In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 171-180, May 1994.

An Approach to Scalability Study of Shared Memory Parallel Systems[*]

Anand Sivasubramaniam *Aman Singla* *Umakishore Ramachandran* *H. Venkateswaran*

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280.
{*anand, aman, rama, venkat*}*@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 isolate the algorithmic overhead and 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.

1 Introduction

Scalability is a notion frequently used to signify the "goodness" of parallel systems, where the term parallel system is used to denote an application-architecture combination. 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, identify application 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 on 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 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 the system.

The main contributions of our work can be summarized as follows: we define the notion of *overhead functions* associated with the different algorithmic and architectural characteristics; we develop a method for separating the algorithmic overhead; we also isolate the overheads due to network *latency* (the actual hardware transmission time in the network) and *contention* (the amount of time spent waiting for a resource to become free in the network) from the overall execution time of an application; we design and implement a simulation platform that quantifies these overheads; and we use this simulator to study the scalability of five applications on shared memory platforms with three different network topologies.

Performance metrics such as speedup [2], scaled speedup [11], sizeup [25], experimentally determined serial fraction [12], and isoefficiency function [13] have been proposed for quantifying the scalability of parallel systems. While these metrics are extremely useful for tracking performance trends, they do not provide adequate information needed to understand the reason why an application does not scale well on an architecture. The overhead functions that we identify, separate, and quantify in this work, help us overcome this inadequacy. We are not aware of any other work that separates these overheads (in the context of real applications), and believe that such a separation is very important in understanding the interaction between applications and architectures. The growth of overhead functions will provide key insights on the scalability of a parallel system by suggesting application restructuring, as well as architectural enhancements.

Several performance studies address issues such as latency, contention and synchronization. The scalability of synchronization primitives supported by the hardware [3, 15] and the limits on interconnection network performance [1, 16] are examples of such studies. While such issues are extremely important, it is necessary 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 [10] and the impact of synchronization and task granularity on parallel system performance [6]. Cypher et al. [9] identify the architectural requirements such as floating point operations, communication, and input/output for message-passing scientific applications. Rothberg et al. [18] 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.

This work is part of a larger project which aims at understanding

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

the significant issues in the design of scalable parallel systems using the above-mentioned top-down approach. In our earlier work, we studied issues such as task granularity, data distribution, scheduling, and synchronization, by implementing frequently used parallel algorithms on shared memory [21] and message-passing [20] platforms. In [24], we illustrated the top-down approach for the scalability study of message-passing systems. In this paper, we conduct a similar study for shared memory systems. In a companion paper [23] we evaluate the use of abstractions for the network and locality in the context of simulating cache-coherent shared memory multiprocessors.

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 quantifies these overhead functions are also discussed in this section. The characteristics of the five applications used in this study are summarized 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 presented in Section 6.

2 Top-Down Approach

Adhering to 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 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 [14]) 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.

Parallel system overheads (see Figure 1) may be broadly classified into a purely algorithmic component (*algorithmic overhead*), and a component arising from the interaction of the algorithm and the architecture (*interaction overhead*). 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 [26] and measuring its deviation from a linear speedup curve. A real execution could deviate significantly from the ideal execution due to overheads such as latency, contention, synchronization, scheduling and cache effects. These overheads are lumped together as the interaction overhead. 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). Therefore, 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.

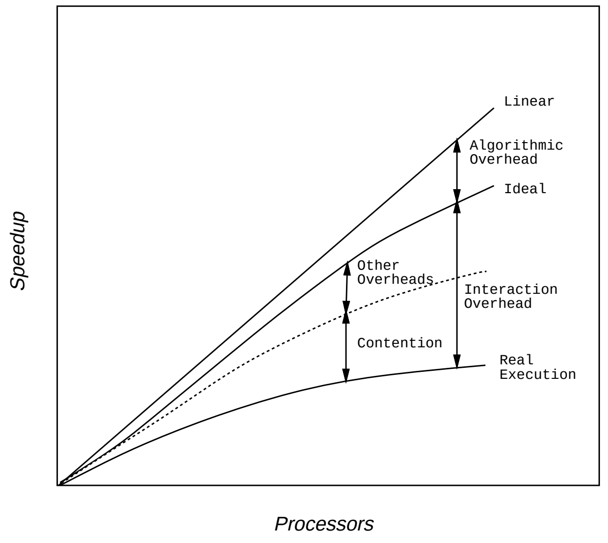

Figure 1: Top-down Approach to Scalability Study

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

- 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 for 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 they exhibit. Experimentation, simulation, and analytical models are three techniques that have been commonly used in such studies. But it is well-known that each has its own limitations. The main focus of our top-down approach is to quantify the overheads that arise from the interaction between the kernels and the architecture and their impact on the overall execution of the application. Experimentation on real architectures does not allow studying the effects of changing individual architectural parameters on the performance. It is not clear that analytical models can realistically capture the complex and dynamic interactions between applications and architectures. Therefore, we use simulation for quantifying and separating the overheads.

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 [26] and measuring its deviation from a linear speedup curve. The interaction overhead is also separated into its component parts. We currently do not address scheduling overheads[1]. Accesses to variables in a shared memory system may involve the network, and the

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

physical limitations of the network tend to contribute to 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* 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* is the total amount of time incurred by a processor due to the time spent waiting for links to become free by the messages. Shared memory systems normally provide some synchronization support that is 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 incurred by a processor during synchronization operations is accounted for in the CPU time which would manifest itself as an algorithmic overhead. The statistics (CPU time, latency overhead, and contention overhead) are quantified and presented for each interesting mode of the program execution (see Section 2.2).

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. 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, 17], 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 (such as LOADS and STORES) that may potentially involve a network access are simulated. The input to the simulator are parallel applications written in C. These programs are pre-processed (to label shared memory accesses), the compiled assembly code is augmented with cycle counting instructions, and the assembled binary is linked with the simulator code. The system parameters that can be specified to SPASM are: the *number of processors (p)*, the *clock speed* of the processor, the *hardware bandwidth* of the links in the network, and the *switching delays*.

2.2.1 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 as well as the contention overhead function 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 that results from 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 Application Characteristics

Three of the applications (EP, IS and CG) are from the NAS parallel benchmark suite [4]; CHOLESKY is from the SPLASH benchmark suite [19]; and FFT is the well-known Fast Fourier Transform algorithm. EP and FFT are well-structured applications 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 applications 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. The reader is referred to [22] for further details of the applications.

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 bidirectional binary hypercube topology and uses the e-cube algorithm for routing. 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 in the mesh are routed along the row until they reach the destination column, upon which they are routed along the column. Messages on all three platforms are circuit-switched using a wormhole routing strategy and the switching delay is assumed to be 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. The memory access time is assumed to be 5 CPU cycles, while the cache access time is assumed to be 1 CPU cycle.

5 Performance Results

In this section, we present 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 system parameters such as link bandwidth 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. Only the results for IS, FFT and CHOLESKY are discussed in this section due to space constraints. Details on the other kernels can be found in [22].

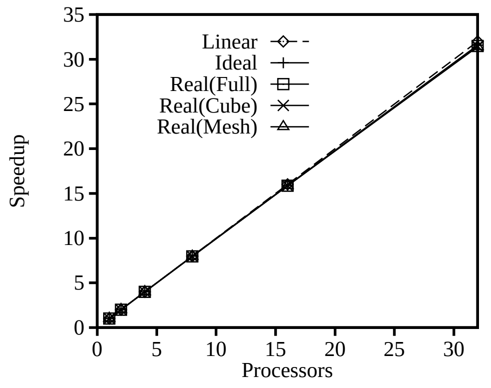

Figure 2: EP: Speedup

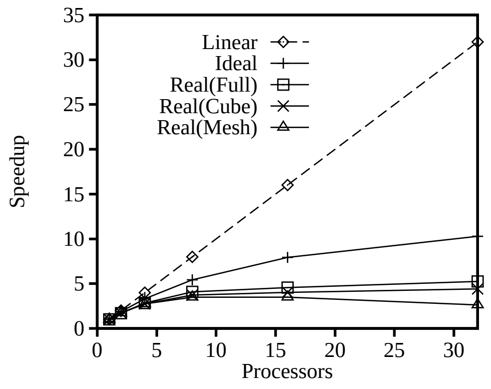

Figure 3: IS: Speedup

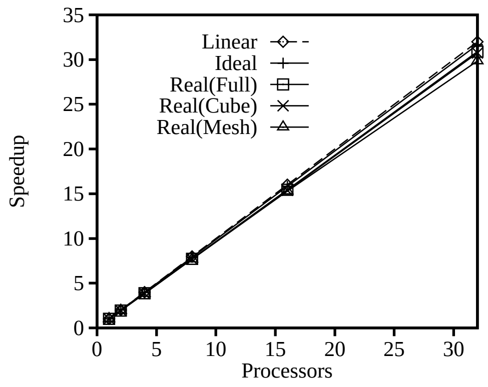

Figure 4: FFT: Speedup

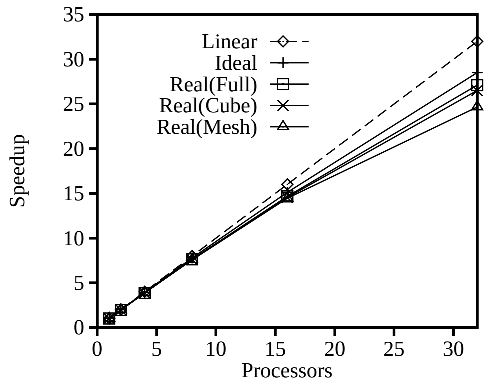

Figure 5: CG: Speedup

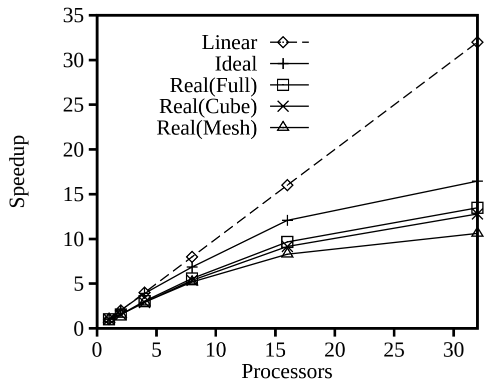

Figure 6: CHOLESKY: Speedup

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 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 7). 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 8 and 9). 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 7), which also explains the dip in the speedup curve for mesh (see Figure 3).

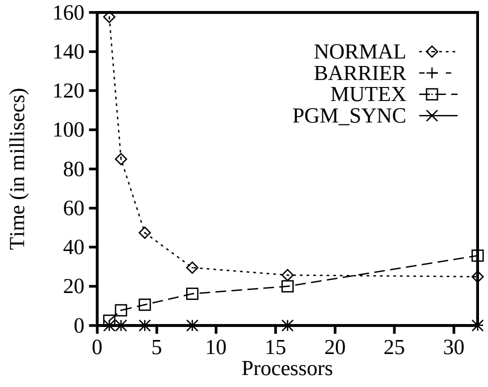

Figure 7: IS: Mode-wise Execn. Time (Mesh)

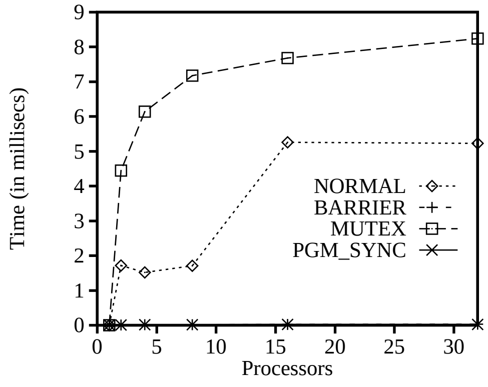

Figure 8: IS: Mode-wise Latency (Mesh)

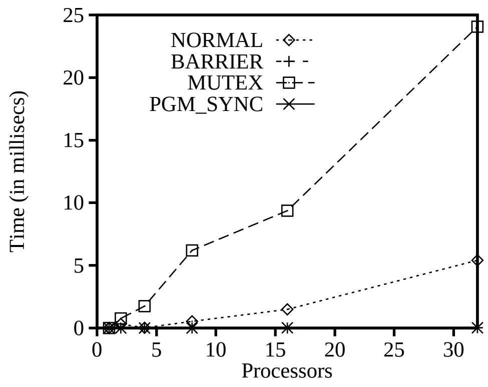

Figure 9: IS: Mode-wise Contention (Mesh)

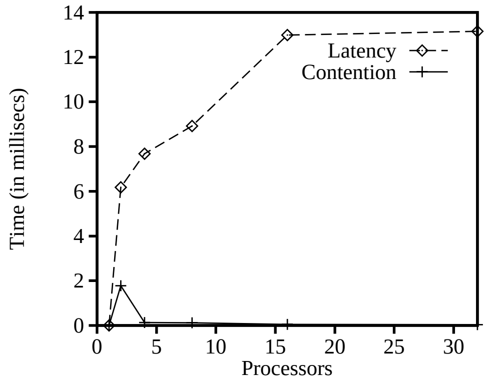

Figure 10: IS: Latency and Contention (Full)

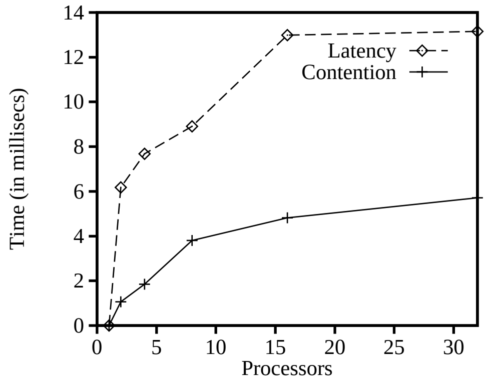

Figure 11: IS: Latency and Contention (Cube)

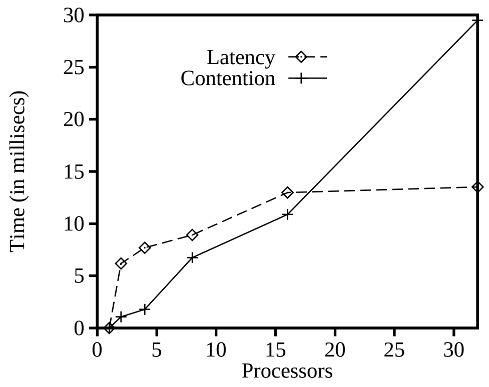

Figure 12: IS: Latency and Contention (Mesh)

Figures 10, 11 and 12. show the latency and contention overheads for the three hardware platforms. In IS, since every processor needs to access the data 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 1 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)	$13.2(1-\frac{1}{p})$	$13.2(1-\frac{1}{p})$	$13.2(1-\frac{1}{p})$
Contention (ms)	$Negligible$	$4.0\log p$	$0.9p$

Table 1: IS : Overhead Functions

Parallelization of this kernel increases the amount of work to be done for a given problem size (see [22]). This inherent algorithmic overhead causes a deviation of the ideal curve from the linear curve (see Figure 3). This is also confirmed in Table 1, 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 1, 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.

5.2 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 13). The latency and contention overheads (Figures 14 and 15) incurred in this mode are insignificant compared to the total execution time, despite the growth of contention overhead with increasing number of processors.

around 28 processors. Table 2 summarizes the overheads for FFT obtained by interpolating the datapoints from our simulation results.

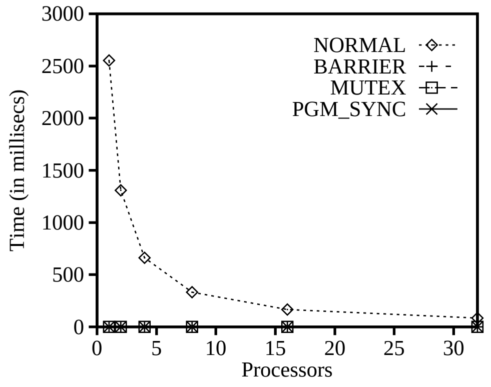

Figure 13: FFT: Mode-wise Execn. Time (Mesh)

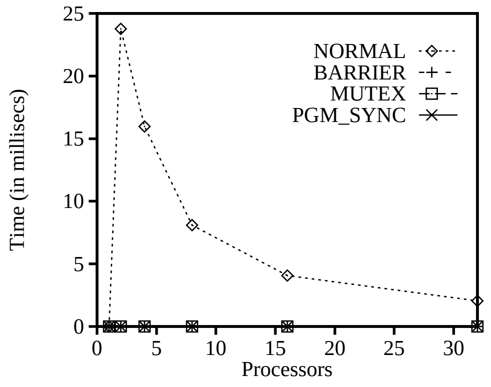

Figure 14: FFT: Mode-wise Latency (Mesh)

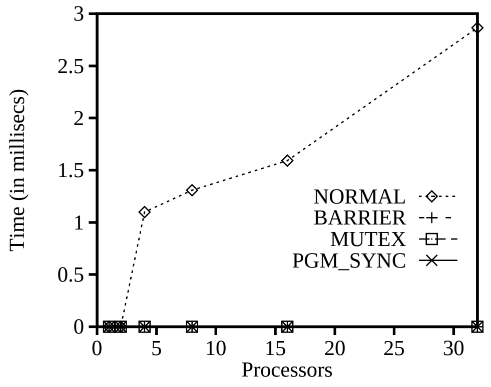

Figure 15: FFT: Mode-wise Contention (Mesh)

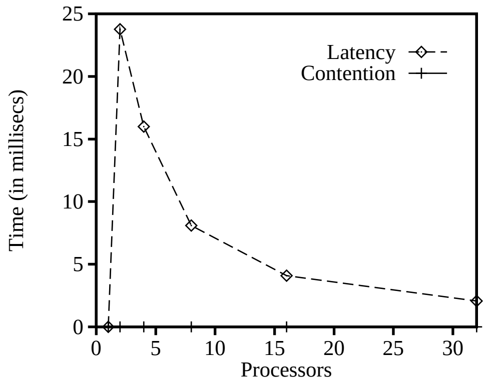

Figure 16: FFT: Latency and Contention (Full)

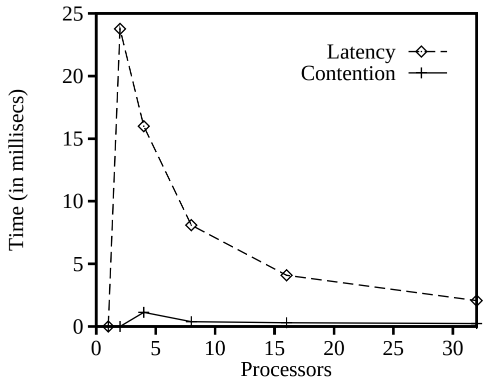

Figure 17: FFT: Latency and Contention (Cube)

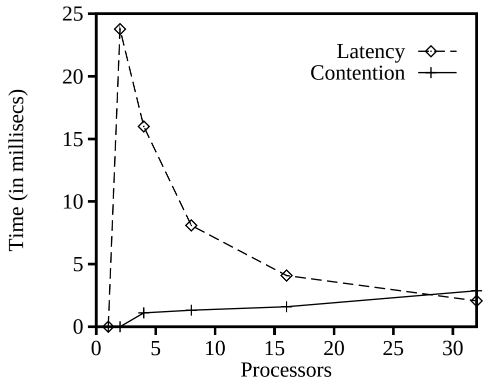

Figure 18: FFT: Latency and Contention (Mesh)

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 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 16) and the cube (Figure 17) platforms. On the mesh (Figure 18), the contention surpasses the latency overhead at

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 2). 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

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 2: FFT : Overhead Functions

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.

5.3 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 19) 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 20 and 21). 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 19).

As with 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 22) and grows with increasing processors for the cube (Figure 23), becoming more dominant than the latency overhead for the mesh (Figure 24) at around 20 processors. Table 3 summarizes the overheads for CHOLESKY obtained by interpolating the datapoints from our simulation results.

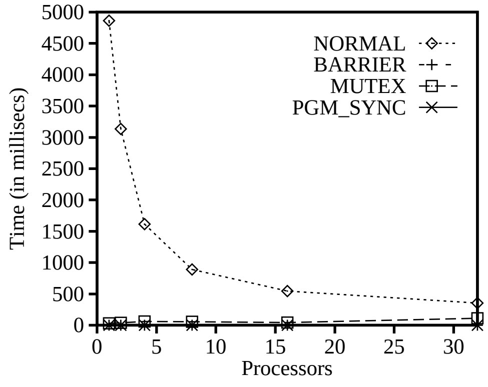

Figure 19: CHOLESKY: Mode-wise Execn. Time (Mesh)

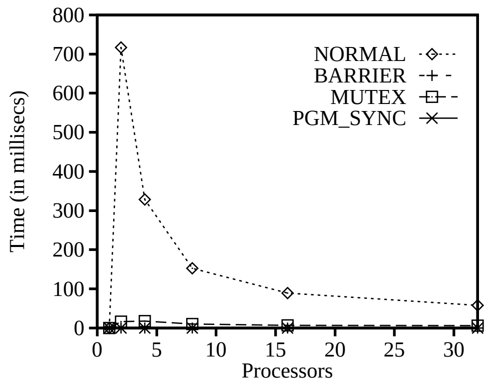

Figure 20: CHOLESKY: Mode-wise Latency (Mesh)

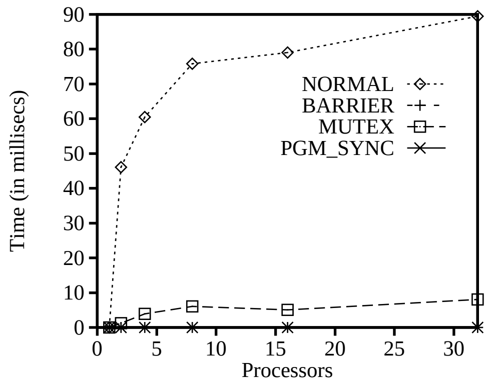

Figure 21: CHOLESKY: Mode-wise Contention (Mesh)

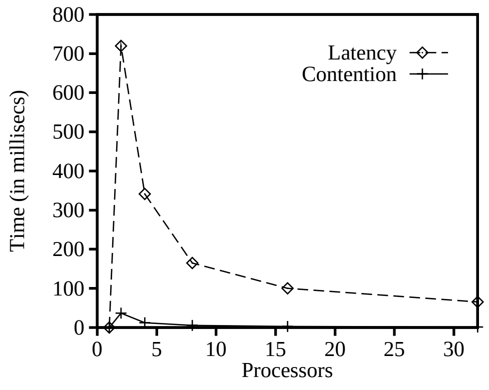

Figure 22: CHOLESKY: Latency and Contention (Full)

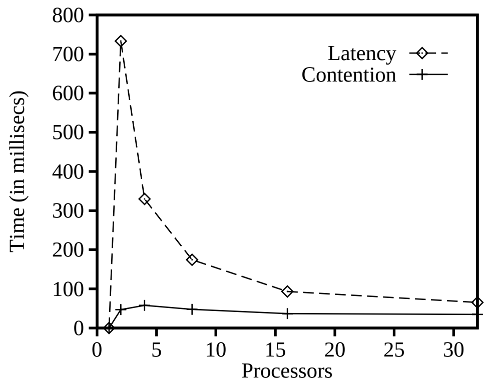

Figure 23: CHOLESKY: Latency and Contention (Cube)

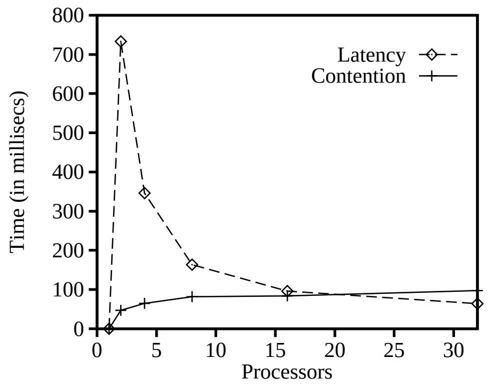

Figure 24: CHOLESKY: Latency and Contention (Mesh)

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 3: 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 3, 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 for understanding the scalability of parallel systems. 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] A. 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] T. 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] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. 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] D. 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] 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.

[10] S. J. Eggers and R. 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.

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

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

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

[14] 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.

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

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

[17] 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.

[18] E. Rothberg, J. P. Singh, and A. 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.

[19] 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.

[20] A. Sivasubramaniam, U. 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.

[21] A. Sivasubramaniam, G. Shah, J. Lee, U. 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.

[22] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. An Approach to Scalability Study of Shared Memory Parallel Systems. Technical Report GIT-CC-93/62, College of Computing, Georgia Institute of Technology, October 1993.

[23] A. Sivasubramaniam, A. Singla, U. 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.

[24] A. Sivasubramaniam, A. Singla, U. 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.

[25] X-H. Sun and J. L. Gustafson. Towards a better Parallel Performance Metric. *Parallel Computing*, 17:1093–1109, 1991.

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