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

Steven S. Lumetta, Arvind Krishnamurthy, †
and David E. Culler
Computer Science Division
University of California, Berkeley

April 4, 1996

Abstract: We present and analyze a portable, high-performance algorithm for finding connected components on modern distributed memory multiprocessors. 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 global collection of subgraphs. We implement the algorithm in Split-C and measure performance on the the Cray T3D, the Meiko CS-2, and the Thinking Machines CM-5 using a class of graphs derived from cluster dynamics methods in computational physics. On a 256 processor Cray T3D, the implementation outperforms all previous solutions by an order of magnitude. A characterization of graph parameters allows us to select graphs that highlight key performance features. We study the effects of these parameters and machine characteristics on the balance of time between the local and global phases of the algorithm and find that edge density, surface-to-volume ratio, and relative communication cost dominate performance. By understanding the effect of machine characteristics on performance, the study sheds light on the 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 and computational science. In computer vision, for example, edge detection and object recognition depend on connected components [16]. Connected components algorithms have also advanced the study of physical phenomena, including properties of magnetic materials near critical temperatures. However, the problem offers a unique challenge for parallel computing. Sequential solutions are well understood and commonly used as an application of depth-first and breadth-first search. Parallel solutions have received a great deal of attention from theorists, and have proven difficult. Algorithms such as Shiloach-Vishkin obtain good results with the CRCW PRAM model [3, 11, 12, 28], which assumes uniform memory access time and

†This material is based in part upon work supported by a National Science Foundation Presidential Faculty Fellowship Award, a Graduate Research Fellowship, Infrastructure Grant numbers CDA-8722788 and CDA-9401156, and award number CCR-9210260, by the Lawrence Livermore National Laboratory Grant LLL-B28 3537, by the Advanced Research Projects Agency of the Department of Defense under contracts DABT63-92-C-0026, F30602-95-C-0014, and F30602-95-C-0136, by the Department of Energy under grant number DE-FG03-94ER25206, and by the California MICRO Program. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of any organization.

arbitrary bandwidth to any memory location, but the inherent contention in the algorithm makes even EREW solutions much more challenging [6, 15, 18, 20]. Implementation of the theoretical work has been restricted to shared-memory machines [12] and SIMD machines with very slow processors [12, 17, 23]. Many practical solutions have been developed independently of theoretical work for modern MIMD massively parallel platforms (MPP's) [7, 10, 13, 14, 21, 26] and vector machines [9, 26]. With the exception of [5], which focuses on 2D graphs for robot vision, these solutions typically emphasize performance over portability, scalability, and generality, but still rarely obtain good performance.

We present a fast, portable, general-purpose algorithm for finding connected components on a distributed memory machine. Implemented in Split-C [8], the algorithm is a hybrid of the classic depth-first search on the subgraph local to each processor and a variant of the Shiloach-Vishkin PRAM algorithm on the global collection of subgraphs. On a 256 processor T3D, our implementation performs an order of magnitude better than any previous solution. Using probabilistic mesh graphs found in cluster dynamics problems, we measure the performance of the algorithm on several MPP's with a wide range of computational and communication performance: the Cray T3D, the Meiko CS-2, and the Thinking Machines CM-5. In order to better understand the impact of machine characteristics and graph parameters such as edge density and the surface-to-volume ratio of the subgraphs, we develop a qualitative model of the algorithm's performance. The model explains observed speedup on our three platforms and outlines the possibilities for less tightly integrated systems, where greater computational performance is obtained by sacrificing communication performance [1]. Finally, the modeling process serves as a case study to aid in the understanding of other algorithms.

The remainder of the paper is structured as follows: Section 2 describes the pieces of our study, detailing the parallel algorithm, the implementation language, the machine architectures, and the graph class, then shows the high-level results. Section 3 explains our choice of data points, logically narrowing the range of inputs to a few graph types by identifying key performance issues and justifying a probabilistic analysis. Section 4 presents our results and develops a model of the performance characteristics of the hybrid algorithm across the range of machines and graph types. Section 5 summarizes our results and draws conclusions.

2 Components of the Study

This section discusses general requirements for achieving good performance on MPP's and highlights the components used in our study: the hybrid algorithm, the implementation language, and the parallel platforms. We next describe the class of graphs used as input to the algorithm and explain the significance of these graphs. Finally, we present a high-level overview of performance results and illustrate the advantages of the hybrid strategy over purely global techniques.

At the hardware level, all modern MPP's rely on distributed memory, providing a multi-level memory hierarchy with the cost of a remote access typically hundreds of times that of a local access. If an algorithm does not minimize remote accesses through data locality, the algorithm performs poorly. By splitting the algorithm into distinct local and global phases, we mirror the memory access hierarchy of the machine. During the local phases, the algorithm deals only with data local to each processor. In the global phases, the algorithm addresses the issues of efficient remote data access and synchronization between processors. The cost model used for optimization thus breaks into two simpler models, allowing for more effective optimization.

A parallel *hybrid* algorithm uses this approach, blending a sequential algorithm optimized for local access with an efficient parallel algorithm optimized for the global phases. With a combination of global and local phases, performance generally depends on the balance of time spent in the two types of phases, and is usually good if the local phases dominate the total time.

2.1 Hybrid connected components algorithm

We follow a hybridization strategy to create our algorithm, merging a fast, local depth-first search (DFS) with a powerful and efficient variant of Shiloach-Vishkin (S-V) [28]. The graph is distributed across the processor memories; each node is local to some processor, and we call edges crossing processor boundaries *remote edges*. The DFS then collapses connected components in the subgraph local to a processor, resulting in a much smaller graph for the global phase. The optimized algorithm follows. For more detail on the data structures or on the process of optimization, see [22].

1. **Local Phase.** Perform a local DFS on each processor's portion of the graph, collapsing each local connected component into a representative node. Mark each component of this global graph with a unique value for the global phase.

2. **Global Initialization.** Complete 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, repeat the following:

   a. *Termination Check.* If no components remain to be processed on any processor, quit.

   b. *Hooking Step.* Attempt to merge each component into another component by collapsing an edge. Conditions on merging guarantee that representative nodes remain unique.

   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.

   d. *Self-Loop Removal Step.* For each component, remove edges that point to nodes within the component, *i.e.*, nodes with the same value.

   e. *Edge-List Concatenation Step.* For all leaf nodes of components, move remaining edges to the root.

4. **Local Cleanup.** For each node not selected as a representative node in the global graph, update the value of the node from its representative.

2.2 Programming language

Split-C [8] provides a cost model that mirrors the basic structure of distributed memory machines with a set of simple programming abstractions. The model is valid across many machines, allowing for portable high-performance. Split-C combines the set of processor address spaces into a single *global address space*. Any processor can access any memory location in the global address space using *global pointers*, which are equivalent to standard pointers except for their broader range. Just as pointers help to build data structures in C, global pointers help to create distributed data structures in Split-C. The edges of a distributed graph, for example, are merely global pointers between vertices distributed across the global address space. Nevertheless, distinguishing between local and global data remains straightforward, allowing the local phases of our algorithm to process remote edges differently than local edges without significant overhead. The Split-C cost model and programming abstractions let us focus on the process of optimization while hiding specific hardware details. Split-C also gives our implementation portability, with versions running on the Cray T3D, the IBM SP-1 and SP-2, the Intel Paragon, the Thinking Machines CM-5, the Meiko CS-2, and networks of workstations [2, 24, 27, 34].

2.3 Parallel platforms

We consider three large-scale parallel machines: the Cray T3D, the Meiko CS-2, and the Thinking Machines CM-5. 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 compiler optimizes code for global accesses to take full advantage of hardware specific communication capabilities.

	T3D	CS-2	CM-5
Clock (MHz)	150 (1.00)	90 (0.60)	33 (0.22)
Comm. (Mreads/s)	1.00 (1.00)	0.05 (0.05)	0.08 (0.08)
DFS (Mn/s)	0.78 (1.00)	0.50 (0.64)	0.22 (0.29)

Table 1: Performance variation between platforms. The DFS rating (in millions of nodes processed per second) measures computational performance for our algorithm. Numbers in parentheses are normalized to values on the T3D processor.

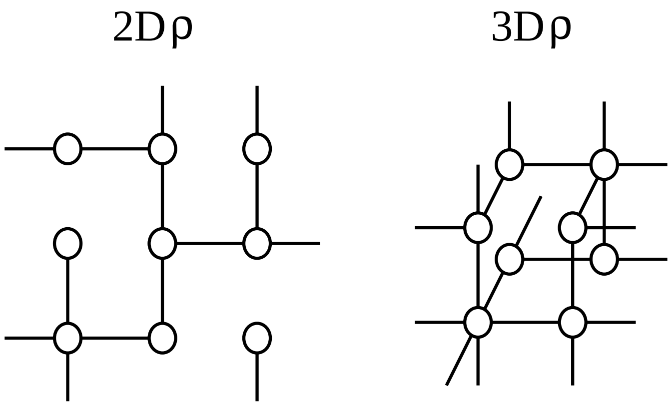


Figure 1: Probabilistic meshes used to study physical phenomena. Each edge in the underlying mesh is present in a given graph instance with probability ρ, independent of other edges.

The T3D processor is 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 uses a 90 MHz, dual-issue Sparc microprocessor with a large cache. A dedicated "ELAN" processor within the network interface supports communication, and is capable of accessing 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 [27, 34].

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 to reply with the value, taking approximately 12 μs [8, 24].

Traditional measures of the computational performance of the node, such as LINPACK, MFLOPS, and SPECmarks, offer little indication of the performance on this integer graph algorithm, which stresses the storage hierarchy, so instead we calibrate the local node performance empirically in Section 4.1. Data on each processor are summarized in Table 1.

2.4 Graph class

We investigate the performance of the algorithm on 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 generally grows as $O(L^2)$ or worse using traditional methods [4, 19, 25, 32, 33]. More recent methods, such as the Swendsen-Wang algorithm

Graph Type	T3D/256 (Mn/s)	CS-2/32 (Mn/s)	CM-5/256 (Mn/s)	C90/1 (Greiner) (Mn/s)
2D30	-	-	-	4.3
2D40/200	128	6.75	33.1	-
2D60/200	83.6	3.31	12.1	2.5
3D20/30	105	2.91	19.1	4.2
3D40/30	38.6	0.79	2.38	2.1

Table 2: 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. The last column shows performance on a single node of a Cray C90 using an algorithm developed by Greiner[12]; Greiner used 2D30 rather than 2D40 in his work.

(S-W), reduce correlation time for the simulations. For example, the correlation time using S-W grows as O($L^{0.35}$) [30, 31] for a two-dimensional Ising model, allowing much larger samples to be studied.

At the heart of S-W is a connected components algorithm. The S-W algorithm repeatedly generates a random graph, finds the connected components of the graph, and applies a simple transformation to each component [31]. In the years since the introduction of the S-W algorithm, cluster dynamics, and hence the use of connected components, has found widespread use in other areas, including computational field theory, experimental high energy and particle physics, and statistical mechanics.

The random graph in S-W is a *probabilistic mesh.* Built on an underlying lattice structure, an instance of the graph is produced by using a single probability ρ 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: 2Dρ is built on a 2-dimensional lattice, and 3Dρ is built on a 3-dimensional lattice.

Although our algorithm accepts arbitrary graphs as input, obtaining good performance requires a reasonable partitioning of the graph across processors. A good partitioning ensures that most of the work occurs in the local phases of the algorithm and that this work is balanced well between processors. Probabilistic meshes offer a natural geometric partitioning via the underlying lattice structures, and we exploit this partitioning for high performance. We simply cut the underlying lattice into equal-size sections, one for each processor.

2.5 Summary of performance results

The performance of our hybrid algorithm is summarized in Table 2 for four important graphs. 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. These graphs highlight the different performance features of the algorithm, as explained in Section 3. Fixed problem size per node scaling is used, so the CS-2 is solving a smaller problem than the T3D or CM-5. However, the problem size is not extremely large, so that the problems use only a reasonable amount of memory. The use of larger graphs only makes obtaining high performance easier. Also shown in the table is the best published uniprocessor performance, obtained on a Cray C90, for comparable graphs.[1] The C90 performance is independent of problem size. The fastest previous result, by Flanigan and Tamayo, achieved 12.5 million nodes per second on much larger 2D graphs using a 256 processor CM-5 [10]. Hackl et. al. achieved nearly 7 Mn/s on a 32 processor Intel IPSC/860, again for very large 2D graphs [13]. Neither of these results can be compared directly, since they were applied to a particular problem and do

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

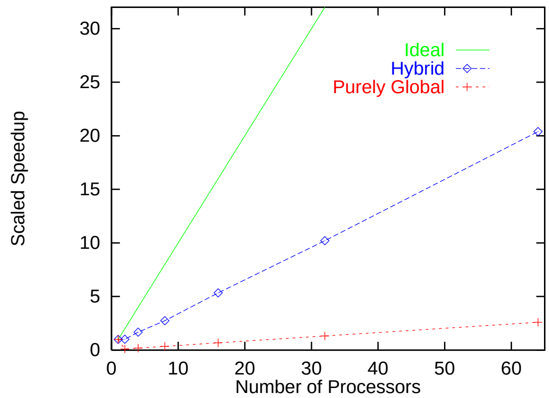


Figure 2: Comparison of scaled speedup for hybrid and purely global algorithms on the CM-5 for 3D20/30. The poor performance of the purely global algorithm illustrates the large overheads incurred by skipping the local phase on parallel executions.

not specifically state the value of ρ for the graphs. As the table demonstrates, both machine characteristics and graph parameters affect performance. Section 4 explores these relationships in depth.

The importance of the hybrid strategy is apparent in Figure 2, which compares scaled speedup of the hybrid and purely global (S-V variant) algorithms for a 3D20/30 graph on the CM-5. Although the purely global algorithm does achieve a constant fraction of ideal speedup, that fraction is only $1/24^{th}$. Results for other graphs are not shown, but are similarly poor, with 2D40/200 obtaining $1/20^{th}$ of ideal speedup. On highly connected graphs, the purely global algorithm never achieves the speed of a single processor due to a phenomenon discussed in Section 3.2.

3 Selection of Input Parameters

The summary data above intimates several key aspects of our methodology for characterizing the behavior of our connected components solution. The space of possible input graphs for any particular machine size is obviously immense. We choose to prune the input space by using four graphs varying in mesh dimension and edge probability. For each graph, we scale the size of the graph with the number of processors, so that the nodes per processor is held constant, *i.e.*, memory constrained scaling [29]. For each data point, we average execution time for twenty graph instances with the specified degree and edge probability. The result is presented as a normalized rate: millions of nodes processed per second (Mn/s). In this section, we explain each of these choices and build a framework for understanding the detailed performance results that follow.

3.1 Graph size and dimension

Memory constrained scaling reflects the desire to use larger machines to solve larger problems, rather than to solve the same problem in less time. However, if the inherent work in the algorithm increases too rapidly with problem size, problems on large machines run too long to be useful [29]. For connected components, however, the work is roughly proportional to the number of nodes in the graph. A connected components algorithm must mark each node, examine each edge, and label each component in the graph. Intuitively, one expects the total work W required for the algorithm to be a sum of terms linear in the number of nodes V, the number of edges E, and the number of components C. For probabilistic meshes, the expected number of edges is proportional to the number of nodes: $\bar{E} = D\rho V$, where D is the dimension of the underlying lattice and ρ is the edge presence probability. The sampling of graphs in Figure 3 indicates that for fixed D and ρ, the expected number of components is also proportional to V: $\bar{C} = c(\rho, D)V$, where c represents the

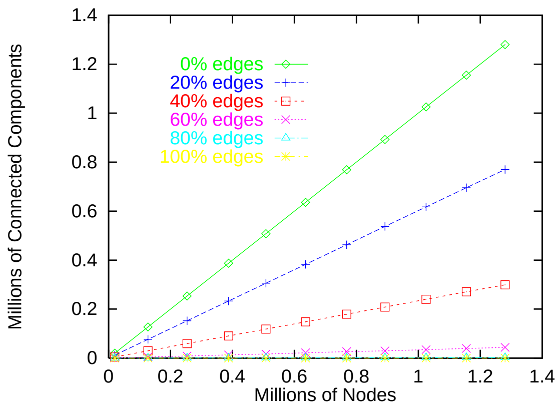


Figure 3: Expected number of connected components for 2D graphs. For a given edge presence probability ρ and underlying lattice dimension D, the expected number of components is linear in the number of nodes.

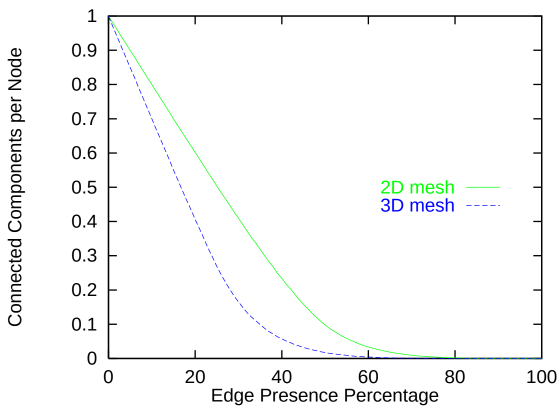


Figure 4: Expected number of connected components per node, $c(\rho, D)$. The lines in the figure are normalized from graphs with 1.28 million (2D) and 0.864 million (3D) nodes. Fractional variance decreases as graph size increases, making these functions very accurate for large graphs.

expected number of connected components per node. Each term in W is proportional to V, hence the work required for a probabilistic mesh is linear in the size of the graph.

Unfortunately, we cannot perform further factorization of the work function. Figure 4 shows the number of components per node, c, for 2D and 3D graphs over a range of ρ. No relationship is apparent between the two curves and both indicate highly non-linear behavior. The work required to find the components of a probabilistic mesh is thus left as the product of V with a function of ρ and D. In our study, we simply fix D, considering the two physically interesting values, $D = 2$ and $D = 3$. We defer discussion of ρ until the following section.

For any particular value of ρ and D, a fixed amount of work must be done for each node in the graph, hence nodes per second (n/s) is a meaningful performance metric. Furthermore, by employing equal nodes per processor scaling with fixed ρ and D, all the graphs used in constructing a scaled speedup curve have the same the same surface-to-volume ratio for the local subgraphs. As the surface-to-volume ratio grows, a larger fraction of edges cross processor boundaries, requiring more communication and reducing performance. We chose to fix the surface-to-volume ratio for each data point, allowing us to investigate other, more subtle, performance effects without trying to factor out the effect of variations in the surface-to-volume ratio. Rather

Name	Edge Presence	Nodes per Processor	Surface/ Volume	Memory per Processor	Size of Largest Component
2D40/200	40%	40,000	1.99%	3.3 MB	0.032%
2D60/200	60%	40,000	1.99%	4.3 MB	95%
3D20/30	20%	27,000	18.7%	1.9 MB	0.039%
3D40/30	40%	27,000	18.7%	2.9 MB	93%

Table 3: Graphs selected for measurement. We vary surface-to-volume ratio by changing the dimension of the graph and examine both disconnected and highly connected graphs. Each graph uses a fixed amount of memory per processor (memory constrained scaling [29]). The last column shows the percentage of nodes in the largest component.

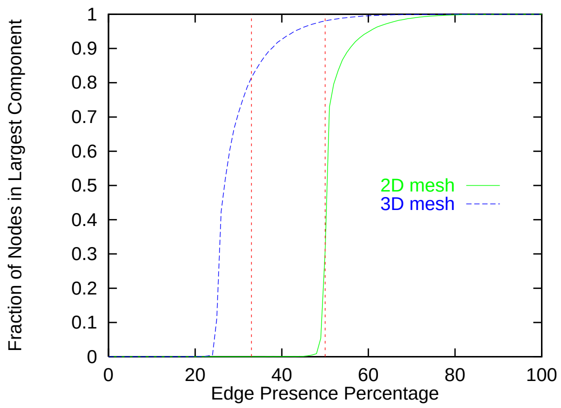


Figure 5: Size of the largest connected component in 2D and 3D probabilistic meshes as a function of ρ. Note the rapid transition in both 2 and 3 dimensions.

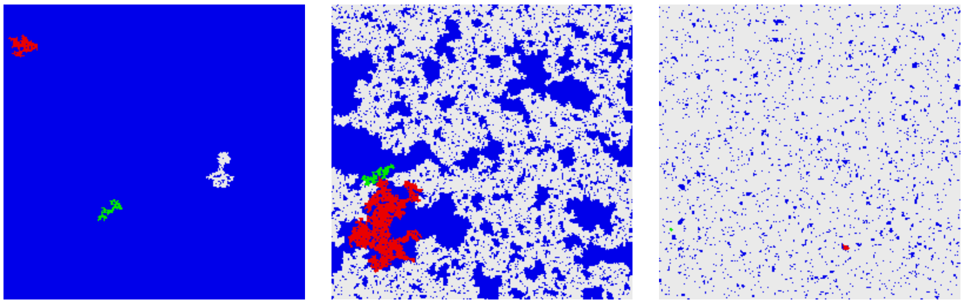
than filling memory to obtain the most impressive performance results, we use moderate sized graphs, much larger than the caches, as indicated in Table 3.

3.2 Edge probability

Having established a strategy for selecting values of V and D, we now consider the edge density ρ. It is known that as edges are added randomly to a graph, the components in the graph grow slowly until a critical point, at which the entire graph begins to collapse into a single, large connected component. Figure 5 shows that this phenomenon is dramatic with probabilistic meshes. Below the sharp phase transition boundary, the graph is in the *liquid state* consisting of a large number of small components. Above the boundary, in the *solid state*, the graph consists of one very large component and a number of tiny components.

The vertical bars in Figure 5 indicate where ρ crosses the value at which the nodes have an average degree of two, 50% for 2D graphs and 33% for 3D graphs, allowing nodes to link into long chains. Figure 6 provides an illustration of the phase transition for 2D graphs. For values of ρ below, on, and above the boundary, the figure highlights the largest three components in a random graph instance.

For our study, we select values of ρ a short distance to either side of the phase transition boundary for each of the two underlying mesh dimensions, giving us four basic graphs which are scaled with the number of processors.



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

Figure 6: Largest connected components for several 256x256 2D graphs. 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. The table gives the number of nodes in each of the components shown; notice both how the size of the largest component increases and the relative size of the second largest component decreases as we increase ρ.

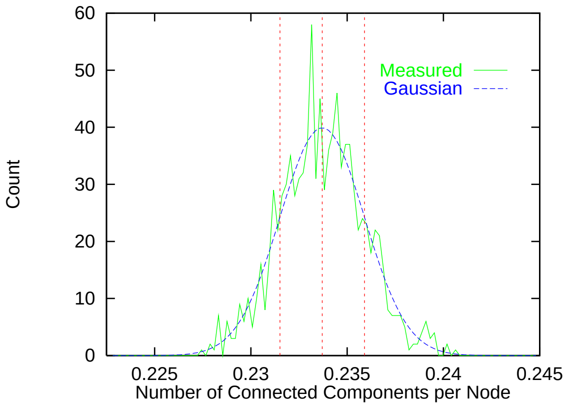


Figure 7: Distribution of number of connected components. Note how closely the distribution of 1,000 samples from a 256x256 2D40 graph matches a Gaussian distribution with the same mean and standard deviation. The vertical lines mark the mean and one standard deviation in either direction.

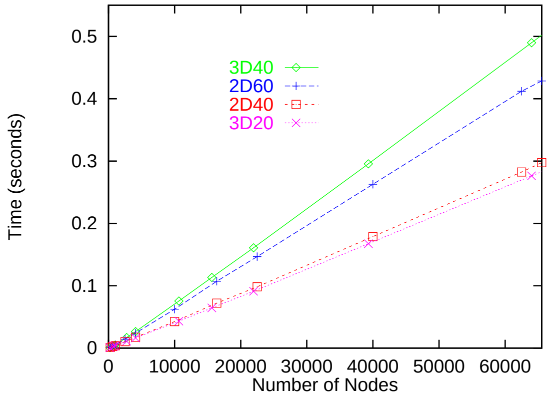


Figure 8: Local phase execution time on a CM-5 node. Execution time is linear in the size of the graph. Results for other platforms reveal a similar dependence.

3.3 Sampling graph instances

Working with classes of graphs rather than specific problem instances implies that we must average our measurements over a number of random instances. The validity of averaging depends on the nature of the functions involved, requiring not only that the distributions of random variables be reasonably narrow, but also that the functions be smooth near the area of interest. Narrow distributions allow a small number of random samples to provide accurate results, while smooth functions guarantee that minor random variations result in minor changes to the measured values.

For probabilistic meshes, we need only demonstrate that $\bar{E}$ and $\bar{C}$ have narrow distributions. The work function W expressed in linear terms of V, E, and C is certainly smooth, and V is fixed for any given measurement. $\bar{E}$ has a binomial distribution, giving decreasing fractional variance as the graph size increases. Perhaps unexpectedly, $\bar{C}$ is also statistically well-behaved and can be treated as a normal random variable. Consider Figure 7, which shows the distribution of number of connected components per node found in 1,000 2D40 graphs with 65,536 nodes. The mean falls at 0.234, as we saw in Figure 4, and the standard deviation is 0.00220. The bin width in the figure is one tenth of the standard deviation. For easy comparison, the figure also shows a Gaussian distribution with the same mean and standard deviation. With a smooth function and narrow distributions, we can reliably measure the function with the average of a small number of random samples.

4 Performance and Models

This section reports the results of our measurements, highlighting the balance between the execution time of the local and global phases. We develop intuition about the effects of graph parameters and machine characteristics on performance by studying their effects on the local/global balance. We begin with an investigation of the initial local phase, then consider the global phases for the liquid and solid states, and conclude with a comparison of speedup results between machines.

4.1 Local phase results

Measurements confirm that the execution time of the local phase is linear in V for each of our graphs, as shown in Figure 8. With slight modifications to remove features unnecessary in a sequential algorithm, the local phase serves as the baseline for measurements of scaled speedup. Table 4 presents the computational capabilities of a single node on each of our platforms. Relative performance is different for each graph and is

Graph Type	T3D (Mn/s)	CS-2 (Mn/s)	CM-5 (Mn/s)	C90 (Greiner) (Mn/s)
2D30	-	-	-	4.3
2D40/200	0.781 (1.00)	0.498 (0.64)	0.223 (0.29)	-
2D60/200	0.574 (1.00)	0.335 (0.58)	0.147 (0.26)	2.5 (4.36)
3D20/30	0.885 (1.00)	0.514 (0.58)	0.239 (0.27)	4.2 (4.75)
3D40/30	0.540 (1.00)	0.303 (0.56)	0.133 (0.25)	2.1 (3.89)

Table 4: Local phase (DFS) performance. These numbers provide a baseline for our measurements of scaled speedup. For comparison, the last column shows Greiner's Cray C90 results [12]. Parenthesized values are normalized to the performance of a T3D processor.

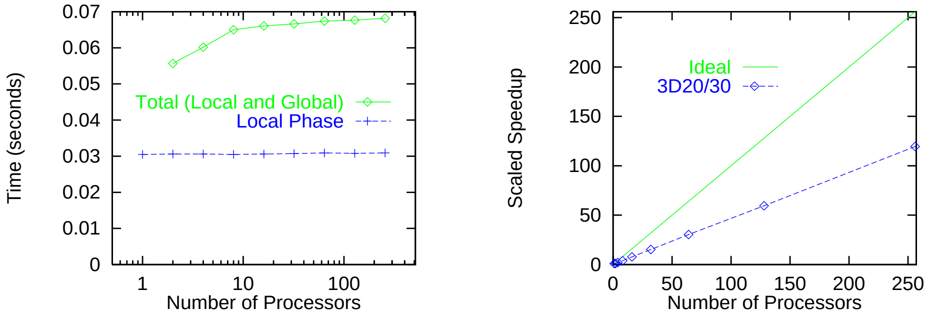


Figure 9: Breakdown of execution time for 3D20/30 on a T3D. The local cost remains nearly constant as the number of processors increases. The global cost for a graph in the liquid state quickly reaches a plateau, resulting in speedup close to a constant fraction of ideal.

a function of many facets of machine architecture, making performance results between machines comparable only for the same graph.

4.2 Global phase results

We next explore the balance between time spent in the local and global phases. Speedup depends most strongly on the phase transition; the performance behavior changes significantly between the liquid and solid states. We begin with an exploration of the liquid state. Figure 9 shows the local and global execution times for 3D20/30 for a T3D. For graphs in the liquid state, the algorithm obtains a constant fraction of ideal speedup. The disconnected nature of the liquid state makes interactions between processors owning non-adjacent subgraphs rare, hence the execution time for the global phase is essentially independent of the number of processors. The slight upward trend in global execution time arises from a load imbalance explicable through simple statistics. A single random value chosen from a normal distribution is likely to fall near the mean, but the maximum of multiple values chosen from the same distribution rises as the number of values increases. The barrier synchronization between phases of the hybrid algorithm acts as a MAX function across the individual processor execution times, explaining the form seen in the figure.

The speedup obtained for a graph depends on the relative cost of the local and global phases. For larger surface-to-volume ratios, the global phase must process a larger fraction of the graph edges, resulting in a smaller fraction of ideal speedup. 2D40/200, with a surface-to-volume ratio of 1.99%, achieves a speedup of $0.634P$, while 3D20/30 with a surface-to-volume ratio of 18.7%, achieves somewhat less, $0.467P$.

Unlike graphs in the liquid state, the global time for 3D40/30 grows almost linearly in the number of processors. The linear increase in the cost of the global phase causes speedup to drop and eventually leads to an asymptotic limit on the speedup achieved by the algorithm. Figure 10 displays local and global execution times for 3D40/30 on a T3D. Graphs in the solid state are dominated by a single component. After the DFS,

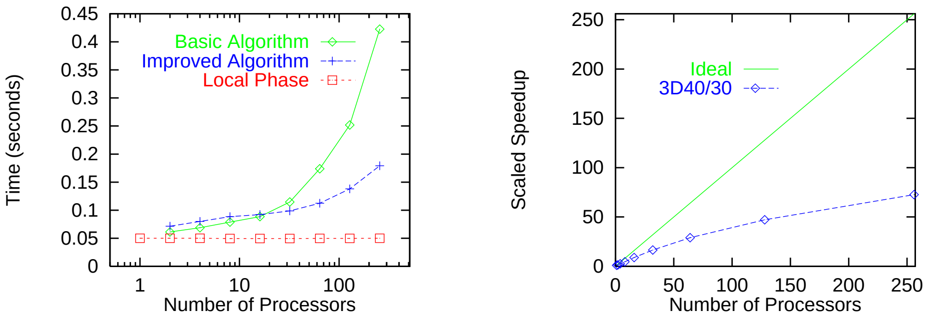


Figure 10: Breakdown of execution time and scaled speedup for 3D40/30 on a T3D. Our basic algorithm suffered from both load balance and contention problems resulting in an almost linear increase in the global cost. With a minor modification to the local phase, we reduced the load imbalance significantly. Scaled speedup corresponds to the improved algorithm.

	Basic	Improved
Maximum	8,594	155
Average	960	8

Table 5: Number of remote edge structures read for a 3D40/30 graph instance using 32 processors. Some processors perform an order of magnitude more remote edge reads than other processors. The new numbering scheme decreases the number read and creates a better, but still uneven, load balance.

each processor owns a piece of this big component, which join together in the first iteration of the global phase to form one or more giant components. In the unfortunate case that more than one forms, remaining edges between the giant components propagate to the new component owners and must be investigated in future iterations via remote reads. Since only the owners of the giant components perform these remote accesses, a load imbalance arises, as is apparent from Table 5, which shows the difference between the maximum number of edge structures explored by a single processor and the average number. In fact, the maximum number read by a processor is proportional to P, the number of processors, leading to the execution time behavior we observe.

A minor modification to the algorithm mitigates this problem by ensuring that only a single giant component forms in the first iteration, allowing internal (self-loop) edges to be removed before they propagate. After the DFS, we assign the largest component on a processor a value that allows it to connect only to another processor's largest component. Table 5 and Figure 10 capture the utility of the improvement: an order of magnitude decrease in remote edge reads leading to a significant drop in global execution time for large numbers of processors. The increased time for less than 16 processors corresponds to the time spent finding the largest component in the local phase (the figure shows only the local time for the improved algorithm).

The remaining upward trend for large numbers of processors is due to remote memory contention. The large component is owned by a single processor, drawing nearly all parent link reads to that processor. On a CM-5, for example, we observe an average time of 100 μs for each remote access during the termination check, more than an eightfold increase over the uncongested time of 12 μs. Removing this contention requires only that we unroll the first global iteration from the loop and insert a step to redirect parent links to local addresses, but we have yet to attempt this change.

As with the liquid state, graphs in the solid state obtain better performance with lower surface-to-volume ratios. The 3D40/30 speedup on a T3D has flattened out to about 73 for 256 processors. For 2D60/200, however, the speedup has barely started to fall from linear behavior when it reaches 147 at 256 processors.

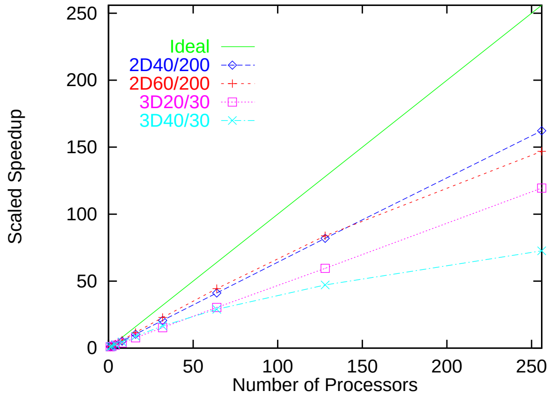


Figure 11: Scaled speedup on a Cray T3D. Note the crossovers between the mostly unconnected and mostly connected graphs in both 2 and 3 dimensions.

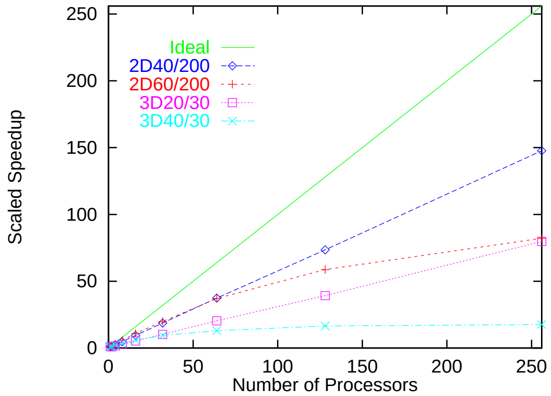


Figure 12: Scaled speedup on a Thinking Machines CM-5. The 3D40/30 speedup has nearly reached its limit, increasing by only 6% between 128 and 256 processors.

To observe an asymptotic limit on speedup, we look to the CM-5 and CS-2 results below; both machines have relatively slow communication compared with a T3D.

4.3 Effect of communication costs

We now combine our understanding of performance behavior for each graph with knowledge of the different machines. Figure 11 shows the scaled speedup for all four graphs on up to 256 processors of a T3D. As mentioned, both graphs in the liquid state attain a constant fraction of ideal speedup: $0.634P$ for 2D40/200 and $0.467P$ for 3D20/30. Although both graphs in the solid state have begun to fall off by 128 processors, neither is close to an asymptotic limit as expected for large numbers of processors.

Interestingly, the results indicate better speedup in the solid state for small numbers of processors. The reason lies in the relative cost of the local and global phases. The cost of the local phase is significantly greater for the solid state than for the liquid state due to the greater number of edges, but this extra cost is not reflected in the global phase for small numbers of processors. The extra remote edges are often redundant and are discarded immediately, making the global phase relatively inexpensive for the solid state.

Figure 12 shows scaled speedup on up to 256 processors of a CM-5. From Table 1, we find that the computation to communication ratio of a CM-5 processor is about 3.5 times that of a T3D processor. With

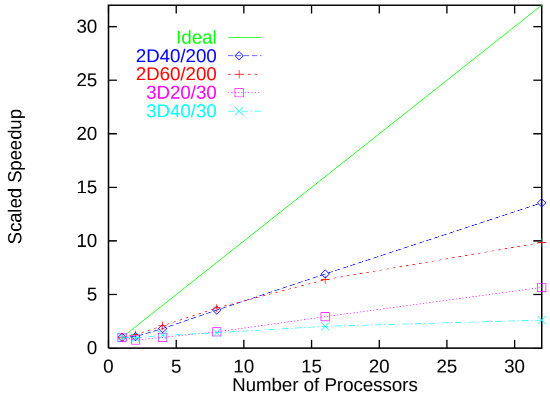


Figure 13: Scaled speedup on a Meiko CS-2. The high cost of communication compared with computation results in poorer speedups than those attained on a T3D and a CM-5.

more expensive communication, the balance of time between the local and global phases shifts towards the global phase, reducing performance. The effect is least noticeable for the 2D40/200 graph. The global phase for this graph is short due to its low surface-to-volume ratio and disconnected structure, allowing a CM-5 to achieve 91% of the T3D speedup. The high surface-to-volume ratio of 3D20/30 reduces performance to only 67% of T3D performance.

The high cost of communication on a CM-5, coupled with more dramatic contention effects, also results in different behavior for the graphs in the solid state. Most obvious is the limited speedup for 3D40/30, which has nearly reached an asymptotic limit of 18. Both graphs begin to fall from linear speedup much earlier than on a T3D, making the difference between the solid and liquid states much more apparent.

Scaled speedup for up to 32 processors of a CS-2 appears in Figure 13. The computation to communication ratio of a CS-2 processor is about 13 times that of a T3D processor, resulting in much longer global phases and greatly reduced speedups. The algorithm performs worst on a CS-2, attaining only 66% of T3D speedup for 2D40/200 and 37% for 3D20/30. Solid state performance has already started to fall off at 8 processors, and is flat by 32 for 3D40/30, achieving a speedup of only 2.6.

5 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 on modern distributed memory machines, an algorithm must exploit local node performance. Our approach utilizes a hybrid algorithm, combining a simple sequential solution on the subgraph local to each node with a variant of a classic PRAM solution on the resulting global graph of subgraphs.

We implemented this algorithm in Split-C and used it to find connected components for a class of graphs called probabilistic meshes, arising from cluster dynamics methods. The graphs are built on an underlying lattice structure, with an instance of a graph produced by using a single probability to decide whether each edge in the underlying lattice is present in the graph instance.

The hybrid approach proved effective because the local phases are very fast and, by significantly simplifying the global graph, reduce the amount of work in the global phases. The resulting solution yields the best performance reported to date on this problem, but demands high communication performance to obtain good speedups.

A thorough analysis of the inherent properties of the algorithm supports the use of memory constrained scaling and exposes a critical phase transition as the edge probability increases. Below the transition boundary, graphs are mostly unconnected collections of small components. The scaled speedup is linear with slope determined by the ratio of communication to computation performance of the machine. Above the transition boundary, one large component dominates the graph, and speedup is limited by remote memory contention and load imbalance; a minor modification alleviated the load imbalance problem, but a second, uncompleted change is required to eliminate the contention and to obtain good speedups for the connected phase. For both phases, speedup increases as the surface-to-volume ration of the local subgraphs decreases. These effects are quantified for the Cray T3D, the Meiko CS-2, and the Thinking Machines CM-5.

The source code for our implementation and an expanded set of results are available from the authors' home pages: *http://HTTP.CS.Berkeley.EDU/*{∼*stevel,*∼*arvindk,*∼*culler*}.

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

[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] D. A. Bader, J. JáJá, "Parallel Algorithms for Image Histogramming and Connected Components with and Experimental Study," University of Maryland Institute for Advanced Computer Science Technical Report #UMIACS-TR-94-133.

[6] 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.

[7] 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.

[8] 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*, November 1993, pp. 262-273.

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

[10] 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.

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

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

[13] 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.

[14] 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.

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

[16] B. K. P. Horn, "Robot Vision," The MIT Press, Cambridge, Massachusetts, 1986.

[17] T.-S. Hsu, V. Ramachandran, N. Dean, "Parallel Implementation of Algorithms for Finding Connected Components," to be published in the *Proceedings of the 3rd Annual DIMACS Challenge*, 1995.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] S. Kumar, S. M. Goddard, J. F. Prins, "Connected-Components Algorithms for Mesh-Connected Parallel Computers," to be published in the *Proceedings of the 3rd Annual DIMACS Challenge*, 1995.

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

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

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

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

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

[29] J. P. Singh, J. Hennessy, A. Gupta, "Scaling Parallel Programs for Multiprocessors: Methodology and Examples," IEEE *Computer* **26(7)**, July 1993, pp. 42-50.

[30] 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.

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

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

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

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

Steven S. Lumetta is a Computer Science graduate student at the University of California at Berkeley. He received an A.B. in Physics from U.C. Berkeley in 1991 and an M.S. in Computer Science in 1994 for his work on the Mantis parallel debugger. When not researching parallel applications, languages, tools, or environments, he spends time with his wife and son and occasionally manages to play volleyball.

e-mail: *stevel@CS.Berkeley.EDU*, home page: *http://HTTP.CS.Berkeley.EDU/~stevel*

Arvind Krishnamurthy is a graduate student at UC, Berkeley. He received his BTech in Computer Science from IIT, Madras in 1991 and his MS in Computer Science from UC, Berkeley in 1994. His research interests include parallel computing, programming languages, and compilers.

e-mail: *arvindk@CS.Berkeley.EDU*, home page: *http://HTTP.CS.Berkeley.EDU/~arvindk*

David E. Culler joined the University of California in 1989 after receiving his Ph.D. from MIT, and has taught both computer architecture and parallel processing. David received the NSF Presidential Young Investigator in 1990 and the Presidential Faculty Fellowship in 1992. His research addresses parallel computer architecture, parallel programming languages, and high performance communication structures. He is known for his work on resource management in dataflow systems, compilation of lenient parallel languages using a Threaded Abstract Machine (TAM), fast communication on modern parallel machines using Active Messages, and Split-C. He is currently working on networks of workstations (NOW).

e-mail: *culler@CS.Berkeley.EDU*, home page: *http://HTTP.CS.Berkeley.EDU/~culler*