An Approach to Scalability Study of Shared Memory Parallel Systems*

Anand Sivasubramaniam
Aman Singla
Umakishore Ramachandran
H. Venkateswaran

Technical Report GIT-CC-93/62
October 1993

College of Computing
Georgia Institute of Technology
Atlanta, Ga 30332-0280
Phone: (404) 894-5136
FAX: (404) 894-9442
e-mail: rama@cc.gatech.edu

Abstract

The overheads in a parallel system that limit its scalability need to be identified and separated in order to enable parallel algorithm design and the development of parallel machines. Such overheads may be broadly classified into two components. The first one is intrinsic to the algorithm and arises due to factors such as the work-imbalance and the serial fraction. The second one is due to the interaction between the algorithm and the architecture and arises due to latency and contention in the network. A top-down approach to scalability study of shared memory parallel systems is proposed in this research. We define the notion of *overhead functions* associated with the different algorithmic and architectural characteristics to *quantify* the scalability of parallel systems; we develop a method for separating the algorithmic overhead into a serial component and a work-imbalance component; we also develop a method for isolating the overheads due to network latency and contention from the overall execution time of an application; we design and implement an execution-driven simulation platform that incorporates these methods for quantifying the overhead functions; and we use this simulator to study the scalability characteristics of five applications on shared memory platforms with different communication topologies.

Key words: Scalability, Shared Memory Multiprocessors, Execution-driven Simulation, Application-driven Studies, Latency, Contention

*This work has been funded in part by NSF grants MIPS-9058430 and MIPS-9200005, and an equipment grant from DEC.

1 Introduction

Scalability is a notion frequently used to signify the "goodness" of parallel systems[1]. A good understanding of this notion may be used to: select the best architecture platform for an application domain, predict the performance of an application on a larger configuration of an existing architecture, determine inherent limitations in an application for exploiting parallelism, identify algorithmic and architectural bottlenecks in a parallel system, and glean insight on the interaction between an application and an architecture to understand the scalability of other application-architecture pairs. In this paper, we develop a framework for studying the inter-play between applications and architectures to understand their implications for scalability. Since real-life applications set the standards for computing, it is meaningful to use such applications for studying the scalability of parallel systems. We call such an application-driven approach as a *top-down approach to scalability study*. The main thrust of this approach is to identify the important algorithmic and architectural artifacts that impact the performance of a parallel system, understand the interaction between them, quantify the impact of these artifacts on the execution time of an application, and use these quantifications in studying the scalability of a parallel system.

The following are the main contributions of our work: we define the notion of *overhead functions* associated with the different algorithmic and architectural characteristics to quantify the scalability of parallel systems; we develop a method for separating the algorithmic overhead into a *serial* component and a *work-imbalance* component; we also develop a method for isolating the overheads due to network *latency* (the actual hardware transmission time in the network) and *contention* (the amount of time spent in the network waiting for a resource to become free) from the overall execution time of an application; we design and implement a simulation platform that incorporates these methods for quantifying the overhead functions; and we use this simulator to study the scalability of five applications on shared memory platforms with different communication topologies.

Several performance metrics such as speedup [2], scaled speedup [12], sizeup [30], experimentally determined serial fraction [14], and isoefficiency function [15] have been proposed for quantifying the scalability of parallel systems. While these metrics are extremely useful for tracking performance trends, they do not provide the information needed to understand the reason why an application does not scale well with an architecture. The overhead functions which we identify, separate, and quantify in this work, help us toward this end. We are not aware of any other work that separates these overheads, and believe that such a separation is very important for understanding the interaction between applications and architectures. The

[1]The term, parallel system, is used to denote an application-architecture combination.

growth of overhead functions will provide key insights for the scalability of a parallel system by suggesting application restructuring, as well as architectural enhancements.

There have been several performance studies that have addressed issues such as latency, contention and synchronization. The scalability of synchronization primitives supported by the hardware [3, 19], the limits on interconnection network performance [1, 21], and the performance of scheduling policies [33, 16] are examples of such studies undertaken over the years. While such issues are extremely important, it is appropriate to put the impact of these factors into perspective by considering them in the context of overall application performance. There are studies that use real applications to address specific issues like the effect of sharing in parallel programs on the cache and bus performance [11] and the impact of synchronization and task granularity on parallel system performance [6]. Cypher et al. [10], identify the architectural requirements such as floating point operations, communications, and input/output for message-passing scientific applications. Rothberg et al. [24] conduct a similar study towards identifying the cache and memory size requirements for several applications. However, there have been very few attempts at quantifying the effects of algorithmic and architectural interactions in a parallel system.

The work we present in this paper is part of a larger project which aims at understanding the significant issues in the design of scalable parallel systems using the above-mentioned top-down approach. In [29], we illustrated this approach for the scalability study of message-passing systems. In this paper, we conduct a similar study for shared memory systems. A companion paper [28] develops a framework using the overhead functions for studying machine abstractions and impact of locality on the performance of parallel systems.

The top-down approach and the overhead functions are elaborated in Section 2. Details of our simulation platform, SPASM (Simulator for Parallel Architectural Scalability Measurements), which implements these overhead functions are also discussed in this section. The characteristics of the five applications used in this study are discussed in Section 3, details of the three shared memory platforms are presented in Section 4, and the results of our study with their implications on scalability are summarized in Section 5. Concluding remarks are given in Section 6.

2 Top-Down Approach

In keeping with the RISC ideology in the evolution of sequential architectures, we would like to use *real world applications* in the performance evaluation of parallel machines. However, applications normally tend to contain large volumes of code that are not easily portable and a level of detail that is not very familiar

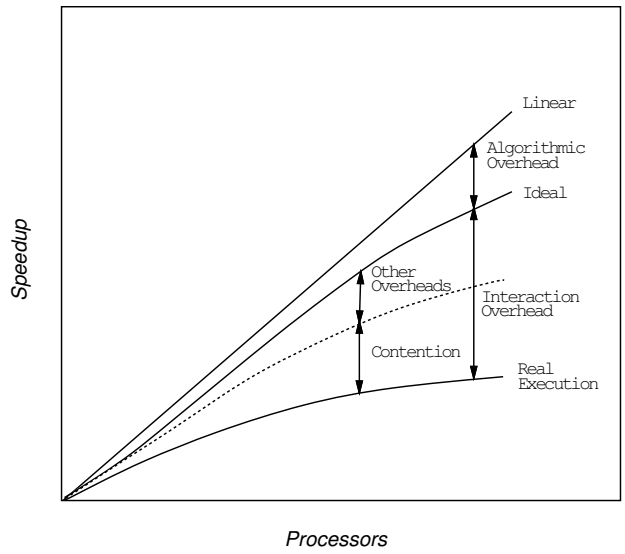

Figure 1: Top-down Approach to Scalability Study

to someone outside that application domain. Hence, computer scientists have traditionally used parallel algorithms that capture the interesting computation phases of applications for benchmarking their machines. Such abstractions of real applications that capture the main phases of the computation are called *kernels*. One can go even lower than kernels by abstracting the main *loops* in the computation (like the Lawrence Livermore loops [18]) and evaluating their performance. As one goes lower, the outcome of the evaluation becomes less realistic.

Even though an application may be abstracted by the kernels inside it, the sum of the times spent in the underlying kernels may not necessarily yield the time taken by the application. There is usually a cost involved in moving from one kernel to another such as the data movements and rearrangements in an application that are not part of the kernels that it is comprised of. For instance, an efficient implementation of a kernel may need to have the input data organized in a certain fashion which may not necessarily be the format of the output from the preceding kernel in the application. Despite its limitations, we believe that the scalability of an application with respect to an architecture can be captured by studying its kernels, since they represent the computationally intensive phases of an application. Therefore, we have used kernels in this study.

It is desirable to see a performance improvement (speedup) that is linear with the increase in the number of processors (as shown by the curve for linear behavior in Figure 1). With increasing number of processors, overheads in the parallel system increase (as shown by the curve for real execution in Figure 1) causing deviation from linear behavior. The overheads may even dominate the added computing power after a

certain stage resulting in potential slow-downs. Parallel system overheads may be broadly classified into a purely algorithmic component (*algorithmic overhead*), and a component arising due to the interaction of the algorithm and the architecture (*interaction overhead*). The algorithmic overhead is due to the inherent *serial* part [2] and the *work-imbalance* in the algorithm, and is independent of the architectural characteristics. Work imbalance could result with a differential amount of work done by the executing threads in a parallel phase. Isolating these two components of the algorithmic overhead would help in re-structuring the algorithm to improve its performance. Algorithmic overhead is the difference between the linear curve and that which would be obtained (the "ideal" curve in Figure 1) by executing the algorithm on an ideal machine such as the PRAM [32]. Such a machine idealizes the parallel architecture by assuming an infinite number of processors, and unit costs for communication and synchronization. A real execution could deviate significantly from the ideal execution due to the overheads such as latency, contention, synchronization, scheduling and cache effects. These overheads are lumped together as the interaction overhead. To fully understand the scalability of a parallel system it is important to isolate the influence of each component of the interaction overhead on the overall performance. For instance, in an architecture with no contention overhead, the communication pattern of the application would dictate the latency overhead incurred by it. Thus the performance of an application (on an architecture devoid of network contention) may lie between the ideal curve and the real execution curve (see Figure 1).

The key elements of our top-down approach for studying the scalability of parallel systems are as follows:

- experiment with real world applications
- identify parallel kernels that occur in these applications
- study the interaction of these kernels with architectural features to separate and quantify the overheads in the parallel system
- use these overheads as a way of predicting the scalability of parallel systems.

2.1 Implementing the Top-Down Approach

Scalability study of parallel systems is complex due to the several degrees of freedom that exist in them. Experimentation, simulation, and analytical models are three techniques that have been commonly used in such studies. But each has its own limitations. We adopted the first technique in our earlier work by experimenting with frequently used parallel algorithms on shared memory [27] and message-passing [26]

platforms. Experimentation is important and useful in scalability studies of existing architectures, but has certain limitations: first, the underlying hardware is fixed making it impossible to study the effect of changing individual architectural parameters; and second, it is difficult if not impossible to separate the effects of different architectural artifacts on the performance since we are constrained by the performance monitoring support provided by the parallel system. Further, monitoring program behavior via instrumentation can become intrusive yielding inaccurate results.

Analytical models have often been used to give gross estimates for the performance of large parallel systems. In general, such models tend to make simplistic assumptions about program behavior and architectural characteristics to make the analysis using the model tractable. These assumptions restrict their applicability for capturing complex interactions between algorithms and architectures. For instance, models developed in [17, 31, 9] are mainly applicable to algorithms with regular communication structures that can be predetermined before the execution of the algorithm. Madala and Sinclair [17] confine their studies to synchronous algorithms while [31] and [9] develop models for regular iterative algorithms. However, there exist several applications [24] with irregular data access, communication, and synchronization characteristics which cannot always be captured by such simple parameters. Further, an application may be structured to hide a particular overhead such as latency by overlapping computation with communication. It may be difficult to capture such dynamic program behavior using analytical models. Similarly, several other models make assumptions about architectural characteristics. For instance, the model developed in [20] ignores data inconsistency that can arise in a cache-based multiprocessor during the course of execution of an application and thus does not consider the coherence traffic on the network.

The main focus in our top-down approach is to quantify the overheads that arise in the interaction between the kernels and architecture and its impact on the overall execution of the application. It is not clear that these overheads can be easily modeled by a few parameters. Therefore, we use simulation for quantifying and separating the overheads.

All three techniques have a significant role to play in the scalability study of large parallel systems. We already observed some of the limitations of experimentation and modeling. Simulation also has its limitations. For example, it may not always be possible to predict system scalability with simulation owing to resource (time and space) constraints in using architectural simulators. But we believe that our simulation technique can be viewed as complementing the analytical one for the following reasons: it does not have the same limitations as the latter in that the datapoints obtained using it are closer to reality; but owing to resource constraints it may not be possible to simulate large systems. Therefore, simulation can be used to

obtain several datapoints for a parallel system, which can then be used as a feedback to refine the existing analytical models to predict the scalability of larger parallel systems. Further, our simulator can also be used to validate existing analytical models using real applications.

Our simulation platform (SPASM), to be presented in the next sub-section, provides an elegant set of mechanisms for quantifying the different overheads we discussed earlier. The algorithmic overhead is quantified by computing the time taken for execution of a given parallel program on an ideal machine such as the PRAM [32] and measuring its deviation from a linear speedup curve. Further, we separate this overhead into that due to the serial part (*serial overhead*) and that due to work imbalance (*work-imbalance overhead*). As we mentioned earlier, the interaction overhead should be separated into its component parts. We currently do not address scheduling overheads[2]. Processes in a parallel program often need to communicate during execution. This communication is explicit in message-passing systems via messages, while it is implicit in a shared memory system via accesses to shared variables. But regardless of the programming paradigm, communication involves network accesses and the physical limitations of the network tend to contribute to the overheads in the execution. These overheads may be broadly classified as latency and contention, and we associate an overhead function with each. The *Latency Overhead Function* ($f_L(p)$) is thus defined as the total amount of time spent by a processor waiting for messages due to the transmission time on the links and the switching overhead in the network assuming that the messages did not have to contend for any link. Likewise, the *Contention Overhead Function* ($f_C(p)$) is the total amount of time incurred by a processor due to the time spent waiting for links to become free by the messages. Synchronization and communication are intertwined in a message-passing system. On the other hand, in a shared memory system it is interesting to separate these two artifacts. Such systems normally provide some synchronization support which may either be as simple as an atomic read-modify-write operation, or may provide special hardware for more complicated operations like barriers and queue-based locks. While the latter may save execution time for complicated synchronization operations, the former is more flexible for implementing a variety of such operations. For reasons of generality, we assume that only the test&set operation is supported by shared memory systems. We also assume that the memory module (at which the operation is performed), is intelligent enough to perform the necessary operation in unit time. With such an assumption, the only network overhead due to the synchronization operation (test&set) is a roundtrip message, and the overheads for such a message are accounted for in the latency and contention overhead functions described earlier. The waiting time in a processor during synchronization operations is accounted for in the CPU time which

[2]We do not distinguish between the terms, *process*, *processor* and *thread*, and use them synonymously in this paper.

would manifest itself as an algorithmic (serial or work imbalance) overhead. Hence, for the rest of this paper, we confine ourselves to the the only two aspects of the interaction overhead that are germane to this study, namely, latency and contention.

Constant problem size (where the problem size remains unchanged as the number of processors is increased), *memory constrained* (where the problem size is scaled up linearly with the number of processors), and *time constrained* (where the problem size is scaled up to keep the execution time constant with increasing number of processors) are three well-accepted scaling models used in the study of parallel systems. Each model is appropriate depending on the nature of the study. Overhead functions can be used to study the growth of system overheads for any of these scaling strategies. In our simulation experiments, we limit ourselves to the constant problem size scaling model.

2.2 SPASM

SPASM is an execution-driven simulator written in CSIM. As with other recent simulators [5, 7, 23], the bulk of the instructions in the parallel program is executed at the speed of the native processor (SPARC in this study) and only the instructions that may potentially involve a network access are simulated. The reader is referred to [29] for a detailed description of the implementation of SPASM. The input parameters and output statistics provided by SPASM are given below.

2.2.1 Parameters

The system parameters that can be specified to SPASM are: the *number of processors (p)*, the *clock speed*, the *hardware setup time* for transmission of a message, the *hardware bandwidth*, the *software latency* for transmission of a message and the sustained *software bandwidth*.

2.2.2 Metrics

SPASM provides a wide range of statistical information about the execution of the program. It gives the *total time* (simulated time) which is the maximum of the running times of the individual parallel processors. This is the time that would be taken by an execution of the parallel program on the target parallel machine. *Speedup* using p processors is measured as the ratio of the total time on 1 processor to the total time on p processors.

Ideal time is the total time taken by a parallel program to execute on an ideal machine such as the PRAM. It includes the algorithmic overhead but does not include the interaction overhead. SPASM simulates an

ideal machine to provide this metric. As we mentioned in Section 2, the difference between the linear time and the ideal time gives the algorithmic overhead.

SPASM quantifies both the latency overhead function ($f_L(p)$) as well as the contention overhead function ($f_C(p)$) seen by a processor as described in Section 2. This is done by time-stamping messages when they are sent. At the time a message is received, the time that the message would have taken in a contention free environment is charged to the latency overhead function while the rest of the time is accounted for in the contention overhead function. Though not relevant to this study, it is worthwhile to mention that SPASM provides the latency and contention incurred by a message as well as the latency and contention that a processor may choose to see. Even though a message may incur a certain latency and contention, a processor may choose to hide all or part of it by overlapping computation with communication. Such a scenario may arise with a non-blocking message operation on a message-passing machine or with a prefetch operation on a shared memory machine. But for the rest of this paper (since we deal with blocking load/store shared memory operations), we assume that a processor sees all of the network latency and contention.

SPASM also provides statistical information about the network. It gives the utilization of each link in the network and the average queue lengths of messages at any particular link. This information can be useful in identifying network bottlenecks and comparing relative merits of different networks and their capabilities.

It is often useful to have the above metrics for different modes of execution of the algorithm. Such a breakup would help identify bottlenecks in the program, and also help estimate the potential gain in performance that may be possible through a specific hardware or software enhancement. SPASM provides statistics grouped together for system-defined as well as for user-defined modes of execution. The system-defined modes are:

- NORMAL: A program is in the NORMAL mode if it is not in any of the other modes. An application programmer may further define sub-modes if necessary.
- BARRIER: Mode corresponding to a barrier synchronization operation.
- MUTEX: Even though the simulated hardware provides only a test&set operation, mutual exclusion *lock* (implemented using test-test&set [3]) is available as a library function in SPASM. A program enters this mode during lock operations. With this mechanism, we can separate the overheads due to the synchronization operations from the rest of the program execution.
- PGM_SYNC: Parallel programs may use Signal-Wait semantics for pairwise synchronization. A lock is unnecessary for the Signal variable since only 1 processor writes into it and the other reads from it.

This mode is used to differentiate such accesses from normal load/store accesses.

The *total time* for a given application is the sum of the *execution times* for each of the above defined modes. The *execution time* for each program mode is the sum of the *computation time*, the *latency overhead* and the *contention overhead* observed in the mode. The metrics identified by SPASM quantify the algorithmic overhead and the interesting components of the interaction overhead. Computation time in the NORMAL mode is the actual time spent in local computation in an application. The sum of latency and contention overheads in the NORMAL mode is the actual time incurred for ordinary data accesses. For the BARRIER and PGM_SYNC modes, the computation time is the wait time incurred by a processor in synchronizing with other processors and results due the algorithmic work imbalance. The computation time in the MUTEX mode is the time spent in waiting for a lock and represents the serial part in an application arising due to critical sections. For the BARRIER and MUTEX modes, the computation time also includes the cost of implementing the synchronization primitive and other residual effects due to latency and contention for prior accesses. In all three synchronization modes, the latency and contention overheads together represent the actual time incurred in accessing synchronization variables.

3 Algorithmic Characteristics

This section briefly describes the characteristics of five kernels used in this study. Three of them (EP, IS and CG) are from the NAS parallel benchmark suite [4]; CHOLESKY is from the SPLASH benchmark suite [25]; and FFT is the well-known Fast Fourier Transform algorithm. The characteristics include the data access pattern, the synchronization pattern, the communication pattern, the computation granularity (the amount of work done) and data granularity (the amount of data communicated) for each phase of the program. EP and FFT are well-structured kernels with regular communication patterns determinable at compile-time, with the difference that EP has a higher computation to communication ratio. IS also has a regular communication pattern, but in addition it uses locks for mutual exclusion during the execution. CG and CHOLESKY are different from the other kernels in that their communication patterns are not regular (both use sparse matrices) and cannot be determined at compile time. While a certain number of rows of the matrix in CG is assigned to a processor at compile time (static scheduling), CHOLESKY uses a dynamically maintained queue of runnable tasks.

EP

EP is the "Embarrassingly Parallel" kernel that generates pairs of Gaussian random deviates and tabulates

Phase	Description	Comp. Gran.	Data Gran.	Synchronization
1	Local Float. Pt. Opns.	Large	N/A	N/A
2	Global Sum	Integer Add	Integer	Wait-Signal

Table 1: Characteristics of EP

the number of pairs in successive square annuli. This problem is typical of many Monte-Carlo simulation applications. It is computation bound and has little communication between processors. A large number of floating point random numbers is generated which are then subject to a series of operations. The computation granularity of this section of the code is considerably large and is linear in the number of random numbers (the problem size) calculated. A data size of 64K pairs of random numbers has been chosen in this study. The operation performed on a computed random number is completely independent of the other random numbers. The processor assigned to a random number can thus execute all the operations for that number without any external data. Hence the data granularity is meaningless for this phase of the program. Towards the end of this phase, a few global sums are calculated by using a logarithmic reduce operation. In step i of the reduction, a processor receives an integer from another which is a distance 2^i away and performs an addition of the received value with a local value. The data that it receives (data granularity) resides in a cache block in the other processor, along with the synchronization variable which indicates that the data is ready (synchronization is combined with data transfer to exploit spatial locality). Since only 1 processor writes into this variable, and the other spins on the value of the synchronization variable (the PGM_SYNC mode described in Section 2.2), no locks are used. Every processor reads the global sum from the cache block of processor 0 when the last addition is complete. The computation granularity between these communication steps can lead to work imbalance since the number of participating processors halves after each step of the logarithmic reduction. However since the computation is a simple addition it does not cause any significant imbalance for this kernel. The amount of local computation in the initial computation phase overshadows the communication performed by a processor. Table 1 summarizes the characteristics of EP.

IS

IS is the "Integer Sort" kernel that uses bucket sort to rank a list of integers which is an important operation in "particle method" codes. A list of 64K integers with 2K buckets is chosen for this study. An implementation of the algorithm is described in [22] and Table 2 summarizes its characteristics. The input list is equally partitioned among the processors. Each processor maintains two sets of buckets. One set of buckets (of size 2K) is used to maintain the information for the portion of the list local to it. The

Phase	Description	Comp. Gran.	Data Gran.	Synchronization
1	Local bucket updates	Small	N/A	N/A
2	Barrier Sync.	N/A	N/A	Barrier
3	Global bucket merge	Small	$chunk * (p-1)$ integers	N/A
4	Global Sum	Integer Add	Integer	Wait-Signal
5	Global bucket updates	Small	N/A	N/A
6	Barrier Sync.	N/A	N/A	Barrier
7	Global bucket updates	Small	2K integers	Lock each bucket
8	Local List Ranking	Small	N/A	N/A

Table 2: Characteristics of IS

other set (of size $chunk$ = 2K/p where p is the number of processors) maintains the information for the entire list. A processor first updates the local buckets for the portion of the list allotted to it, which is an entirely local operation (phase 1). Each list element would require an update (integer addition) of its corresponding bucket. A barrier is used to ensure the completion of this phase. The implementation of the barrier is similar to the implementation of the logarithmic global sum operation discussed in EP, except that no computation need be performed. A processor then uses the local buckets of every other processor to calculate the bucket values for the $chunk$ of the global buckets allotted to it (phase 3). The phase would thus require $chunk * (p-1)$ remote bucket values per processor. During this calculation, the processor also maintains the sum of all the global bucket values in its $chunk$. These sums are then involved in a logarithmic reduce operation (phase 4) to obtain the partial sum for each processor. Each processor uses this partial sum in calculating the partial sums for the $chunk$ of global buckets allotted to it (phase 5) which is again a local operation. At the completion of this phase, a processor sets a lock (test-test&set lock [3]) for each global bucket, subtracts the value found in the corresponding local bucket, updates the local bucket with this new value in the global bucket, and unlocks the bucket (phase 7). The memory allocation for the global buckets and its locks is done in such a way that a bucket and its corresponding lock fall in the same cache block and the rest of the cache block is unused. Synchronization is thus combined with data transfer and false sharing is avoided. The final list ranking phase (phase 8) is a completely local operation using the local buckets in each processor and is similar to phase 1 in its characteristics.

FFT

FFT is the one dimensional complex Fast Fourier Transform of N (64K for this study) points. N is a power of 2 and greater than or equal to the square of the number of processors p. The kernel is implemented in 3 main phases. In phases 1 and 5, processors perform the radix-2 butterfly computation on N/p local

Phase	Description	Comp. Gran.	Data Gran.	Synchronization
1	Local radix-2 butterfly	$O(\frac{N}{p}\log\frac{N}{p})$	N/A	N/A
2	Barrier Sync.	N/A	N/A	Barrier
3	Data redistribution	N/A	$(p-1) * \frac{N}{p^2}$complex numbers	N/A
4	Barrier Sync.	N/A	N/A	Barrier
5	Local radix-2 butterfly	$O(\frac{N}{p}\log p)$	N/A	N/A

Table 3: Characteristics of FFT

points. Phase 3 is the only communication phase in which the *cyclic* layout of data is changed to a *blocked* layout as described in [8]. It involves an all-to-all communication step where each processor distributes its local data equally among the p processors. The communication in this step is *staggered* with processor i starting with reading data ($\frac{N}{p^2}$ points) from processor $i+1$ and ending with reading data from processor $i-1$ in $p-1$ substeps. This communication schedule minimizes contention both in the network and at the processor ends. These three phases are separated by barriers.

CG

Phase	Description	Comp. Gran.	Data Gran.	Synchronization
1	Matrix-Vector Prod.	Medium	Random Float. Pt. Accesses	N/A
2	Vector-vector Prod.			
	a) Local dot product	Small	N/A	N/A
	b) Global Sum	Float. Pt. Add	Float. Pt.	WaitSignal
3	Local Float. Pt. Opns	Medium	N/A	N/A
4	<same as phase 2>			
5	Local Float. Pt. Opns	Medium	N/A	N/A
6	Barrier Sync.	N/A	N/A	Barrier

Table 4: Characteristics of CG

CG is the "Conjugate Gradient" kernel which uses the Conjugate Gradient method to estimate the smallest eigenvalue of a symmetric positive-definite sparse matrix with a random pattern of non-zeroes that is typical of unstructured grid computations. A sparse matrix of size 1400X1400 containing 100,300 non-zeroes has been used in the study. The sparse matrix and the vectors are partitioned by rows assigning an equal number of contiguous rows to each processor (static scheduling). We present the results for five iterations of the Conjugate Gradient Method in trying to approximate the solution of a system of linear equations. There is a barrier at the end of each iteration. Each iteration involves the calculation of a sparse matrix-vector product and two vector-vector dot products. These are the only operations that involve

communication. The computation granularity between these operations is linear in the number of rows (the problem size) and involves a floating point addition and multiplication for each row. The vector-vector dot product is calculated by first obtaining the intermediate dot products for the elements in the vectors local to a processor. This is again a local operation with a computation granularity linear in the number of rows assigned to a processor with a floating point multiplication and addition performed for each row. A global sum of the intermediate dot products is calculated by a logarithmic reduce operation (as in EP) yielding the final dot product. For the computation of the matrix-vector product, each processor performs the necessary calculations for the rows assigned to it in the resulting matrix (which are also the same rows in the sparse matrix that are local to the processor). But the calculation may need elements of the vector that are not local to a processor. Since the elements of the vector that are needed for the computation are dependent on the randomly generated sparse matrix, the communication pattern for this phase is random. Table 4 summarizes the characteristics for each iteration of CG.

CHOLESKY

Phase	Description	Comp. Gran.	Data Gran.	Synchronization
1	Get task	integer addition	few integers	mutex lock
2	Modify supernode	supernode size float. pt. ops.	supernode	N/A
3	Modify s supernodes (s is data dependent)	$s*$ supernode size float. pt. ops	s supernodes	locks for each column
4	Add task (if needed)	integer addition	few integers	lock

Table 5: Characteristics of CHOLESKY

This kernel performs a Cholesky factorization of a sparse positive definite matrix. The sparse nature of the input matrix results in an algorithm with a data dependent dynamic access pattern. The algorithm requires an initial symbolic factorization of the input matrix which is done sequentially because it requires only a small fraction of the total compute time. Only numerical factorization [25] is parallelized and analyzed. Sets of columns having similar non-zero structure are combined into supernodes at the end of symbolic factorization. Processors get tasks from a central task queue. Each supernode is a potential task which is used to modify subsequent supernodes. A *modifications_due* counter is maintained with each supernode. Thus each task involves fetching the associated supernode, modifying it and using it to modify other supernodes, thereby decreasing the *modifications_due* counters of supernodes. Communication is involved in fetching all the required columns to the processor working on a given task. When the counter for a supernode reaches 0, it is added to the task queue. Synchronization occurs in locking the task queue

when fetching or adding tasks, and locking columns when they are being modified. A 1806-by-1806 matrix with 30,824 floating point non-zeros in the matrix and 110,461 in the factor with 503 distinct supernodes is used for the study.

4 Architectural Characteristics

Since uniprocessor architecture is getting standardized with the advent of RISC technology, we fix most of the processor characteristics by using a 33 MHz SPARC chip as the baseline for each processor in a parallel system. Such an assumption enables us to make a fair comparison of the relative merits of the interesting parallel architectural characteristics across different platforms. Input-output characteristics are beyond the purview of this study.

We use three shared memory platforms with different interconnection topologies: the *fully connected network*, the *binary hypercube* and the *2-D mesh*. All three networks use serial (1-bit wide) unidirectional links with a link bandwidth of 20 MBytes/sec. The fully connected network models two links (one in each direction) between every pair of processors in the system. The cube platform connects the processors in a binary hypercube topology. Each edge of the cube has a link in each direction. The 2-D mesh resembles the Intel Touchstone Delta system. Links in the North, South, East and West directions, enable a processor in the middle of the mesh to communicate with its four immediate neighbors. Processors at corners and along an edge have only two and three neighbors respectively. Equal number of rows and columns is assumed when the number of processors is an even power of 2. Otherwise, the number of columns is twice the number of rows (we restrict the number of processors to a power of 2 in this study). Messages are circuit-switched and use a transmission scheme similar to the one used on the Intel iPSC/860 [13]. A circuit is set up between the source and the destination, and the message is then sent in a single packet. Message-sizes can vary upto 32 bytes. We assume that the switching time for setting up a circuit (in a contention free scenario) is negligible.

The simulated shared memory hierarchy is CC-NUMA (Cache Coherent Non-Uniform Memory Access). Each node in the system has a sufficiently large piece of the globally shared memory such that for the applications considered, the data-set assigned to each processor fits entirely in its portion of shared memory. There is also a 2-way set-associative private cache (64KBytes with 32 byte blocks) at each node that is maintained sequentially consistent using an invalidation-based fully-mapped directory-based cache coherence scheme.

5 Performance Results

In this section, we present the results from our simulation experiments showing the growth of the overhead functions with respect to the number of processors and their impact on scalability. The simulator allows one to explore the effect of varying other the system parameters such as link latency and processor speed on scalability. Since the main focus of this paper is an approach to scalability study, we have not dwelled on the scalability of parallel systems with respect to specific architectural artifacts to any great extent in this paper. We also briefly describe the impact of problem sizes on the system scalability for each kernel.

Figures 2, 3, 4, 5 and 6 show the “ideal" speedup curves (section 2) for the kernels EP, IS, FFT, CG and CHOLESKY, as well as the speedup curves for these kernels on the three hardware platforms. There is negligible deviation from the ideal curve for the EP kernel on the three hardware platforms; a marginal difference for FFT and CG; and a significant deviation for IS and CHOLESKY. For each of these kernels, we quantify the different interaction overheads responsible for the deviation during each execution mode of the kernel.

In the following subsections, we show for each kernel the execution time, the latency, and the contention overhead graphs for the mesh platform. The first shows the total execution time, while the latter two show the communication overheads ignoring the computation time. In each of these graphs, we show the curves for the individual modes of execution applicable for a particular kernel. We also present for each kernel the latency and contention overhead curves on the three architecture platforms. The latency overhead in the NORMAL mode (i.e. due to ordinary data access) is determined by the memory reference pattern of the kernel and the network traffic due to cache line replacement. With sufficiently large size cache at each node, it is reasonable to assume that this latency overhead is only due to the kernel, and thus is expected to be independent of the network topology. Due to the vagaries of the synchronization accesses, it is conceivable that the corresponding latency overheads could differ across network platforms for the other modes. However, in our experiments we have not seen any significant deviation. As a result, the latency overhead curves for all the kernels look alike across network platforms. On the other hand, it is to be expected that the contention overhead will increase as the connectivity in the network decreases. This is also confirmed for all the kernels.

5.1 EP

The communication time in this kernel is insignificant compared to the computation time. Figures 8 and 9, show the latency and contention respectively for the different modes of execution of EP. Despite the growth

of these overheads they are insignificant compared to the total execution time (which is dominated by the NORMAL mode), as can be seen in Figure 7.

Figures 10, 11 and 12 show the latency and contention overheads for the three hardware platforms. Since the number of communication events for global sums is logarithmic in the number of processors (see Section 3), the latency overhead curve exhibits a logarithmic behavior. Although there is a potential for contention overhead for the fully connected network due to message queueing at a node, the observed contention is marginal (Figure 10). With less connectivity, the contention overhead even dominates the latency overhead (Figures 11 and 12) with increasing number of processors. For instance, the cross-over point between contention and latency occurs at around 25 processors for the cube (Figure 11) and at around 16 processors for the mesh (Figure 12). Table 6 summarizes the overhead functions obtained by interpolating the datapoints from our simulation results.

EP	Full	Cube	Mesh
Comp. Time (s)	$3.6/p$	$3.6/p$	$3.6/p$
Latency (us)	$140.4 \log p$	$140.4 \log p$	$140.4 \log p$
Contention (us)	$Negligible$	$6.6p$	$13.7p$

Table 6: EP : Overhead Functions

As can be seen from Table 6, the constants associated with the computation time is much larger than those associated with latency and contention. Therefore, we can conclude that EP is a perfectly scalable kernel on all three hardware platforms. Even for a relatively small problem size (64K in this case), it would take nearly 1000 processors for the overheads to start dominating. While an increase in the problem size would increase the coefficient associated with the computation time, it does not have any effect on the coefficients of the other overhead functions. Hence, the scalability of EP for a larger problem size can only get better.

5.2 IS

For this kernel, there is a significant deviation from the ideal curve for all three platforms (see Figure 3). The overheads may be analyzed by considering the different modes of execution. In this kernel, NORMAL and MUTEX are the only significant modes of execution (see Figure 13). The network accesses in the NORMAL mode are for ordinary data transfer, and the accesses in MUTEX are for synchronization. The latency and contention overheads incurred in the MUTEX mode is higher than in the NORMAL mode (see

Figures 14 and 15). As a result of this, the total execution time in the MUTEX mode surpasses that in the NORMAL mode beyond a certain number of processors (see Figure 13), which also explains the dip in the speedup curve for mesh (see Figure 3).

Figures 16, 17 and 18. show the latency and contention overheads for the three hardware platforms. In IS, since every processor needs to access the local buckets of all other processors, and since the data is equally partitioned among the executing processors, the number of accesses to remote locations grows as $(p-1)/p$. This explains the flattening of the latency overhead curve for all three network platforms as p increases. On the mesh network the contention overhead surpasses the latency overhead at around 18 processors. Table 7 summarizes the overheads for IS obtained by interpolating the datapoints from our simulation results.

IS	Full	Cube	Mesh
Comp. Time (ms)	$129.3/p^{0.7}$	$129.3/p^{0.7}$	$129.3/p^{0.7}$
Latency (ms)	$13.2(1-1/p)$	$13.2(1-1/p)$	$13.2(1-1/p)$
Contention (ms)	$Negligible$	$4.0 \log p$	$0.9p$

Table 7: IS : Overhead Functions

Parallelization of this kernel increases the amount of work to be done for a given problem size (see Section 3). This inherent algorithmic overhead causes a deviation of the ideal curve from the linear curve (see Figure 3). This is also confirmed in Table 7, where the computation time does not decrease linearly with the number of processors. This indicates the kernel is not scalable for small problem sizes. As can be seen from Table 7, the contention overhead is negligible and the latency overhead converges to a constant with a sufficiently large number of processors on a fully connected network. Thus for a fully connected network, the scalability of this kernel is expected to closely follow the ideal curve. For the cube and mesh platforms, the contention overhead grows logarithmically and linearly with the number of processors, respectively. Therefore, the scalability of IS on these two platforms is likely to be worse than for the fully connected network. From the above observations, we can conclude that IS is not very scalable for the chosen problem size on the three hardware platforms. However, if the problem is scaled up, the coefficient associated with the computation time will increase thus making IS more scalable.

5.3 FFT

The algorithmic and interaction overheads for the FFT kernel are marginal. Thus the real execution curves for all three platforms as well as the ideal curve are close to the linear one as shown in Figure 4. The execution time is dominated by the NORMAL mode (Figure 19). The latency and contention overheads (Figures 20 and 21) incurred in this mode are insignificant compared to the total execution time, despite the growth of contention overhead with increasing number of processors.

The communication in FFT has been optimized as suggested in [8] into a single phase where every processor accesses the data of all the other processors in a skewed manner. The number of such non-local accesses incurred by a processor is grows as $O((p-1)/p^2)$ with the number of processors, and the latency overhead curves for all three networks reflect this behavior. As a result of skewing the communication among the processors, the contention is negligible on the full (Figure 22) and the cube (Figure 23) platforms. On the mesh (Figure 24), the contention surpasses the latency overhead at around 28 processors. Table 8 summarizes the overheads for FFT obtained by interpolating the datapoints from our simulation results.

FFT	Full	Cube	Mesh
Comp. Time (s)	$2.5/p$	$2.5/p$	$2.5/p$
Latency (ms)	$49.9/p^{0.9}$	$49.9/p^{0.9}$	$49.9/p^{0.9}$
Contention (us)	$Negligible$	$Small$	$63.5p$

Table 8: FFT : Overhead Functions

With marginal algorithmic overheads and decreasing number of messages exchanged per processor (latency overhead), the contention overhead is the only artifact that can cause deviation from linear behavior. But with skewed communication accesses, the contention overhead has also been minimized and begins to show only on the mesh network where it grows linearly (see Table 8). Thus we can conclude that the FFT kernel is scalable for the fully-connected and cube platforms. For the mesh platform, it would take 200 processors before the contention overhead starts dominating for the 64K problem size. With increase in problem size (N), the local computation that performs a radix-2 Butterfly is expected to grow as $O((N/p)\log(N/p))$ while the communication for a processor is expected to grow as $O(N(p-1)/p^2)$. Hence, increase in data size will increase its scalability on all hardware platforms.

5.4 CG

Interaction overheads for CG (Figure 5) cause a larger deviation from an ideal behavior than for EP but the difference is not as pronounced as in IS. The NORMAL mode is the only dominant mode of execution as depicted in Figure 25. The communication in the NORMAL mode for data accesses (the matrix-vector product) outweighs the overhead in accesses for synchronization variables during the BARRIER and PGM_SYNC modes (Figures 26 and 27). But the communication is still insignificant compared to the overall execution time for the range of processors considered.

In this kernel, since the input matrix is sparse, the fewer the rows assigned to a processor the fewer will be the number of elements of the vector that may need to be accessed for the matrix-vector product. Therefore, as the number of processors increases, the number of rows of the sparse matrix allocated to a processor decreases, thereby decreasing the likelihood of non-local memory references. Hence, the latency overhead decreases with an increase in the number of processors. The contention overhead increases from the full (Figure 28) to the cube (Figure 29) and surpasses the latency overhead for the mesh (Figure 30) at around 17 processors. Table 9 summarizes the overheads for CG obtained by interpolating the datapoints from our simulation results.

CG	Full	Cube	Mesh
Comp. Time (ms)	$571.2/p$	$571.2/p$	$571.2/p$
Latency (ms)	$2.9/p^{0.4}$	$2.9/p^{0.4}$	$2.9/p^{0.4}$
Contention (us)	$Negligible$	$8.1p$	$68.2p$

Table 9: CG : Overhead Functions

As can be seen from Table 9, the latency overhead decreases with increasing number of processors and the contention overhead is more pronounced. The contention overhead is negligible for the fully-connected network, grows linearly for the cube and the mesh with a larger coefficient for the mesh compared to the cube. We can thus conclude that the CG kernel is scalable for the fully-connected network and becomes less scalable for networks with lower connectivity like the cube and the mesh. The NORMAL mode consists of the two program phases, matrix-vector product and vector-vector product (see Section 3). Since the NORMAL mode dominates the total execution time, the scalability of the matrix-vector product would determine the scalability of the kernel. Scaling the problem size increases the number of non-local memory accesses linearly while increasing the amount of local computation quadratically. Thus an increase in problem size is likely to enhance scalability of CG on all three network platforms.

5.5 CHOLESKY

The algorithmic overheads for CHOLESKY cause a significant deviation from linear behavior for the ideal curve as shown in Figure 6. An examination of the execution times (Figure 31) shows that the bulk of the time is spent in the NORMAL mode which performs the actual factorization. The communication overheads in the NORMAL mode for the data accesses of the sparse matrix outweigh the accesses for synchronization variables (Figures 32 and 33). Thus the time spent in the MUTEX mode (which represents dynamic scheduling and accesses to critical sections) is insignificant compared to the NORMAL mode Although, the contention overhead in the NORMAL mode increases quite rapidly with the number of processors the overall impact of communication on the execution time is insignificant (see Figure 31).

As with CG and FFT, the number of non-local memory accesses made by a processor decreases with increasing number of processors explaining a decreasing latency overhead. The contention overhead is negligible for the fully-connected network (Figure 34) and grows with increasing processors for the cube (Figure 35), becoming more dominant than the latency overhead for the mesh (Figure 36) at around 20 processors. Table 10 summarizes the overheads for CHOLESKY obtained by interpolating the datapoints from our simulation results.

CHOLESKY	Full	Cube	Mesh
Comp. Time (s)	$3.9/p^{0.8}$	$3.9/p^{0.8}$	$3.9/p^{0.8}$
Latency (s)	$1.2/p^{0.9}$	$1.2/p^{0.9}$	$1.2/p^{0.9}$
Contention (ms)	$Negligible$	$Constant$	$39.9 \log p$

Table 10: CHOLESKY : Overhead Functions

The deviation of the ideal from the linear curve (Figure 6) indicates that the kernel is not very scalable for the chosen problem size due to the inherent algorithmic overhead as in IS. As can be observed from Table 10, the latency decreases with increasing number of processors and the scalability of the real execution would thus be dictated by the contention overhead. The contention on the fully-connected and cube networks is negligible thus projecting speedup curves that closely follow the ideal speedup curve for these platforms. On the other hand, the contention grows logarithmically on the mesh making this platform less scalable. With increasing problem sizes, the coefficient associated with the computation time in the above table is likely to grow faster than the coefficients associated with the communication overheads (verified by experimentation). Hence, an increase in problem size would enhance the scalability of this kernel on all hardware platforms.

6 Concluding Remarks

We used an execution-driven simulation platform to study the scalability characteristics of EP, IS, FFT, CG, and CHOLESKY on three shared memory platforms, respectively, with a fully-connected, cube, and mesh interconnection networks. The simulator allows for the separation of the algorithmic and interaction overheads in a parallel system. Separating the overheads provided us with some key insights into the algorithmic characteristics and architectural features that limit the scalability for these parallel systems. Algorithmic overheads such as the additional work incurred in parallelization could be a limiting factor for scalability as observed in IS and CHOLESKY. In shared memory machines with private caches, as long as the applications are well-structured to exploit locality, the key determinant to scalability is network contention. This is particularly true for most commercial shared memory multiprocessors which have sufficiently large caches.

We have illustrated the usefulness as well as the feasibility of our top-down approach. This approach can be used to study the impact of other system parameters (such as link bandwidth and processor speed) on scalability and provide guidelines for application design as well as evaluate architectural design decisions.

References

[1] Anant Agarwal. Limits on Interconnection Network Performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–412, October 1991.

[2] G. M. Amdahl. Validity of the Single Processor Approach to achieving Large Scale Computing Capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 483–485, April 1967.

[3] Thomas E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[4] D. Bailey et al. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications*, 5(3):63–73, 1991.

[5] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl. PROTEUS : A high-performance parallel-architecture simulator. Technical Report MIT-LCS-TR-516, Massachusetts Institute of Technology, Cambridge, MA 02139, September 1991.

[6] D. Chen, H. Su, and P. Yew. The Impact of Synchronization and Granularity on Parallel Systems. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 239–248, 1990.

[7] R. G. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair. The Rice parallel processing testbed. In *Proceedings of the ACM SIGMETRICS 1988 Conference on Measurement and Modeling of Computer Systems*, pages 4–11, Santa Fe, NM, May 1988.

[8] David Culler et al. LogP : Towards a realistic model of parallel computation. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, May 1993.

[9] Zarka Cvetanovic. The effects of problem partitioning, allocation, and granularity on the performance of multiple-processor systems. *IEEE Transactions on Computer Systems*, 36(4):421–432, April 1987.

[10] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina. Architectural requirements of parallel scientific applications with explicit communication. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 2–13, May 1993.

[11] Susan J. Eggers and Randy H. Katz. The Effect of Sharing on the Cache and Bus Performance of Parallel Programs. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–270, Boston, Massachusetts, April 1989.

[12] John L. Gustafson, Gary R. Montry, and Robert E. Benner. Development of Parallel Methods for a 1024-node Hypercube. *SIAM Journal on Scientific and Statistical Computing*, 9(4):609–638, 1988.

[13] Intel Corporation, Oregon. *Intel iPSC/2 and iPSC/860 User's Guide*, 1989.

[14] Alan H. Karp and Horace P. Flatt. Measuring Parallel processor Performance. *Communications of the ACM*, 33(5):539–543, May 1990.

[15] Vipin Kumar and V. Nageswara Rao. Parallel Depth-First Search. *International Journal of Parallel Programming*, 16(6):501–519, 1987.

[16] Scott T. Leutenegger and Mary K. Vernon. The Performance of Multiprogrammed Multiprocessor Scheduling Policies. In *Proceedings of the ACM SIGMETRICS 1990 Conference on Measurement and Modeling of Computer Systems*, pages 226–236, 1990.

[17] Sridhar Madala and James B. Sinclair. Performance of Synchronous Parallel Algorithms with Regular Structures. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):105–116, January 1991.

[18] F. H. McMahon. The Livermore Fortran Kernels : A Computer Test of the Numerical Performance Range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Livermore, CA, December 1986.

[19] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[20] Janak H. Patel. Analysis of multiprocessors with private cache memories. *IEEE Transactions on Computer Systems*, 31(4):296–304, April 1982.

[21] Gregory F. Pfister and V. Alan Norton. Hot Spot Contention and Combining in Multistage Interconnection Networks. *IEEE Transactions on Computer Systems*, C-34(10):943–948, October 1985.

[22] U. Ramachandran, G. Shah, S. Ravikumar, and J. Muthukumarasamy. Scalability study of the KSR-1. In *Proceedings of the 1993 International Conference on Parallel Processing*, pages I–237–240, August 1993.

[23] S. K. Reinhardt et al. The Wisconsin Wind Tunnel : Virtual prototyping of parallel computers. In *Proceedings of the ACM SIGMETRICS 1993 Conference on Measurement and Modeling of Computer Systems*, pages 48–60, Santa Clara, CA, May 1993.

[24] Edward Rothberg, Jaswinder Pal Singh, and Anoop Gupta. Working sets, cache sizes and node granularity issues for large-scale multiprocessors. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 14–25, May 1993.

[25] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-91-469, Computer Systems Laboratory, Stanford University, 1991.

[26] Anand Sivasubramaniam, Umakishore Ramachandran, and H. Venkateswaran. Message-Passing: Computational Model, Programming Paradigm, and Experimental Studies. Technical Report GIT-CC-91/11, College of Computing, Georgia Institute of Technology, February 1991.

[27] Anand Sivasubramaniam, Gautam Shah, Joonwon Lee, Umakishore Ramachandran, and H. Venkateswaran. Experimental Evaluation of Algorithmic Performance on Two Shared Memory Multiprocessors. In Norihisa Suzuki, editor, *Shared Memory Multiprocessing*, pages 81–107. MIT Press, 1992.

[28] Anand Sivasubramaniam, Aman Singla, Umakishore Ramachandran, and H. Venkateswaran. Machine Abstractions and Locality Issues in Studying Parallel Systems. Technical Report GIT-CC-93/63, College of Computing, Georgia Institute of Technology, October 1993.

[29] Anand Sivasubramaniam, Aman Singla, Umakishore Ramachandran, and H. Venkateswaran. A Simulation-based Scalability Study of Parallel Systems. Technical Report GIT-CC-93/27, College of Computing, Georgia Institute of Technology, April 1993.

[30] Xian-He Sun and John L. Gustafson. Towards a better Parallel Performance Metric. *Parallel Computing*, 17:1093–1109, 1991.

[31] D. F. Vrsalovic, D. P. Siewiorek, Z. Z. Segall, and E. Gehringer. Performance Prediction and Calibration for a Class of Multiprocessors. *IEEE Transactions on Computer Systems*, 37(11):1353–1365, November 1988.

[32] J. C. Wyllie. *The Complexity of Parallel Computations*. PhD thesis, Department of Computer Science, Cornell University, 1979.

[33] John Zahorjan and Cathy McCann. Processor Scheduling in Shared Memory Multiprocessors. In *Proceedings of the ACM SIGMETRICS 1990 Conference on Measurement and Modeling of Computer Systems*, pages 214–225, 1990.

Speedup Graphs

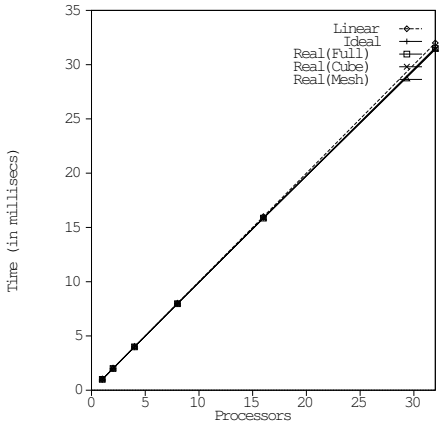

Figure 2: EP

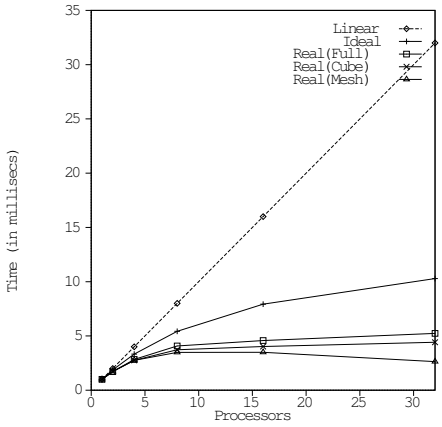

Figure 3: IS

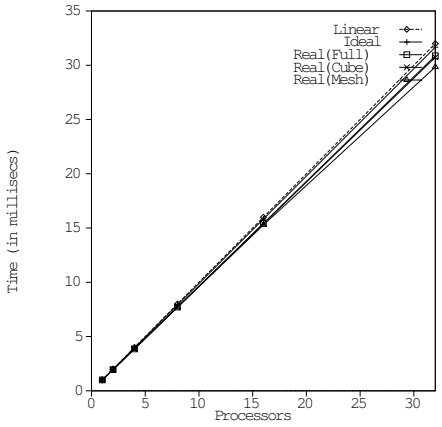

Figure 4: FFT

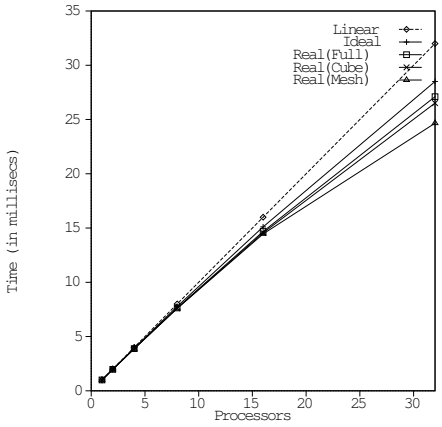

Figure 5: CG

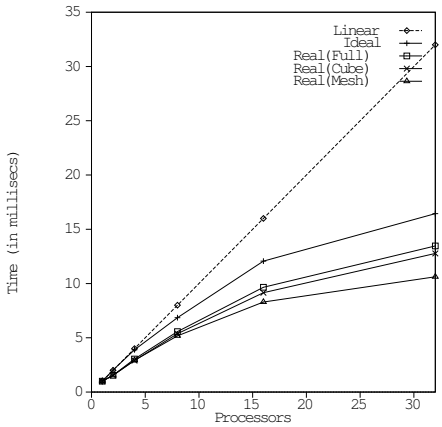

Figure 6: CHOLESKY

EP Graphs

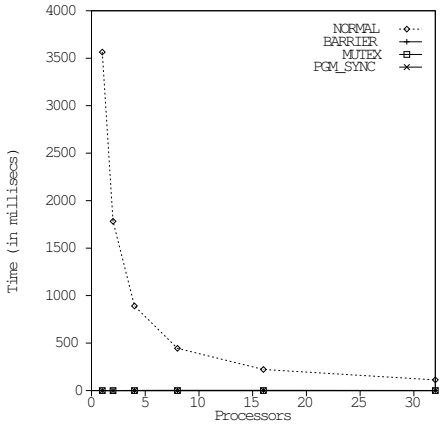

Figure 7: Mode-wise Execn. Time (Mesh)

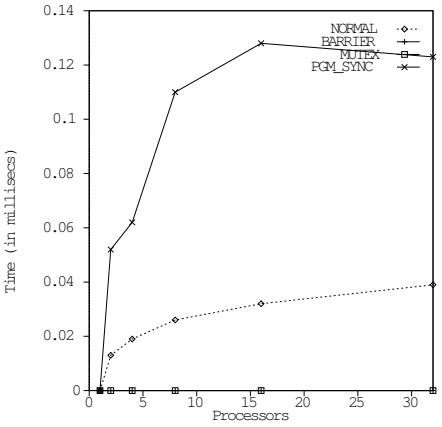

Figure 8: Mode-wise Latency (Mesh)

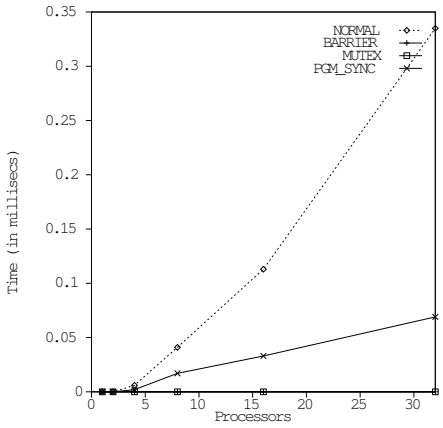

Figure 9: Mode-wise Contention (Mesh)

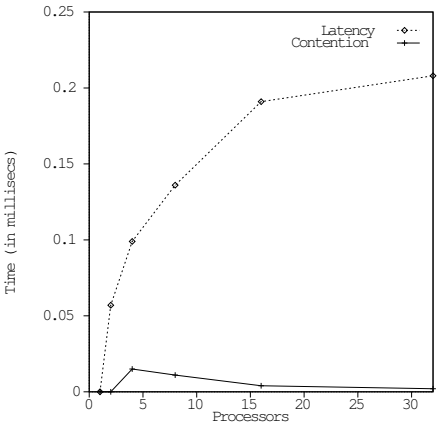

Figure 10: Latency and Contention (Full)

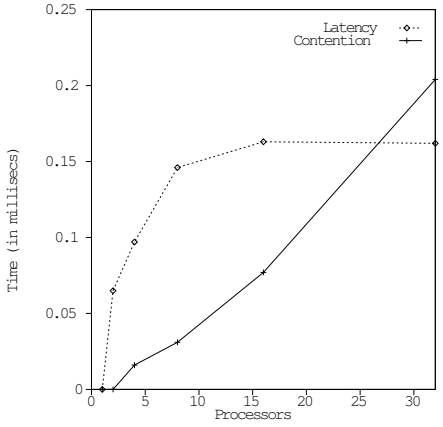

Figure 11: Latency and Contention (Cube)

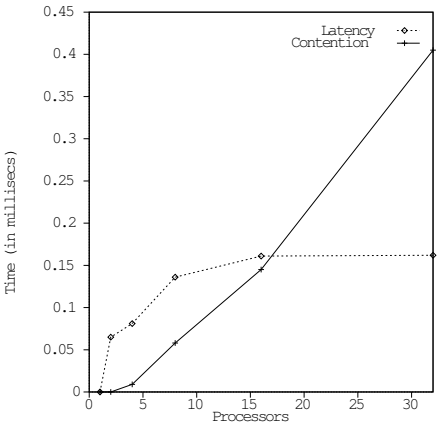

Figure 12: Latency and Contention (Mesh)

IS Graphs

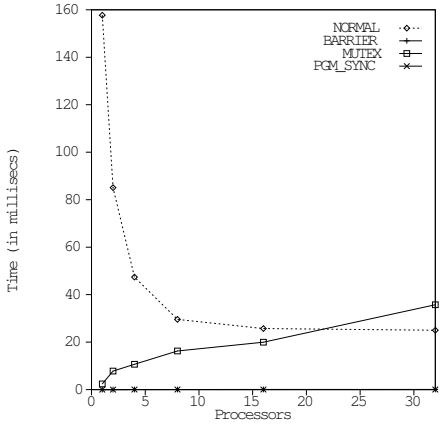

Figure 13: Mode-wise Execn. Time (Mesh)

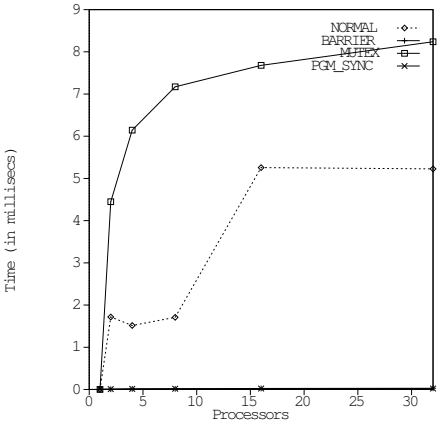

Figure 14: Mode-wise Latency (Mesh)

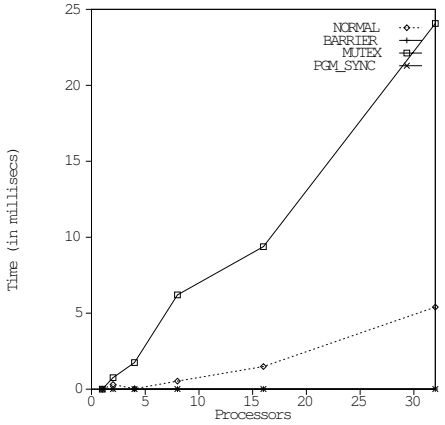

Figure 15: Mode-wise Contention (Mesh)

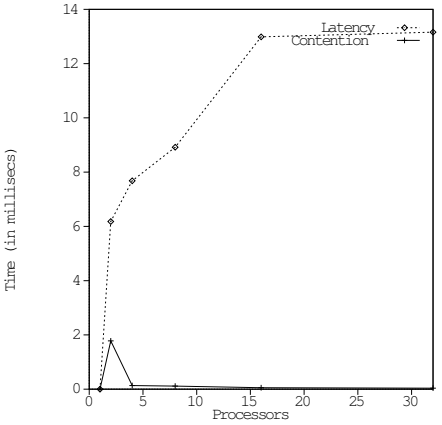

Figure 16: Latency and Contention (Full)

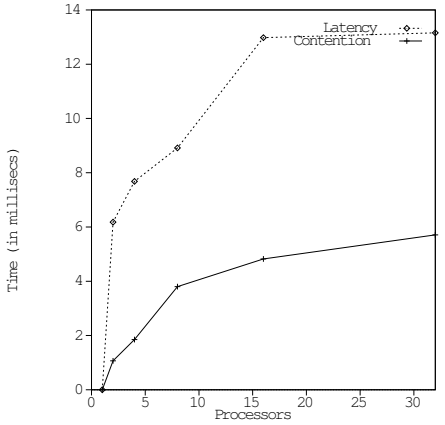

Figure 17: Latency and Contention (Cube)

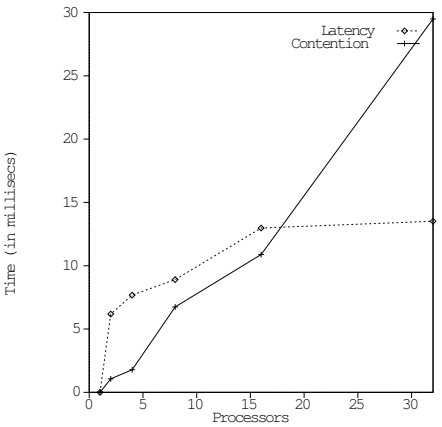

Figure 18: Latency and Contention (Mesh)

FFT Graphs

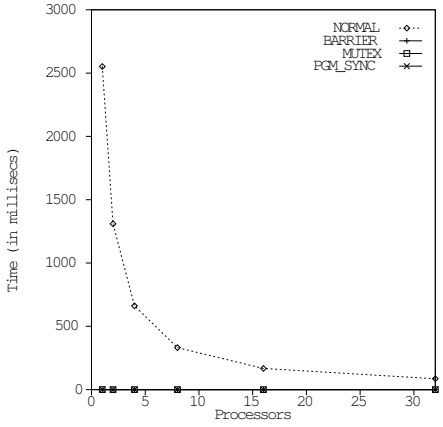

Figure 19: Mode-wise Execn. Time (Mesh)

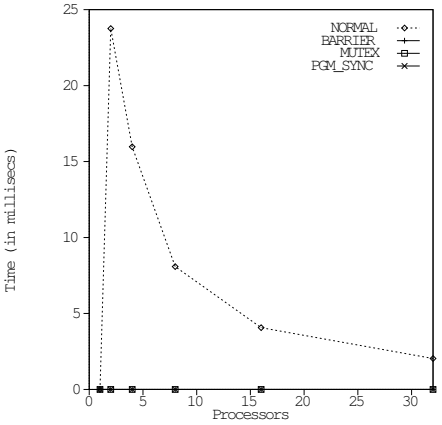

Figure 20: Mode-wise Latency (Mesh)

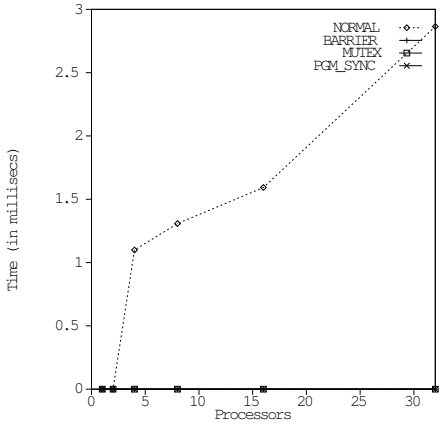

Figure 21: Mode-wise Contention (Mesh)

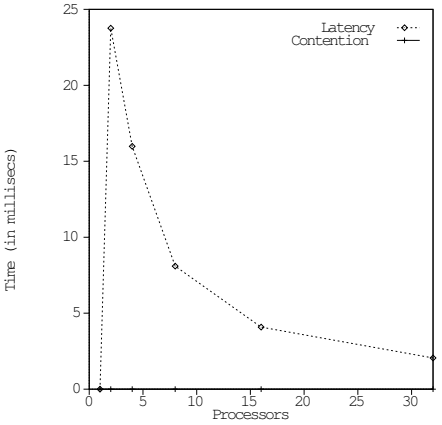

Figure 22: Latency and Contention (Full)

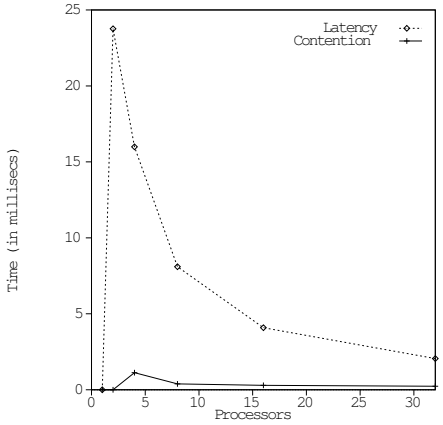

Figure 23: Latency and Contention (Cube)

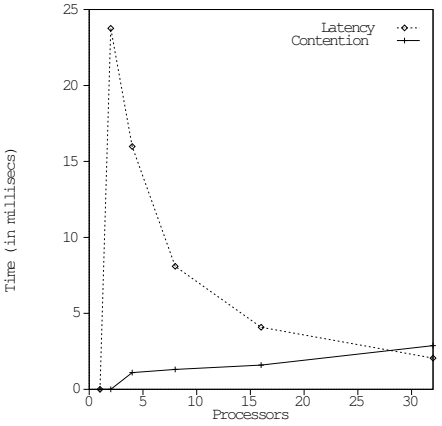

Figure 24: Latency and Contention (Mesh)

CG Graphs

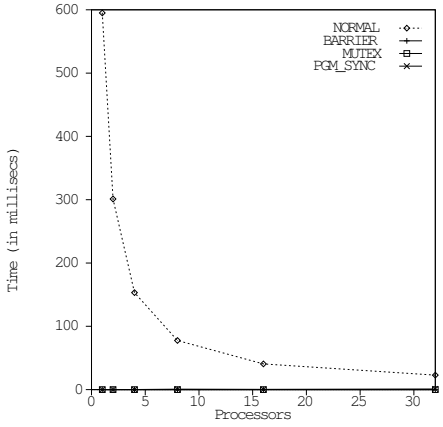

Figure 25: Mode-wise Execn. Time (Mesh)

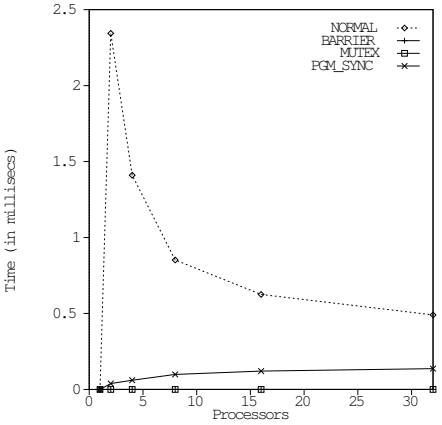

Figure 26: Mode-wise Latency (Mesh)

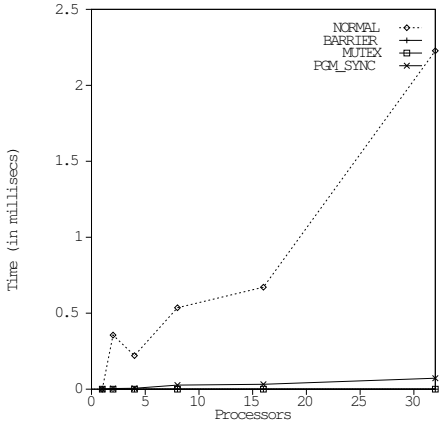

Figure 27: Mode-wise Contention (Mesh)

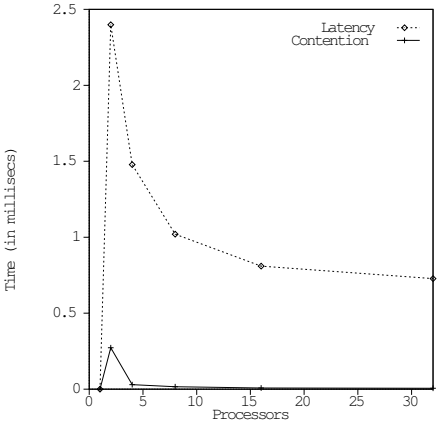

Figure 28: Latency and Contention (Full)

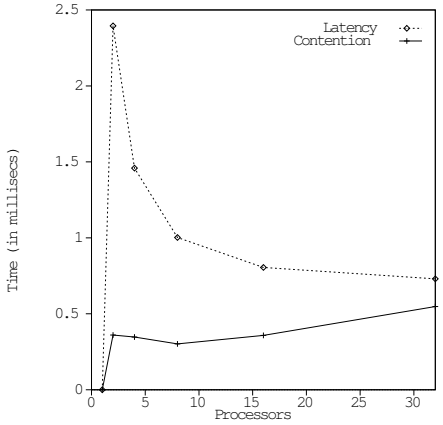

Figure 29: Latency and Contention (Cube)

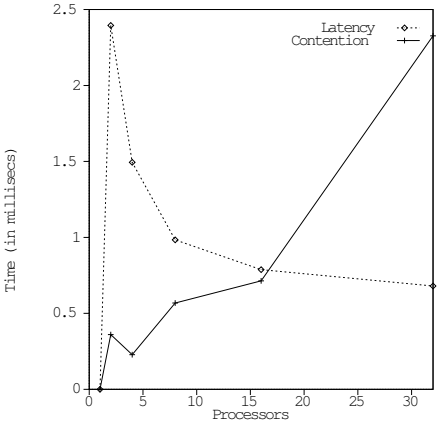

Figure 30: Latency and Contention (Mesh)

CHOLESKY Graphs

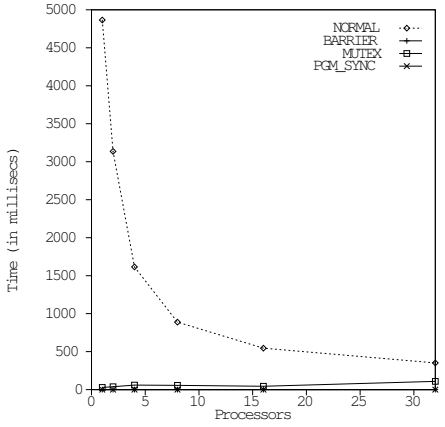

Figure 31: Mode-wise Execn. Time (Mesh)

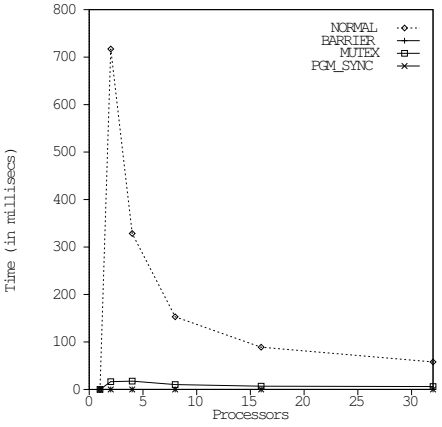

Figure 32: Mode-wise Latency (Mesh)

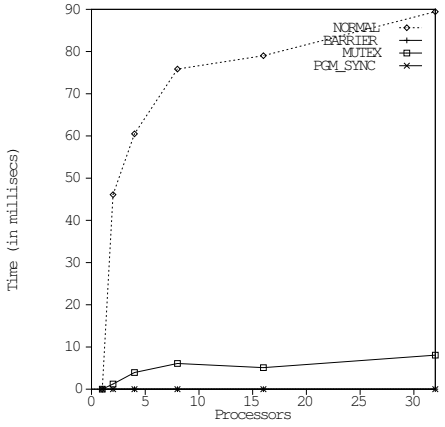

Figure 33: Mode-wise Contention (Mesh)

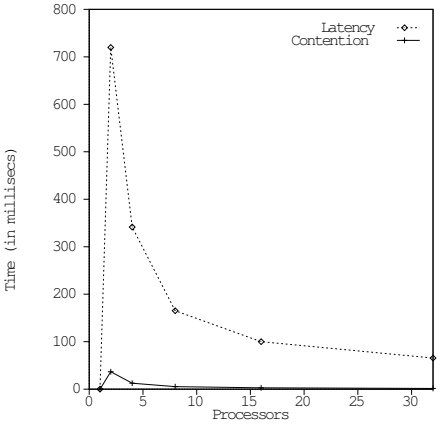

Figure 34: Latency and Contention (Full)

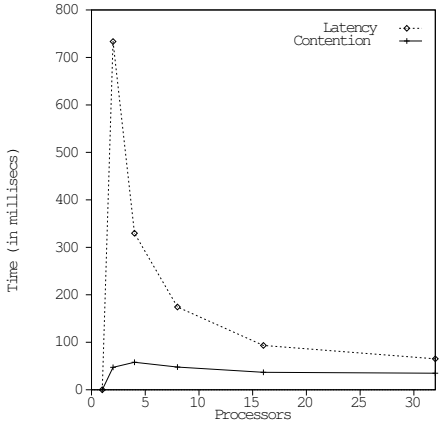

Figure 35: Latency and Contention (Cube)

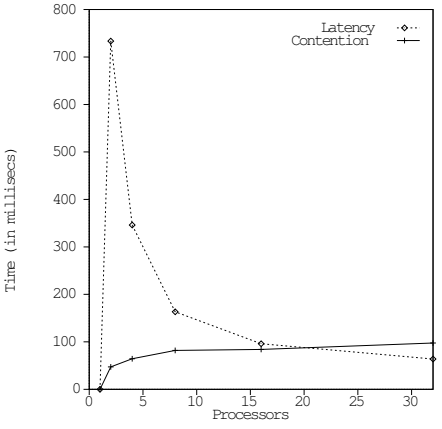

Figure 36: Latency and Contention (Mesh)