Appears in *Proceedings of the 13th Annual International Conference on Supercomputing*, June 1999.

The Scalability of Multigrain Systems

Donald Yeung
Department of Electrical and Computer Engineering
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742

Abstract

Researchers have recently proposed coupling small- to medium-scale multiprocessors to build large-scale shared memory machines, known as *multigrain shared memory systems*. Multigrain systems promise low cost because they leverage commodity multiprocessor nodes, and high performance because each multiprocessor node provides fine-grain shared memory mechanisms. Unfortunately, a quantitative study to evaluate the scalability of multigrain systems has thus far been lacking. Such scalability studies are difficult to undertake because of the limited system size that experimental evaluation can explore.

This paper studies the scalability of multigrain systems using analysis. The paper proposes a novel methodology for analyzing program behavior on multigrain systems, called *synchronization analysis*. Synchronization analysis predicts communication volume by examining an application's synchronization behavior, an effective technique because on software DSMs, actual communication closely follows synchronization. Then, the paper presents a performance model that computes end-to-end application runtime based on an estimated cost for the predicted communication. On five shared memory applications, the performance model is accurate to within 18% of measured runtime for four applications, and within 22% for all five. Using the model, the paper conducts an in-depth study of multigrain system scalability. The paper shows that for multigrain systems with 512 processors, high performance can be achieved on four out of our five applications if each multiprocessor node is at least 16-way.

1 Introduction

Recently, researchers have proposed building large-scale distributed shared memory (DSM) systems by coupling multiple small-scale shared memory multiprocessors [1, 2, 3, 4, 5]. These systems combine fine-grain cache-coherence mechanisms supported in hardware (within a small-scale multiprocessor) and coarse-grain software page-based mechanisms supported in software (between small-scale multiprocessors). Because they employ two different coherence units (both cache-lines and pages), these systems have been referred to as *multigrain shared memory systems* [5].

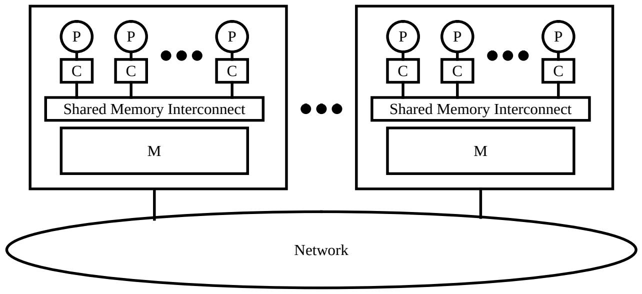

Figure 1: A multigrain shared memory system is built using small-scale multiprocessors as DSM nodes. “P” denotes processors, “C” hardware caches, and “M” physical memory modules on each DSM node.

Figure 1 shows the architecture of a multigrain shared memory system. Like any conventional DSM, a multigrain system consists of a collection of nodes connected over a network. Each node is a shared memory multiprocessor containing a few (2-100) processors each with its own hardware cache, special-purpose hardware support for shared memory and cache coherence, local physical memory, and a network interface to the external network. Two different multiprocessor architectures are possible for the node: the symmetric multiprocessor (SMP) or the cache-coherent non-uniform memory access multiprocessor (CC-NUMA). SMPs are the choice for the near future due to their commercial success, but CC-NUMAs would enable larger nodes due to their scalability.

Since each DSM node in a multigrain system is a multiprocessor, support for shared memory is already provided in hardware between processors collocated on the same DSM node. To synthesize a single shared memory address space across the entire cluster, multigrain systems must also provide a shared memory layer between DSM nodes–this is accomplished in software using page-based techniques, as initially proposed in [6]. The software shared memory layer employs many of the same communication reduction techniques found in conventional page-based DSMs that use uniprocessor workstations as DSM nodes, such as relaxed memory consistency, multiple writers [7], and lazy coherence [8]. In addition, mechanisms specific to multigrain systems that optimize for the collocation of processors within the same DSM node are desirable for high performance (see Section 4.1). Finally, an im-

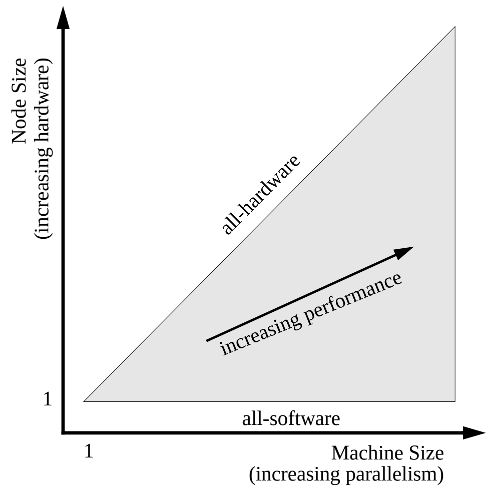

Figure 2: The machine space of multigrain systems parameterized by node size along the Y-axis and machine size along the X-axis. Performance increases as both parameters are scaled.

portant component of the inter-node shared memory layer is the network which connects DSM nodes. We envision some kind of high-performance local area network for this purpose, such as ATM or switched Ethernet.

Multigrain systems are attractive architectures for two reasons. First, they are extremely cost-effective because they employ commodity nodes. Small- to medium-scale multiprocessors are already ubiquitous thanks to the demand for high-performance servers. As desktop machines become more powerful, multiple processors will make their way into client workstations as well. Second, multigrain systems provide support for fine-grain sharing. Conventional wisdom maintains that software DSMs are incapable of supporting fine-grain applications due to the lack of efficient mechanisms for communication [9]. However, unlike traditional software DSMs that use uniprocessor nodes, multigrain systems provide hardware cache-coherence between processors within each multiprocessor; therefore, fine-grain sharing is supported efficiently within DSM nodes.

1.1 Scaling Multigrain Systems

Multigrain systems can be scaled along two different dimensions: node size, and machine size. Node size is the number of processors in each shared memory multiprocessor (*i.e.* in a single DSM node), and can be scaled by adding processors to each multiprocessor. Machine size is the total number of processors in the whole system. Scaling machine size can be achieved by either adding nodes, or by scaling node size. Each type of scaling has a different impact on machine behavior. Scaling node size increases the amount of shared memory hardware in the system. If node size is scaled while the number of nodes is kept fixed, then the fraction of processors that communicate via hardware shared memory increases relative to the fraction of processors that communicate via software shared memory. Therefore, node size scaling increases the system's ability to support fine-grain applications. In contrast, scaling machine size increases the amount of parallelism supported by the system.

Figure 2 graphically illustrates multigrain system scaling. The figure plots node size along the Y-axis and machine size along the X-axis.[1] A loci of points is drawn which represents the complete space of multigrain shared memory systems that can be realized by scaling node size and machine size. The machine space is bounded by two lines intersecting at a 45^o angle. Each line corresponds to points in the machine space where node size scaling causes the multigrain systems to degenerate into either all-software or all-hardware systems, beyond which scaling is undefined. When node size is scaled down to a single processor (lower bound), the multigrain system becomes an all-software page-based DSM with uniprocessor nodes. When node size is scaled up to the size of the machine (upper bound), the multigrain system becomes an all-hardware cache-coherent machine. Overall system performance increases with scaling along both dimensions, as indicated by the slanted line in Figure 2.

Understanding how multigrain system performance varies as a function of both node size and machine size is important to systems architects. For instance, given a desired level of performance, a systems architect can meet the performance specification either by building a larger system with small nodes, or a smaller system with large nodes, *i.e.* comparing systems along a curve in machine space that has negative slope. Such a curve identifies *performance-equivalent* systems. However, making such a design tradeoff requires knowing the effects of simultaneously scaling node and machine size. The interaction between these two types of scaling may be complex, particularly for applications with lots of communication.

1.2 Studying Scalability

Preliminary studies [5] have provided early evidence that multigrain systems are effective architectures, even for difficult fine-grain applications. However, these studies were performed on small (32-processor) multigrain systems because they were limited by the size of the experimental platform available for the study. Important questions, such as how much fine-grain support is needed and whether localized fine-grain sharing is adequate for applications, cannot be addressed on such small systems. Furthermore, many of the node and machine scaling effects discussed in Section 1.1 do not appear except on very large configurations. Therefore, addressing these issues experimentally, particularly those concerning scalability, is impractical due to the size limitations of experimental platforms. This paper adopts an alternate approach: use analysis to study large-scale multigrain systems.

The remainder of this paper consists of two major parts. First, we present a novel analysis technique, known as synchronization analysis (Section 2), and performance model (Section 3) that enables a quantitative evaluation of multigrain system performance at arbitrary node and machine sizes. Second, Section 4 validates the performance model and uses it to conduct an in-depth study of the scalability of multigrain systems on several shared memory programs. Section 5 presents our conclusions.

[1] While the figure suggests that the two dimensions are orthogonal, in actuality scaling node size without changing machine size (*i.e.* a vertical line) implies that the number of nodes is scaled inversely proportional to node size since scaling node size by itself will increase machine size as well.

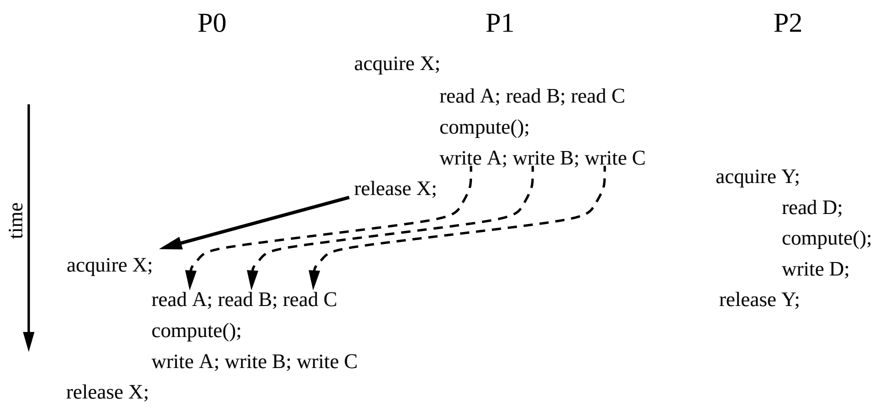

Figure 3: Given an explicitly parallel code, data dependences (dashed lines) between shared memory references performed on different processors can be identified by examining synchronization dependences (solid bold line).

2 Analyzing Software DSM Behavior

Cache-memory systems are difficult to analyze because they exhibit highly unpredictable behavior. Consider the following three types of cache misses that can occur on a hardware cache-coherent machine: capacity, conflict, and coherence. Predicting capacity and conflict misses requires detailed information about each processor's reference stream, such as the number of unique references and how temporally related references map to lines in the cache. Predicting coherence misses requires knowledge about which references performed on different processors conflict and how these conflicting references interleave in time.

In this paper, we demonstrate that software shared memory systems are much more amenable to accurate analysis for two reasons. First, software caching uses main memory for cache storage. This provides an effectively infinite cache (on most workloads) which is fully associative; therefore, software DSMs will never suffer capacity or conflict misses. Second, in software caching, the analysis of coherence misses is made tractable by the properties of the release consistency (RC) memory model. RC allows the consistency of updated data to be delayed until special memory operations in a program, known as *releases* and *acquires* [10]. Furthermore, properly written shared memory programs for RC memory models must include source-level annotations that identify release and acquire operations. Therefore, analysis of the application source code can yield all the potential communication sites in the application, and ultimately, the communication volume.

In the next section, we present an analysis technique that computes communication volume for applications running on software DSMs, called *synchronization analysis*.

2.1 Synchronization Analysis

In most high-performance software shared memory systems [7, 11, 12], communication occurs at either releases or acquires; shared memory accesses performed in between these special communication sites are buffered locally in order to minimize communication. Therefore, all *potential* communication sites in an application can be identified by creating an execution graph which represents the dynamic execution of all acquires and releases. To compute total communication volume, our analysis must perform two tasks: identify those dynamic acquire and release instances that actually generate communication, and for each of these communication sites, compute the volume of communication generated.

The identification of communication sites can be greatly simplified if we assume that applications use different names to perform coherence operations on unrelated data. Figure 3 illustrates the behavior of an application that obeys this assumption. The figure shows three processors making mutually exclusive accesses to four shared memory locations named A, B, C, and D, using synchronization variables X and Y. In this example, modifications to locations A, B, and C are always performed together, and use synchronization variable X. Modifications to location D are performed separately and use synchronization variable Y. Because different synchronization variables are used, modifications to location D can occur simultaneously with modifications to the other three shared memory locations. However, processors $P0$ and $P1$ must serialize their modifications to locations A, B, and C because they use the same synchronization variable X, thus enforcing mutual exclusion.

Figure 3 shows that each synchronization dependence between two processors represented as a release $\rightarrow$ acquire dependence over the same synchronization variable signifies one or more data dependences, and thus data sharing. In our example, processors $P0$ and $P1$ share locations A, B, and C. This sharing is marked by the release $\rightarrow$ acquire dependence over the synchronization variable X from $P1$ to $P0$. Conversely, because there is no data sharing between processor $P2$ and processors $P0$ and $P1$, no release $\rightarrow$ acquire dependence can be identified between these processors due to the use of separate synchronization variables. In essence, our approach identifies data dependences, and thus communication sites, through the analysis of synchronization dependences. Because we use synchronization dependence information to infer data dependences, we call the approach synchronization analysis.

After identifying the communication sites using synchronization analysis, we must compute the volume of data communicated at each site. This requires analyzing data access information to determine what memory locations have been modified prior to the communication site. In particular, we must compute the number of unique shared memory locations modified within the code surrounded by the acquire and the release prior to the release $\rightarrow$ acquire synchronization dependence. For instance, the code just prior to the synchronization dependence on variable X in Figure 3 running on processor $P1$ modifies three shared memory locations. From these shared memory accesses, we must then compute the number of unique pages that have been modified. Finally, we must determine how many of these modified pages will actually require coherence and thus contribute to communication across the synchronization dependence. We pessimistically assume that all the pages updated between the acquire and release are communicated. This is a valid assumption when the programmer matches the granularity of synchronization to the granularity of data sharing. From our experience, this assumption is valid for most applications.

While we will show that synchronization analysis can yield very accurate predictions of communication volume on several applications, it has some limitations. Because the technique relies on detecting communication through analysis of synchronization behavior, any communication that isn't explicitly synchronized goes undetected. For instance, synchronization analysis cannot detect false sharing communication. Because false sharing arises in the absence of true data dependences, communication due to false sharing will never be explicitly synchronized. In addition, synchronization analysis will not be effective on applications that permit data races. An example might be an application which provides mutual exclusion on shared data for writes, but does not do the same for reads.

2.2 Clustering Analysis

Synchronization analysis accurately estimates communication volume for a program running on a software DSM via analysis of the program's source code. The technique can be applied on any software DSM system that supports a release consistent memory model, and that makes use of delayed coherence to reduce internode communication. The presentation of synchronization analysis in Section 2.1, however, assumes a flat all-software shared memory system. The analysis technique must be extended to handle multigrain shared memory systems.

In multigrain systems, not all release $\rightarrow$ acquire dependences invoke software communication; only those synchronization dependences between processors on different DSM nodes incur software overhead. Synchronization dependences between processors inside a single multiprocessor node are handled by hardware mechanisms and thus bypass software shared memory. For synchronization analysis to accurately predict communication on multigrain systems, the analysis must consider the clustering of individual processors within DSM nodes. The extension, known as *clustering analysis*, is quite simple: out of all the release $\rightarrow$ acquire dependences computed by synchronization analysis, identify only those that cross DSM node boundaries as the communication-generating dependences.

Figure 4 illustrates clustering analysis. The figure shows a synchronization dependence graph from a hypothetical application in which each graph node represents a piece of code surrounded by an acquire and a release. The arrows with filled arrowheads represent synchronization dependences, while the arrows with unfilled arrowheads represent control dependences for graph nodes executed on the same processor. In this particular example, there are four processors, $P0$ through $P3$, organized across two DSM nodes of two processors each. The dotted line represents the physical node boundary between the four processors. Of the seven synchronization dependence arcs in the example graph, only three of the arcs cross the DSM node boundary. Clustering analysis will identify these three arcs as the communication-generating arcs. The other four arcs are "hidden" from the software shared memory layer and thus do not generate communication.

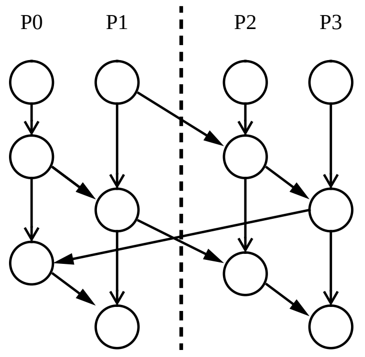

Figure 4: Clustering analysis determines which synchronization dependences cause communication.

3 Performance Model

Section 2 discusses how to compute communication volume on a multigrain shared memory system. In this section, we introduce a performance model that predicts application runtime using the computed communication volume.

To predict an application's runtime on a multigrain system with a total machine size of P processors, our performance model assumes that the execution time on a P-processor all-hardware cache-coherent shared memory machine, R_{hw}, is known. In general, R_{hw} is difficult to predict. We expect the value to be provided to the model by measurement. The value can either be directly measured on a hardware cache-coherent machine with the desired number of processors, or in those instances where the target multigrain system is too large, the value can be extrapolated to the desired machine size from speedup curves obtained on a smaller hardware cache-coherent machine. The performance model then predicts the execution time on the multigrain system by dilating the cache-coherent runtime with the overheads that arise as a consequence of using software shared memory. The prediction of execution time contains five terms:

$$runtime = R_{hw} + SM_{lat} + SM_{occ} + SYN_{lock} + SYN_{bar} \quad (1)$$

where SM_{lat} and SM_{occ} (discussed in Section 3.1) are overheads due to software shared memory, and SYN_{lock} and SYN_{bar} (discussed in Section 3.2) are overheads due to synchronization. In Equation 1, R_{hw} is constant irrespective of node size. In general, this R_{hw} is too large for multigrain systems because some of the inter-processor communication is handled by software shared memory and is already accounted for in the SM_{lat} and SM_{occ} terms. For simplicity, our model does not consider this effect.

3.1 Shared Memory Overhead

Our performance model accounts for two types of shared memory overhead: latency and occupancy. Figure 5 shows a simple

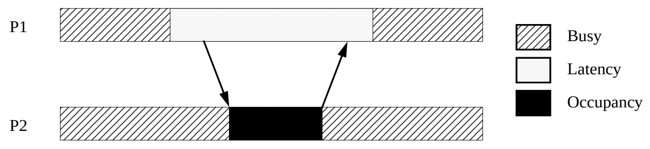

Figure 5: An application running on a software shared memory system will incur two types of shared memory overhead: latency and occupancy.

shared memory transaction that illustrates these two types of overhead. The figure shows activity on two separate processors, $P1$ and $P2$. Initially, both processors are executing application code. Then, processor $P1$ performs a shared memory access that requires service from processor $P2$ in software (for instance, $P2$ may be the home node for a needed page). To satisfy this request, $P1$ sends a message to $P2$ which invokes a software shared memory handler on $P2$. The handler runs and eventually sends a message back to $P1$ that satisfies the request. The transaction completes when $P1$ returns to the application code. The time during which $P1$ initiates the shared memory transaction and waits for remote service constitutes latency overhead, and the time during which $P2$ services the remote shared memory handler constitutes occupancy overhead. Both take cycles away from application code.

Shared memory latency, SM_{lat}, is simply the product of the total number of pages communicated by the application and the amount of latency incurred per page:

$$SM_{lat} = (\#\ \ pages)\,(latency\ \ per\ \ page) \tag{2}$$

The total page volume is computed using synchronization analysis, as described in Section 2. The latency per page is the number of cycles incurred by communicating a page of data across a release $\rightarrow$ acquire dependence through the software shared memory layer, an overhead that is system dependent. In this study, we will assume the system in [5], called MGS. Every page communicated across a release $\rightarrow$ acquire dependence in MGS can suffer three shared memory operations that require software service: page fault, upgrade fault, and release. The page fault initially brings the page from a remote memory node to the requesting processor's node. If the initial memory request that invoked the page fault was a load, then any subsequent store performed on data in the page will suffer an upgrade fault to upgrade the page from read privilege to write privilege. Finally, when the processor is done modifying the page, it will perform a release making all its updates visible to other processors.

In addition to the three shared memory operations described above, any other processor in the same node that wishes to access data in the page will suffer a TLB fault in order to map the page. This corresponds to the sharing pattern in which a subsequent release $\rightarrow$ acquire dependence occurs between processors on the same DSM node (see Section 2.2). Notice, though, that no internode communication is required for this sharing pattern since the initial page fault performs all the necessary inter-node communication on behalf of the entire DSM node. Table 1 shows the latencies for the three page-level overheads as well as the overhead of a TLB fault as measured on the MGS system. The current version of MGS available runs on the Alewife multiprocessor [13] which clocks at 20 MHz. All numbers are reported in Alewife cycles.

Event	Latency
TLB Fault	2288
Page Fault	32323
Upgrade Fault	12441
Release	9992

Table 1: Software shared memory transaction latencies to move a page across a release $\rightarrow$ acquire dependence on the MGS system. All values are in cycles at 20 MHz.

Event	Memory	Client
TLB Fault	0	0
Page Fault	3608	6390
Upgrade Fault	150	0
Release	2803	0

Table 2: Software shared memory transaction occupancies to move a page across a release $\rightarrow$ acquire dependence on the MGS system. All values are in cycles at 20 MHz.

Shared memory occupancy, SM_{tot_occ}, can be computed in a similar fashion as latency. The total amount of occupancy overhead is again the product of two terms.

$$SM_{tot\_occ} = (\#\ \ pages)\,(occupancy\ \ per\ \ page) \tag{3}$$

The first term in Equation 3 is identical to the first term in Equation 2, computed using synchronization analysis. The second term in Equation 3 is the total occupancy, in cycles, incurred to communicate a page of data across a release $\rightarrow$ acquire dependence. For each shared memory transaction whose latency is listed in Table 1, there is a corresponding occupancy overhead charged to those processors that must execute the software shared memory handlers involved in each shared memory transaction. Table 2 lists these occupancy overheads. The occupancy numbers have been broken down into two categories: those that occur on the home node for the page in question, labeled “Memory,” and those that occur on 3rd-party remote clients, labeled “Client.” Again, all overheads were measured on the MGS system. TLB faults are purely local operations handled entirely by the faulting processor; therefore, there is no occupancy overhead associated with TLB faults. The other three shared memory operations all invoke handlers on the home node. In addition, the page fault handler must perform invalidation on the remote client that owns the most recent copy in order to obtain a coherent copy of the page (see [14] for more details).

For simplicity, we assume the total occupancy, SM_{tot_occ}, is distributed evenly across all processors in the system. This is only an approximation. In reality, the occupancy overhead may not be distributed equally across processors either because a few processors are the home node for a disproportionate number of pages, or because sharing on a few pages is disproportionately more frequent compared to other pages.

While the expression in Equation 3 yields the total occupancy overhead, not all of the occupancy overhead contributes to the application's critical path. It is possible that when a handler arrives, the processor being interrupted is not performing useful work, but is instead waiting on a remote software transaction. In this case, the handler's execution will be hidden and will not dilate execution time. To account for this effect, we scale SM_{tot_occ} by the proba-

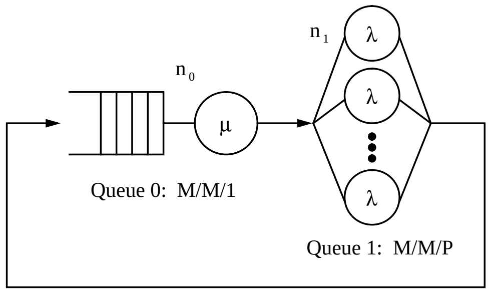

Figure 6: Modeling lock contention using a closed queuing network.

bility that a handler will actually interrupt useful work. This is the effective occupancy overhead, SM_{occ}.

$$SM_{occ} = \left(\frac{R_{hw} + SM_{occ}}{R_{hw} + SM_{occ} + SM_{lat}}\right) SM_{tot\_occ} \qquad (4)$$

In Equation 4, R_{hw} is the parallel runtime without any software overheads, as given in Equation 1. The fraction is the ratio of time spent doing useful work to the total execution time; this is the probability that a handler will interrupt useful work. Notice we assume that interrupting a handler already in progress will contribute to the application's critical path–that is why SM_{occ} appears in the numerator as well as the denominator. Also, we don't account for those cases when a handler partially contributes to the critical path. We assume for simplicity that a handler either interrupts useful work, or the handler interrupts an idle processor and its occupancy is completely hidden. Solving Equation 4 for SM_{occ} yields the effective occupancy seen by the application.

3.2 Synchronization Overhead

Our performance model accounts for two types of synchronization overhead: contention at locks, and load imbalance at barriers. While these overheads impact cache-coherent machines as well as multigrain systems, their effects are typically more severe on multigrain systems due to software shared memory. In this section, we account for lock contention and barrier load imbalance caused specifically by software shared memory.

Lock contention is the serialization of multiple lock operations performed simultaneously on a single lock variable. In a software shared memory system, it can be particularly severe due to *critical section dilation* [14]. When a processor successfully obtains a lock and enters a critical section, it may access one or more shared memory locations that invoke software support. Especially for those critical sections that involve very little computation, the introduction of expensive software shared memory overheads can significantly lengthen the duration for which the lock is held by the processor, thus leading to increased lock contention. For example, in the Water benchmark which we will study in Section 4, lock contention is not a significant problem for hardware cache-coherent machines, but on multigrain systems, it accounts for as much as 40% of overall execution time.

We model lock contention due to critical section dilation using the closed queuing network in Figure 6. The queuing network consists of two queues, an $M/M/1$ queue (Queue 0) and an $M/M/s$ queue (Queue 1) where $s = P$, the total number of processors in the system. Customers represent processors, so there are a total of P customers in the network which remains constant because the network is closed. A customer entering Queue 0 represents a processor trying to acquire a lock. If the queue is empty, then the customer enters the server immediately, corresponding to a successful lock acquire. If however the queue is busy, then the customer must wait. The customer remains in Queue 0 for as long as the processor holds the lock which is modeled as an exponential process with rate μ. When a customer leaves Queue 0, it immediately enters Queue 1 representing the release of the lock and the return to non-critical section code. We model the inter-lock acquire time using an exponential process with rate λ.

The queuing network in Figure 6 is known as a *Jackson network*. For the particular Jackson network we use, the probability mass function can be given as

$$p(n_0, n_1) = \frac{1}{G}\left(\frac{1}{\mu}\right)^{n_0}\left(\frac{1}{\lambda}\right)^{n_1}\frac{1}{n_1!} \qquad (5)$$

where G is a constant that normalizes the total cumulative mass to the value 1. From the probability mass function, we can compute the expectation of $n0$, the average number of customers waiting for the lock. Multiplying by the average lock service time, $\frac{1}{\mu}$, and the number of lock acquire operations yields the synchronization overhead due to lock contention, SYN_{lock}.

$$SYN_{lock} = (\#\ \ lock\ \ acquires)\left(\frac{1}{\mu}\right) E[n_0] \qquad (6)$$

The number of lock acquires, used in Equation 6, can be derived from synchronization analysis. The average lock service time, $\frac{1}{\mu}$, is computed by examining the average amount of software shared memory latency suffered inside a critical section. And the average inter-lock acquire time, $\frac{1}{\lambda}$, is computed by dividing the application execution time without software shared memory overhead by the number of lock acquires. See [14] for a more detailed discussion.

In addition to more severe lock contention, the overhead of load imbalance at barriers can be more severe in software shared memory systems as well. An imbalance in the number of shared memory operations performed will have a larger performance impact on software systems as compared against hardware cache-coherent machines because the cost for each shared memory operation is much more expensive in the software case. To account for such load imbalance caused by software shared memory, our performance model computes the average imbalance in the number of release $\rightarrow$ acquire dependences across processors during synchronization analysis. From this analysis, the model computes the extra software shared memory latency that arises due to the imbalance in release $\rightarrow$ acquire dependences. This extra software shared memory latency due to load imbalance at barriers is the SYN_{bar} term in Equation 1. We do not, however, account for load imbalance in shared memory occupancy overhead.

4 Results

This section reports the results of an in-depth study on the scalability of multigrain systems using our performance model. We

App	Problem Size	R32	S32
Jacobi	1k X 1k grid	51.24	30.0
	4k X 4k grid		
Water	1k molecules, 1 iteration	89.67	28.3
	2k molecules, 1 iteration		
Water-tiled	1k molecules, 1 iteration	86.65	28.5
	2k molecules, 1 iteration		
TSP	10 cities	3.14	17.6
	18 cities		
Unstructured	2800 nodes, 17377 edges	13.44	15.2
	50653 nodes, 273165 edges		

Table 3: Application summary. The five columns list the application name, problem sizes studied, and running time (R32) and speedup (S32) on 32 processors, respectively.

	Error					
App	1	2	4	8	16	Total
Jacobi	-1.52	-1.11	-2.12	-2.32	-2.4	1.96
Water	16.4	-11.5	-13.8	-21.5	-16.0	16.2
Water-tiled	1.24	3.12	-1.3	-3.52	-3.58	2.76
TSP	-24.1	-25.2	-1.26	3.96	-16.1	17.3
Unstructured	-10.5	-25.7	-23.1	-23.8	-18.7	21.1

Table 4: Performance model validation. The five columns labeled "Error" report the percentage discrepancy between the model and measurements taken on MGS assuming a node size of 1 - 16 in powers of 2. The column labeled "Total" reports the root-mean-square average of the error columns.

describe the applications used in the study and the validation of our model in Section 4.1. Section 4.2 presents the applications' performance on multigrain systems with 512 processors. Finally, Section 4.3 studies performance-equivalent multigrain systems.

4.1 Applications and Validation

For our scalability study, we use 5 shared memory applications: Jacobi, Water, Water-tiled, TSP, and Unstructured. Jacobi performs an iterative relaxation over a two-dimensional grid. Water, from the SPLASH-I benchmark suite [15], is a molecular dynamics code that simulates the motion of water molecules in three-dimensional space. For Water, we only consider the force computation phase which accounts for most of the parallel runtime. Water-tiled is identical to Water except we perform a loop tiling transformation that improves locality. TSP is the traveling salesman problem that uses a branch and bound algorithm and a distributed work queue to dynamically load balance work across the machine. Finally, Unstructured [16] is a computation over an unstructured mesh. Again, as in Water, we only consider a single phase of the computation (loops that compute values across mesh edges) where the application spends a significant amount of time.

Table 3 summarizes the applications. The second column specifies the different problem sizes used in our study. For each application, the first problem size listed is the baseline problem used to validate the predictions of our model. The second problem size is the one used for the scaling study. Since the scaling study examines machines that are up to 16 times larger than the machine size used for validation (512 processors as compared to 32 processors), the second problem size was chosen such that sequential running time increases by a factor of 16 relative to the baseline problem. The last two columns, labeled "R32" and "S32," report the running time (in millions of cycles) and speedup, respectively, of the baseline problem size on a 32-node Alewife machine (all-hardware shared memory).

The accuracy of our performance model was carefully validated against measurements taken on the MGS system. MGS is an experimental multigrain shared memory system that runs on the Alewife multiprocessor. Two features of the MGS system make it attractive for our study. First, the software DSM layer in MGS is optimized for multigrain systems. It identifies pages that are shared exclusively by processors within a single DSM node and relaxes coherence management for these pages such that software overhead is completely eliminated. Each page is brought back to normal coherence management when the exclusive sharing pattern is violated.[2] Second, MGS permits arbitrary reconfiguration of node size because the DSM nodes are emulated through *virtual clustering*. MGS runs on a single large Alewife machine. Virtual clustering controls what groups of processors are allowed to communicate using cache-coherent shared memory by disallowing address translation mappings across emulated nodes forcing communication between nodes to trap into software shared memory. While virtual clustering allows flexible node size configuration, its disadvantage is that communication between virtual clusters is simulated. MGS simulates a fixed inter-node communication latency of 30 μsec. Contention in the inter-node network that would increase this base latency in an actual multigrain system is not taken into account (see [5] and [14] for more details).

Table 4 reports the validation of our performance model against the MGS system. For each application, five runtime measurements were taken on MGS with 32 total processors, one for each possible node size (1, 2, 4, 8, and 16 processors). The middle columns of Table 4, labeled "Error," report the agreement between the prediction from our model and the five MGS measurements. The last column of Table 4 reports the root-mean-square error for all five individual predictions. In almost every case, our model underestimates runtime, as indicated by the negative percentages. The undershoot is due to the inability of synchronization analysis to capture false sharing communication, and the lack of a model for serialization of software shared memory handlers. Despite these deficiencies in the model, however, the model agreement is within 18% of measured runtime for four applications, and within 22% for all five.

4.2 Performance of Large Multigrain Systems

In this section, we report node size scaling results for our applications when machine size is fixed at 512 processors. Figures 7 and 8 show the normalized parallel runtime and speedup, respectively, for all five of our applications. To obtain the R_{hw} value needed by the model, we interpolated the runtimes measured on the 32-processor MGS system to obtain the 512-processor runtimes. In both figures, we scale node size from 1 to 512 processors in powers of 2. The smallest node size corresponds to all-software shared memory systems since each node is a uniprocessor, while the largest node size corresponds to all-hardware cache-coherent machines since node size equals machine size. In Figure 7, each runtime is normalized against the parallel runtime on a 512-processor hardware cache-coherent machine running the same application.

[2] This feature of MGS was taken into consideration when computing the overheads for communicating a page across a synchronization dependence, as discussed in Section 3.1.

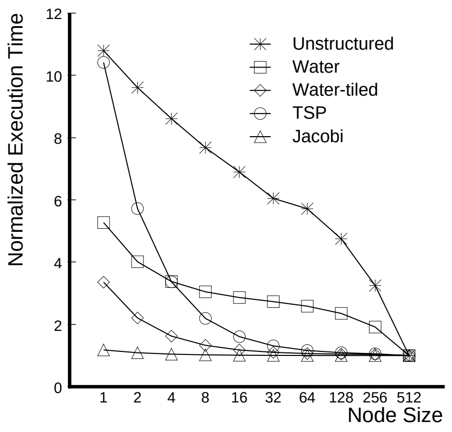

Figure 7: Normalized execution time versus node size. Total machine size is fixed at 512 processors.

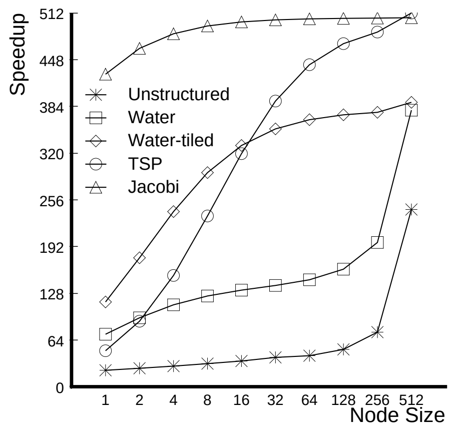

Figure 8: Speedup versus node size. Total machine size is fixed at 512 processors.

The application with the best performance is Jacobi. The normalized parallel runtime at every node size in Figure 7 is very close to 1 indicating that Jacobi's performance is largely insensitive to the relative amounts of hardware and software shared memory. For Jacobi, software-supported shared memory, while expensive, is adequate since communication happens so infrequently (coarse-grain sharing). This result is confirmed by Figure 8 which shows that Jacobi achieves high speedups across all node sizes.

At the other extreme, both Water and Unstructured have extremely demanding communication requirements. Both of these applications exhibit frequent write sharing that results in significant software coherence overhead in multigrain systems. As Figure 7 illustrates, performance is poor, particularly at small node sizes. As node size is increased, an increasing amount of the write sharing is supported by hardware cache coherence provided on the node resulting in performance gains. In addition, Unstructured suffers from load imbalance at barriers. As software shared memory overheads are mitigated by increasing node size, so are the effects that software shared memory has on load imbalance. For these difficult applications, node size scaling provides significant performance gains. However, even at very large node sizes (for instance, 256 processors), the all-hardware cache-coherent machine still outperforms the multigrain system by a factor of 2 for Water and by greater than a factor of 3 for Unstructured. Figure 8 shows the same qualitative result–the speedups achieved for Water and Unstructured on the multigrain systems are significantly lower than those achieved on the hardware cache-coherent machine, though Water achieves reasonable speedups at moderate node sizes.[3]

Finally, TSP and Water-tiled can achieve high performance on multigrain systems, but they exhibit *some* fine-grain sharing that requires modest hardware support. TSP requires fine-grain mechanisms to initially distribute work through the work queue data structure. In TSP, each node has a local work queue that is accessed using hardware shared memory, and there is a single global work queue that is accessed through software shared memory. When the application begins, all the work is placed on the global queue, so severe lock contention (due to critical section dilation) occurs as all the nodes compete for exclusive access to the global queue. Eventually, work is spread across the nodes and distribution to individual processors happens efficiently through hardware shared memory using the local queues. Scaling node size relieves some of the burden of the initial work distribution from the global queue leading to higher performance. In Water-tiled, a loop tiling transformation has been manually applied to the basic Water application to increase locality. Processors performing computation on the same tile share the tile in a fine-grain manner. Processors working on separate tiles communicate only when the work on an entire tile has been completed and a new tile is selected. Furthermore, the tiling transformation picks a tile size to match the size of each node so that fine-grain sharing is confined within nodes and uses hardware shared memory, while only the less frequent communication of entire tiles uses software shared memory. As node size is increased, the computation-to-communication ratio for each tile increases as well.

TSP and Water-tiled both exhibit *clustered fine-grain sharing*; therefore, they make good use of hardware shared memory provided within small nodes. As Figure 7 illustrates, without any hardware shared memory, these applications perform poorly. However, beyond a node size of 8 or 16 processors, their performance closely matches the all-hardware cache-coherent machine. For these applications, a little hardware support goes a long way.

4.3 Performance-Equivalence Results

Figures 9 through 13 report the model predictions for our five applications when both node size and machine size are scaled simultaneously. Each graph presents a slice of the machine space depicted in Figure 2 between machine sizes of 32 and 512 processors. As in Figure 2, we plot node size along the Y-axis and machine size along the X-axis. The machine space is bounded from above by all-hardware shared memory machines, from below by all-software shared memory machines, and the rest are multigrain systems (see Section 1.1). Dots have been placed inside the machine space at the intersection of each node and machine size to indicate those architectures that were evaluated by our model. Finally, contours have

[3] For instance, at node sizes of 16 processors and higher, Water achieves a speedup that exceeds 128.

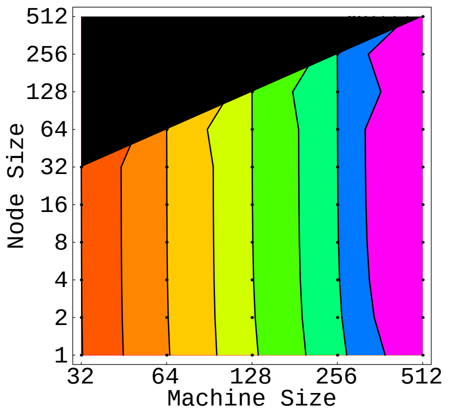

Figure 9: Jacobi.

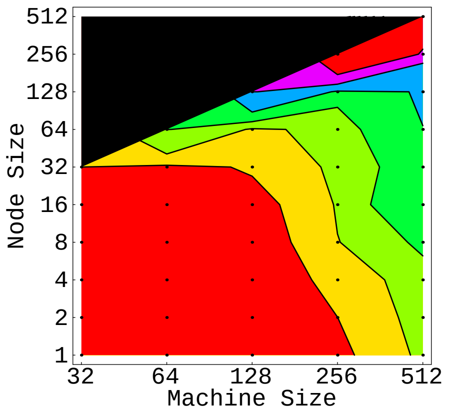

Figure 11: Unstructured.

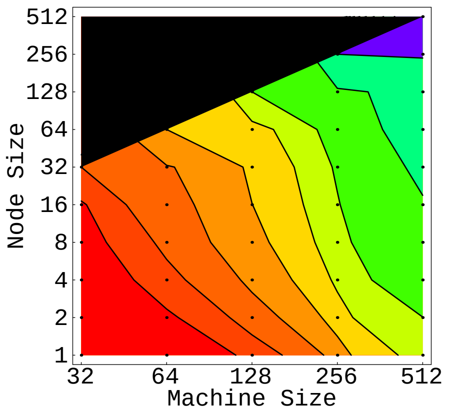

Figure 10: Water.

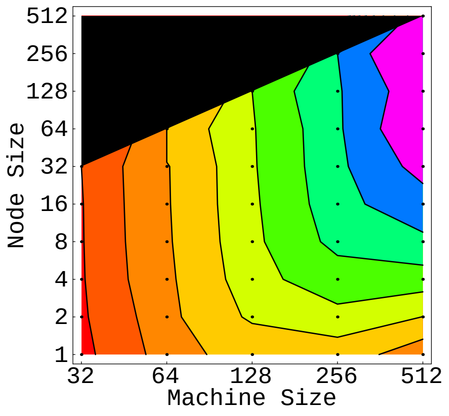

Figure 12: TSP.

been drawn through the machine space to indicate those machines that deliver equivalent performance on each application. The spacing between contours has been chosen such that every 2nd contour represents a factor of two in performance, increasing from the origin to the upper-right corner of each graph.

In Figure 9, the insensitivity of Jacobi performance to the underlying shared memory implementation as observed in Section 4.2 manifests itself in the form of vertical contours. The vertical contours imply that at any machine size, the choice of node size is irrelevant–Jacobi will perform well independent of this choice.

Figures 10 and 11 show the performance-equivalence results for Water and Unstructured, respectively. As noted in Section 4.2, these applications have significant communications requirements. For Water, the contours are diagonal indicating that performance is highly sensitive to node size since at any given machine size, scaling node size will cross several contours. The diagonal contours also imply that the performance lost by using smaller nodes can be compensated by building a larger machine. For instance, our model predicts that for Water, a 128-processor all-hardware shared memory machine has equivalent performance to a 256-processor multigrain system built using 16-way nodes. Overall, the results for Water suggest that a factor of two increase in machine size is roughly equivalent to a factor of eight reduction in node size. The contours for Unstructured are close to horizontal except at very large machine sizes, reflecting the fact that all-hardware machines significantly outperform the multigrain systems on this difficult application.

Finally, Figures 12 and 13 show the results for TSP and Water-tiled. The clustered fine-grain sharing exhibited by these applications manifest themselves in the "bending" of the contours. As long as node size is large enough (*i.e.* provide enough hardware shared memory), the contours are vertical indicating that the multigrain systems are competitive with the hardware shared memory machines. Once node size drops below a certain threshold (which increases with machine size), then the contours bend and flatten indicating a degradation in performance. The effect is particularly pronounced in TSP.

5 Conclusion

We believe the modeling work presented in this paper demonstrates that accurate performance prediction, which is crucial for exploring the large systems needed to conduct scalability studies, is feasible for software DSM systems. Our model predicts application runtime within 18% of experimentally measured values for four of our applications, and within 22% for all five. We believe these accuracies

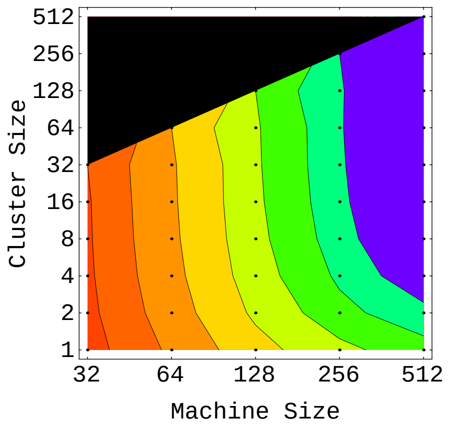

Figure 13: Water-tiled.

are adequate for the purpose of scalability evaluation.

In addition, several results were obtained regarding the scalability of multigrain systems. First, we find that multigrain systems universally outperform all-software systems at all machine sizes. Our conclusion is that multigrain systems offer much better scalability than conventional software DSMs built from uniprocessor workstation nodes. Second, we find that on difficult fine-grain applications, multigrain systems cannot match all-hardware machines in absolute performance. As we saw on the Water and Unstructured workloads for 512 processors, all-hardware machines provide between a factor of 2 or 3 in performance over the highest performing multigrain systems. Therefore, we conclude that all-hardware machines exhibit superior scalability. However, multigrain systems can closely match the absolute performance of all-hardware machines on applications with clustered fine-grain sharing patterns, though such applications may require compiler-assisted optimizations to improve locality (*e.g.* Water-tiled). Third, on difficult applications, multigrain systems may provide enough performance to be competitive with slightly smaller all-hardware machines. For instance, on the Water workload, we find that a 256-processor multigrain system built using 16-way nodes performs roughly equivalent to a 128-processor hardware cache-coherent machine. The larger multigrain system may have lower cost and thus better cost-performance because it uses commodity small-scale nodes. Unfortunately, on the most difficult application, Unstructured, the performance discrepancy is so large that multigrain systems are unlikely to deliver better cost-performance than all-hardware machines. Finally, our results show that the requisite node size scaling that allows multigrain systems to be competitive with all-hardware machines in those instances mentioned above is modest, even when total machine size is scaled to 512 processors. On the TSP and Water-tiled workloads, large-scale multigrain systems are competitive with all-hardware machines when node size is as small as 16 processors. We conclude that multigrain systems demonstrate scalability even without significant node size scaling.

References

[1] Rudrajit Samanta, Angelos Bilas, Liviu Iftode, and Jaswinder Pal Singh. Home-Based SVM Protocols for SMP Clusters: Design and Performance. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, Las Vegas, NV, February 1998. IEEE.

[2] Robert Stets, Sandhya Dwarkadas, Nikolaos Hardavellas, Galen Hunt, Leonidas Kontothanassis, Srinivasan Parthasarathy, and Michael Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, October 1997. ACM.

[3] Daniel J. Scales, Kourosh Gharachorloo, and Anshu Aggarwal. Fine-Grain Software Distributed Shared Memory on SMP Clusters. In *Proceedings of the 4th IEEE Symposium on High-Performance Computer Architecture*, Las Vegas, NV, February 1997.

[4] Andrew Erlichson, Neal Nuckolls, Greg Chesson, and John Hennessy. SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory. In *Proceedings of the Seventh ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 210–221, Cambridge, Massachusetts, October 1996. ACM.

[5] Donald Yeung, John Kubiatowicz, and Anant Agarwal. MGS: A Multigrain Shared Memory System. In *Proceedings of the 1996 International Symposium on Computer Architecture*, Philadelphia, May 1996.

[6] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[7] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th Annual Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[8] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 13–21, Gold Coast, Australia, May 1992.

[9] Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, and Willy Zwaenepoel. Software Versus Hardware Shared-Memory Implementation: A Case Study. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 106–117, Chicago, IL, April 1994.

[10] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings 17th Annual International Symposium on Computer Architecture*, New York, June 1990. IEEE.

[11] Pete Keleher, Alan Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. *Proceedings of the 1994 Usenix Conference*, pages 115–131, January 1994.

[12] Kirk Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain, Colorado, December 1995.

[13] Anant Agarwal et. al. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, June 1995.

[14] Donald Yeung. Multigrain Shared Memory. MIT-LCS TR-743, Massachussetts Institute of Technology, February 1998.

[15] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-92-526, Stanford University, June 1992.

[16] Shubu Mukherjee, Shamik Sharma, Mark Hill, Jim Larus, Anne Rogers, and Joel Saltz. Efficient Support for Irregular Applications on Distributed-Memory Machines. In *Proceedings of the 5th Annual Symposium on Principles and Practice of Parallel Programming*, pages 68–79. ACM, July 1995.