaled speedup for 2D40/200 and 3D40/30 is predicted using the scaled speedup of a T3D, the relative costs for communication and computation on the two machines, and a rudimentary cost model.

specifically, dominated by remote reads of small pieces of data. Calling the cost of the local phase on P processors c^P_{local} and the cost of the global phase c^P_{global}, we can express the scaled speedup S as follows:

$$S = \frac{P \ c^P_{local}}{c^P_{local} + c^P_{global}} = \frac{P}{1 + c^P_{global} \ / \ c^P_{local}} \tag{1}$$

Inverting the equation, we can express the ratio of global phase cost to local phase cost in terms of the number of processors and the speedup:

$$\frac{c^P_{global}}{c^P_{local}} = (\frac{P}{S} - 1) \tag{2}$$

Using the assumptions about the content of the phases, we can use (1) and (2) to predict speedup on one machine using the speedup on another machine and the relative costs of communication and computation.

For example, for a 2D40 graph, our local phase measurements indicate that a CM-5 takes 3.5 times longer than a T3D. Similarly, measurements discussed in Section 3 of the time to read a remote value on these machines indicate that the CM-5 takes 12 times as long as the T3D. We can multiply (2) by 12 / 3.5 = 3.426 and then plug back in to (1) to predict speedup on the CM-5. Figure 16 shows the results of such an attempt for 2D40 and 3D40. The predictions are not terribly good, as we should expect from such a simple model, but are within 25% for all values of 2D40 and 18% for all values of 3D40.

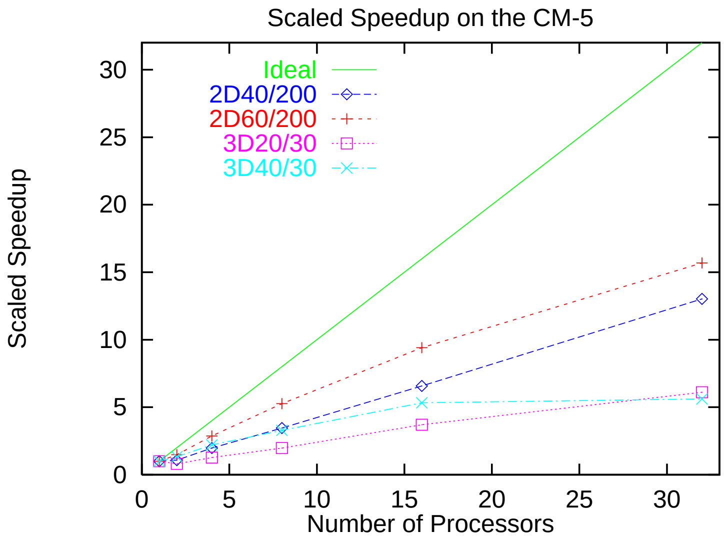


Figure 17: Scaled speedup on the Thinking Machines CM-5. The 2D60 speedup has yet to drop below that of the 2D40 graph at 32 processors.

5.5 Results for CM-5 and CS-2

Figure 17 gives the speedup results for the CM-5, and Figure 18 gives the results for the CS-2. As we expect, the scalability of the algorithm decreases as the relative communication performance of the machine goes down.

6 Conclusion

Connected components is an important problem underlying emerging areas of computational physics, as well as many areas of computer science. Although it has a simple, elegant sequential solution, its parallel solution has proved challenging in practice. In order to obtain high performance in modern distributed memory machines, it is critical to exploit the local node performance. Our approach utilizes a hybrid algorithm, which combines the simple sequential solution on the subgraph local to each node with a variant of classic PRAM solutions on the resulting global graph of subgraphs.

We implemented this alogithm in Split-C and used it to find connected components in a class of graphs called probabilistic meshes, arising in cluster dynamics methods. The graphs are built on an underlying lattice structure, where an instance of the graph is produced by using a single probability to decide whether an edge in the underlying lattice is present in the graph instance. The hybrid approach proved very effective because the local phases are very fast and, by reducing the global graph, reduced the amount of work in the global phases. The resulting solution yields the best performance reported to date on this problem. However, this solution still demands very high communication performance to obtain reasonable speedups. A thorough analysis of the inherent properties of the algorithm provides a natural scaling rule and exposes a critical phase transition

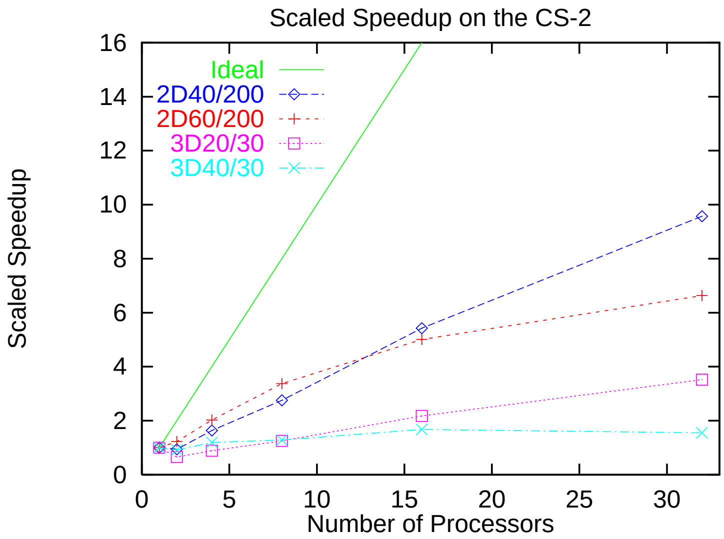


Figure 18: Scaled speedup on the Meiko CS-2.

as the edge probability increases. Below this point, graphs are mostly unconnected collections of small components. The scaled speedup is nearly linear with slope determined by the essential ratio of communication to computational performance. Above this point, the graph is mostly a single large component and the speedup is limited due to load imbalance and remote memory contention. These effects are quantified for the Cray T3D, the Meiko CS-2, and the Thinking Machines CM-5.

References

[1] T. E. Anderson, D. E. Culler, D. Patterson, "A Case for NOW," *IEEE Micro*, February 1995, pp. 54-64.

[2] R. H. Arpaci, et al.,"Empirical Evaluation of the Cray T3D: a compiler perspective," to appear in *Proceedings of the International Symposium on Computer Architecture*, 1995.

[3] B. Awerbuch, Y. Shiloach, "New Connectivity and MSF Algorithms for Ultracomputer and PRAM," *International Conference on Parallel Processing*, 1983, pp. 175-179.

[4] M. Aydin, M. C. Yalabik, *Journal of Physics A* **17**, 2531, 1984.

[5] K. W. Chong, T. W. Lam, "Finding Connected Components in O(log n log log n) Time on EREW PRAM," *4th ACM-SIAM Symposium on Discrete Algorithms*, 1993, pp. 11-20.

[6] A. Choudhary, R. Thakur, "Connected Component Labeling on Coarse Grain Parallel Computers: An Experimental Study," *Journal of Parallel and Distributed Computing* **20**, 1994, pp. 78-83.

[7] 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.

[8] H. G. Evertz, "Vectorized Cluster Search," *Nuclear Physics B, Proceedings Supplements*, 1992, pp. 620-622.

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

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

[11] J. Greiner, "A Comparison of Parallel Algorithms for Connected Components," *6th ACM Symposium on Parallel Algorithms and Architectures*, 1994, pp. 16-23.

[12] R. Hackl, H.-G. Matuttis, J. M. Singer, T. H. Husslein, I. Morgenstern, "Parallelization of the 2D Swendsen-Wang Algorithm," *International Journal of Modern Physics C*, Vol. 4, No. 6, 1993, pp. 59-72.

[13] S. Hambrusch, L. TeWinkel, "A Study of Connected Component Labeling Algorithms on the MPP," *3rd International Conference on Supercomputing*, Vol. 1, May 1988, pp. 477-483.

[14] D. S. Hirschberg, A. . Chandra, D. V. Sarwate, "Computing Connected Components on Parallel Computers," *Communications fo the ACM*, 22(8), 1979, pp. 461-464.

[15] D. B. Johnson, P. Metaxas, "Connected Components in O($\log^3/2$n) Parallel Time for the CREW PRAM," *32nd Annual IEEE Symposium on Foundations of Computer Science*, 1991, pp. 688-697.

[16] C. Kalle, *Journal of Physics A* **17**, L801, 1984.

[17] D. R. Karger, N. Nisan, M. Parnas, "Fast Connected Components Algorithm for the EREW PRAM," *4th ACM Symposium on Parallel Algorithms and Architectures*, 1992, pp. 373-381.

[18] J. Kertész, D. Stauffer, "Swendsen-Wang Dynamics of Large 2D Critical Ising Models," *International Journal of Modern Physics C*, Vol. 3, No. 6, 1992, pp. 1275-1279.

[19] A. Krishnamurthy, S. Lumetta, D. Culler, K. Yelick, "Connected Components on Distributed Memory Machines," to be published in the *Proceedings of the 3rd Annual DIMACS Challenge*, 1995.

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

[21] G. F. Mazenko, M. J. Nolan, O. T. Valls, *Physical Review Letters* **41**, 500, 1978.

[22] H. Mino, "A Vectorized Algorithm for Cluster Formation in the Swendsen-Wang Dynamics," *Computer Physics Communications* **66**, 1991, pp. 25-30.

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

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

[25] R. H. Swendsen, J.-S. Wang, "Nonuniversal Critical Dynamics in Monte Carlo Simulations," *Physical Review Letters*, Vol. 58, No. 2, January 1987, pp. 86-88.

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

[27] J. K. Williams, *Journal of Physics A* **18**, 49, 1985.

[28] H. Yahata, M. Suzuki, *Journal of the Physics Society of Japan* **27**, 1421, 1969.

[29] C. Yoshikawa, "Split-C on the Meiko CS-2," *http://www.CS.Berkeley.EDU/~chad/meiko.ps*, 1995.