A Simulation-based Scalability Study of Parallel Systems*

Anand Sivasubramaniam
Aman Singla
Umakishore Ramachandran
H. Venkateswaran

Technical Report GIT-CC-93/27
April 1993

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

Abstract

Scalability studies of parallel architectures have used scalar metrics to evaluate their performance. Very often, it is difficult to glean the sources of inefficiency resulting from the mismatch between the algorithmic and architectural requirements using such scalar metrics. Low-level performance studies of the hardware are also inadequate for predicting the scalability of the machine on real applications. We propose a *top-down* approach to scalability study that alleviates some of these problems. We characterize applications in terms of the frequently occurring kernels, and their interaction with the architecture in terms of overheads in the parallel system. An *overhead function* is associated with the algorithmic characteristics as well as their interaction with the architectural features. We present a simulation platform called SPASM (Simulator for Parallel Architectural Scalability Measurements) that quantifies these overhead functions. SPASM separates the algorithmic overhead into its components (such as serial and work-imbalance overheads), and interaction overhead into its components (such as latency and contention). Such a separation is novel and has not been addressed in any previous study. We illustrate the top-down approach by considering a case study in implementing three NAS parallel kernels on two simulated message-passing platforms.

Key words: parallel systems, parallel kernels, scalability, execution-driven simulation, performance evaluation, performance metrics.

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

1 Introduction

With rapid advances in technology, the past decade has witnessed an evolution of parallel machines, both in industry as well as in academia. Coupled with this evolution there has also been a growing awareness in the computer science community to go beyond pure algorithmic work to the actual experimentation of parallel algorithms on real machines. Algorithmic work has usually been based on abstract models of parallel machines that may not accurately capture the features of the architecture that are important from the performance standpoint. These models have evolved from their corresponding sequential counterparts. While machine models used in sequential algorithm design have been extremely successful in predicting the running time on uniprocessors within a constant factor, experimentation has revealed that parallel systems[1] do not enjoy the same luxury. The reason for this disparity is that parallel systems have several additional degrees of freedom compared to sequential processing, such as task granularity, synchronization, data allocation and movement, and work imbalance. Analytical models for parallel systems are even more difficult to build and often use simplistic assumptions about the system to keep the complexity of such models reasonable for purposes of analysis [28, 18]. *Scalability* is a notion frequently used to signify the "goodness" of parallel systems. A good understanding of this notion may be used to: select the best algorithm-architecture combination for a problem, predict the performance of an algorithm on an architecture with a larger number of processors, determine the optimal number of processors to be used for the algorithm and the maximum speedup that can be obtained, and glean insight on the influence of the algorithm on the architecture and vice-versa to enable us to understand the scalability of other algorithm-architecture pairs.

Several performance metrics such as speedup [2], scaled speedup [11], sizeup [27], experimentally determined serial fraction [14], and isoefficiency function [15] have been proposed over the years for capturing 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 sources of inefficiency in a given architecture with respect to a given algorithm. An understanding of the interaction between the algorithmic and architectural characteristics of a parallel system can give us such information. Studies undertaken by Kung [16] and Jamieson [13] help identify some of these characteristics from a theoretical perspective but they do not provide any means of quantifying their effects.

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

Parallel algorithms designed for an idealized machine model, project asymptotic estimates for their performance that may not be realizable in practice. Architects are usually concerned with low-level performance issues such as latency, contention and synchronization. The scalability of synchronization primitives supported by the hardware [3, 20], the limits on interconnection network performance [1, 21], and the performance of scheduling policies [30, 17] 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 [10] and the impact of synchronization and task granularity on parallel system performance [7]. Cypher et al. [9], identify the architectural requirements such as floating point operations, communications, and input/output for scientific applications. However, there have been very few attempts at quantifying the effects of algorithmic and architectural interactions in a parallel system.

Since real-life applications set the standards for computing, it is meaningful to use the same applications for the evaluation 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 (section 2). To this end, we have developed a simulation platform called SPASM (Simulator for Parallel Architecture Scalability Measurements), which identifies different *overhead functions* that help quantify deviations from ideal behavior of a parallel system (section 3).

The following are the important contributions of this work:

- We propose a top-down approach to the performance evaluation of parallel systems.
- 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 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 algorithms and architectures.

- We design and implement a simulation platform that incorporates these methods for quantifying the overhead functions.

We illustrate the top-down approach through a case study, implementing a few NAS parallel kernels [4] on two message-passing platforms (a bus and a binary hypercube) simulated on SPASM. The algorithmic characteristics of these kernels are discussed in Section 4, details of the two architectural platforms are presented in Section 5, and the results of our study are summarized in Section 6. Concluding remarks and directions for future research are given in Sections 7 and 8.

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. Moreover, they tend to contain 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 [19]) and evaluating their performance. As one goes lower in the hierarchy, the outcome of the evaluation becomes less realistic. Our top-down approach uses a *hierarchical* method to benchmarking based on the granularity of the benchmarks. The Perfect Club Benchmarks [5], SPLASH [23] and the NAS Benchmarks [4] are examples of application suites that have been proposed for studying the performance of parallel machines. Such applications are representative of real workloads and appear at the top of our hierarchy.

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

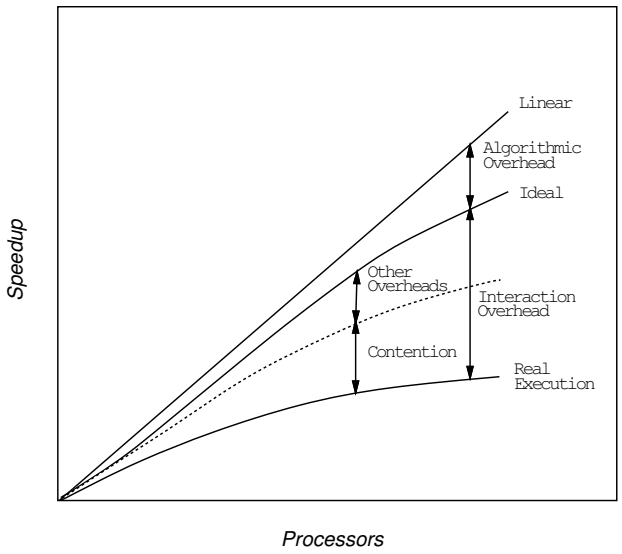

Figure 1: Top-down Approach to Scalability Study

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 of 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, in particular the NAS parallel kernels [4] that have been derived from a large number of Computational Fluid Dynamics applications.

One would like 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. For instance, if in certain parallel phases of an algorithm the number of processors utilized changes then it would create work imbalance. 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 [29, 25]. Such a machine idealizes the parallel architecture by assuming an infinite number of processors, and unit costs for communication and synchronization. Hence, the 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 that has a fully connected interconnection network there is 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 in between the ideal curve and the real execution curve (see Figure 1).

One can use either simulation or direct experimental evaluation of the applications on the real hardware to implement the top-down approach. We adopted the latter technique in our earlier studies by experimenting with frequently used parallel algorithms on shared memory [26] and message-passing [24] platforms. This technique 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. In this study, we use the simulation technique to overcome these drawbacks. Experimentation is used in conjunction with simulation to understand the performance of real applications on real architectures, and to identify the interesting kernels that occur in these applications for subsequent use in the simulation studies.

Our simulation platform (SPASM), to be presented in the next 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 [29] 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

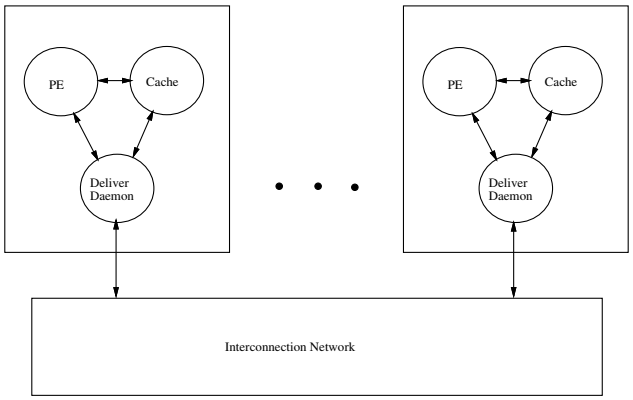

Figure 2: Architecture of SPASM

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 by assuming that the number of processes spawned in a parallel program is equal to the number of processors in the simulated machine, and that a process is bound to a processor and does not migrate [2] We have also confined ourselves to message-passing platforms in this study, where synchronization and communication are intertwined. Thus the interaction overhead is quantified using the *latency overhead function* ($f_L(p)$) and the *contention overhead function* ($f_C(p)$) (p is the number of processors) that are described in the next section. In a shared memory platform, it would be interesting to consider the impact of communication and synchronization in the algorithm on latency and contention separately but are outside the scope of this paper. 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.

3 SPASM

SPASM (Simulator for Parallel Architectural Scalability Measurements) is an execution-driven simulator that enables us to conduct a variety of scalability measurements of parallel applications on a number of simulated hardware platforms. SPASM has been written using CSIM, a process oriented sequential simulation package, and currently runs on SPARCstations. SPASM provides support for process control, communication and synchronization. The implementation of these mechanisms for each simulated platform is identified by well-defined interfaces to library routines.

[2] We do not distinguish between the terms, *process*, *processor* and *thread*, and uniformly refer to the term as processor in this paper.

There is a library for each simulated platform, that is linked to the rest of the simulator modules.

Architectural simulators have normally tended to be very slow, making it tedious to get a whole range of data points on realistic problems. Simulating the entire instruction set of a processor can be considerably slow. Since the thrust of this study is to understand the interesting characteristics of parallel machines and their impact on the algorithm, instruction-level simulation is not likely to contribute extensively to this understanding. Hence, we have confined ourselves to simulating the interesting aspects of parallel machines. The bulk of the processor instruction streams is executed at the speed of the native processor (the SPARC in this case) and only those instructions that could potentially involve the network are simulated by SPASM. Examples of such instructions include sends and receives on a message-passing platform, and loads and stores on a shared memory platform. Such instructions are trapped by our simulator and simulated exactly according to the semantics of these instructions on each particular platform (through the library routines). SPASM reconciles the real time with the simulated time using these trapped instructions. Upon such a trap, SPASM computes the time for the block of instructions that were executed at the native speed since the previous trap,[3] and updates the simulation clock of the processor. This strategy has considerably lowered the time overhead for simulation (the simulation time is at most a factor of two compared to the real time) for the applications considered. The approach has also been used in other recent simulation studies [6, 8, 22].

Figure 2 depicts the architecture of our simulation platform. Each node in the parallel machine is abstracted by a processor (PE), a cache module (Cache) and a network interface (Deliver Daemon). These three entities are implemented as CSIM processes. The CSIM process representing a processor executes the code associated with the processor. These CSIM processes are spawned at the start of the program, and execute as co-routines. A CSIM process executes to completion if there are no more instructions that need to be trapped by the simulator. If not, control is passed to the next CSIM process ready to execute, while the simulation of the trapped instruction is performed. The simulation thus keeps the CSIM processes in loose synchrony. On a shared memory platform, the processor issues load/store requests to its cache module. The cache module services the request, invoking the deliver daemon if required. Since the shared memory platform is beyond the purview of this study, we do not discuss any further details of its implementation. On a message-passing platform, the processor directly interacts with the deliver daemon and the

[3]Since CSIM executes its processes as co-routines, there is only one CSIM process executing in this time interval.

explicit send/receive in a program are the only instructions that would need this interaction. On a send, the processor creates a message (a data structure) and places the message in a mailbox. The daemon waits for a message to arrive on the mailbox associated with its processor. It picks up the message from the mailbox, determines the routing information, waits for the relevant links of the network to become free, accounts for the software and hardware overheads, and delivers the message to the mailbox of the destination processor [4]. On a synchronous receive, the processor blocks until a message (delivered by a deliver daemon) appears in its mailbox.

SPASM provides us with a wide range of input parameters and output statistics for understanding the scalability of parallel systems.

3.1 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*.

Even though we assume that each processor is a SPARC chip, SPASM allows varying the clock speed of the simulated processor for the same instruction set. This is useful for understanding the scalability of parallel systems of the future built with faster processors. It also provides us with a mechanism for varying the computation to communication ratio of a parallel system.

3.2 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.

[4] For reasons of efficiency, there is only 1 deliver daemon associated with a processor on start-up. When a processor finds that its associated deliver daemon is busy delivering a message, it spawns another daemon and passes on the message to it. The newly spawned daemon dies after it delivers the message while the start-up daemon goes back for the next message.

A processor may wait for an event (such as a synchronization or a communication operation) even before the event occurs. For the message passing platform being considered the only events are the sending and receiving of messages. The difference between the time the receive was posted and the time the message was sent is due to skews between the processors and is called the *wait time* of a processor. In an ideal machine the wait time is entirely due to the work-imbalance overhead, and is a metric provided by SPASM. The difference between the algorithmic overhead and the work-imbalance overhead gives the serial overhead in the algorithm.

As mentioned in section 2, we would like to isolate the effects of latency and contention in the system. If a machine is to have a fully connected network, there would be no contention in the system and the overhead of a message would be purely due to software and hardware latencies for communication. Each processor performing a blocking receive is expected to see this latency given that all other conditions are ideal. The sum of all these overheads seen by a processor is called the *Network Latency*. If a processor is to receive messages $m_1, m_2, ...m_k$ and the latencies of these messages are $l_1, l_2, ...l_k$ (includes both software and hardware components), then the network latency ($f_l(p)$) *incurred* by a processor is given by $f_l(p) = \sum_{i=1}^{k} l_i$. But this may not necessarily reflect the *real* latencies observed by a processor since some of it may be hidden by the overlap of computation with communication. We call the latency *observed* by a processor as the *latency overhead function* ($f_L(p)$). SPASM gives the network latency of a processor as well as the latency overhead function seen by a processor. SPASM measures the latter entity by time-stamping messages at the sending processor. SPASM checks to see if the destination processor posted a receive for the message after it was sent in which case only a corresponding part of the network latency is accounted for as the latency overhead. If the destination processor posted a wait for the message before it was sent, then the entire network latency is charged to it as the latency overhead.

As with latency, SPASM provides information about the *network contention* ($f_c(p)$) that a processor is supposed to incur and the *contention overhead function* ($f_C(p)$) actually observed by the processor at the receiving end. Network contention incurred by a processor is the sum of all the waiting times (due to network links not being available) for all the messages that it receives. If a processor is to receive messages $m_1, m_2, ...m_k$ and the amount of time spent by these messages waiting for links to become free in the network are $c_1, c_2, ...c_k$ respectively, then the *network contention* ($f_c(p)$) *incurred* by a processor is given by $f_c(p) = \sum_{i=1}^{k} c_i$. A processor may choose to hide a part of this contention by overlapping computation with communication, or a

processor may simply find a message already available when it posts a receive (in which case it does not see any contention). The contention actually *observed* by a processor is called the *contention overhead function* ($f_C(p)$). SPASM calculates this overhead using time-stamped messages and the time that would have been taken by a message on a contention-free network (i.e. the network latency).

The wait time experienced by a processor on a real machine includes the work-imbalance overhead (a purely algorithmic characteristic), as well as processor skews introduced due to the latency and contention experienced by the messages. Let us denote, the wait times due to work-imbalance, latency, and contention by W_w, W_l, and W_c; and the wait times measured by SPASM on an ideal machine, real machine with a fully connected network, and the real machine by W_i, W_f, and W_r respectively. Then the component wait times can be computed using the following expressions:

$$W_w = W_i$$

$$W_l = W_f - W_i$$

$$W_c = W_r - W_f$$

From the above discussion, it follows that:

$Total\ Time = Ideal\ Time + f_L(p) + f_C(p) + W_r$

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. Thus the metrics identified by SPASM quantify the algorithmic overhead and the interesting components of the interaction overhead.

4 Algorithmic Characteristics

Kernels are abstractions of the major phases of computation in an application that account for the bulk of the execution time. A *parallel kernel* is characterized by the data access pattern, the synchronization pattern, the communication pattern, the computation granularity (which is the amount of work done between synchronization points), and the data granularity (which is the amount of data manipulated between synchronization points). The last two together define the task granularity of the parallel kernel. These attributes are as seen from the point of view of

the individual processors implementing the parallel kernel. If the parallel kernel is implemented using the message-passing style, then the data access pattern becomes unimportant (except for any cache effects) since all data accesses are to private memory. Further, the synchronization is usually merged with the communication in such an implementation. On the other hand, if a shared memory style programming is used, the communication pattern is not explicit and gets merged with the data access pattern.

The Numerical Aerodynamic Simulation (NAS) program at NASA Ames has identified a set of kernels [4] that are representative of a number of large scale Computational Fluid Dynamics codes. In this study, we consider three of these kernels for the purposes of illustrating the top-down approach using SPASM. In this section, we identify their characteristics in a message-passing style implementation.

Phase	Description	Comp. Gran.	Data Gran.
1	Local Floating Pt. Opns.	Large	N/A
2	Global Sum	Integer Addition	4 bytes

Figure 3: Algorithmic Characteristics of EP Kernel

Phase	Description	Comp. Gran.	Data Gran.
1	Local bucket updates	Small	N/A
2	Global bucket merge	Small	8K bytes
3	Local bucket updates	Small	N/A

Figure 4: Algorithmic Characteristics of IS Kernel

EP is the "Embarassingly Parallel" kernel that generates pairs of Gaussian random deviates and tabulates the number of pairs in successive square annuli. This problem is typical of many Monte-Carlo simulation applications. The kernel is computation bound and has little communication among the processors. A large number of floating point random numbers is calculated and a sequence of floating point operations is performed on them. 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 totally independent of the other random numbers. The processor assigned to a random number can thus execute all the operations for that

Phase	Description	Comp. Gran.	Data Gran.
1	Local Floating Pt. Opns	Medium	N/A
2	Matrix-Vector Product		
2a	Global Vector Merge	N/A	$(11200/p) * 2i$ in step i
2b	Local Matrix-Vector Product	Medium	N/A
3	Vector-vector dot product		
3a	Local vector-vector dot product	Small	N/A
3b	Global Sum	Floating Pt. Addition	8 bytes
4	Local Floating Pt. Opns	Medium	N/A
5	same as phase 3		
6	Local Floating Pt. Opns	Medium	N/A

Figure 5: Algorithmic Characteristics of CG Kernel

number without any external data. Hence the data granularity is meaningless for this phase of the computation. Towards the end of this computation phase, a few global sums are calculated by using a logarithmic reduce operation. In step i of the reduction, a processor receives data from another which is a distance 2^i away and performs an addition of the received value with a local value. The size of the data exchanged (data granularity) in these logarithmic communication steps is 4 bytes (an integer). 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 suggesting a near linear speedup curve on most machines (unless the processing speed is to reach unrealistic limits). Figure 3 summarizes the characteristics of the EP kernel.

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. The input data is equally partitioned among the processors. Each processor maintains its own copy of the buckets for the chunk of the input list that is allocated to it. Hence, updates to the buckets for the chunk of data allocated to a processor is an entirely local operation to the processor. This computation phase is again linear in the problem size but the granularity of the computation is not as intensive as in EP. The processing of each list element needs only the update (an integer addition) of the corresponding local bucket. The buckets are then merged using a logarithmic reduce operation and propagated back to the individual processors. The logarithmic operation takes place

as in EP, the difference being in the computation granularity and the data granularity (size of the messages exchanged). The message size (data granularity) in the communication steps is 8Kbytes (2K integers). The computation granularity of the reduction is not a simple addition as in EP, but involves an integer addition for each of the buckets. This can lead to non-trivial algorithmic work imbalance depending on the chosen bucket size. The data size is chosen to be 64Kbytes with 2K buckets to illustrate this work imbalance. Each processor then uses the merged buckets to calculate the rank of an element in its chunk of the input list. This phase of the kernel exhibits the same characteristics as the first computation phase (updating the local buckets). Figure 4 summarizes the characteristics of the IS kernel.

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. This kernel lies between EP and IS with respect to computation to communication ratio requirements. The sparse matrix and the vectors are partitioned by rows assigning an equal number of contiguous rows to each processor. The kernel performs twenty five iterations in trying to approximate the solution of a system of linear equations using the Conjugate Gradient method. 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 calculating 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. The computation granularity in the reduction is a floating point addition and the data granularity is 8 bytes (size of a double precision number). 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 involves the elements of the input vector that are not local to a processor. Hence before the computation, the different portions of the input vector present on different processors are merged globally using a

logarithmic reduce operation and the complete vector is replicated on each processor. The matrix-vector operation can then be carried out with entirely local operations. The logarithmic reduce operation for the merging does not have any computational granularity, but the data granularity doubles after each step of the operation. Initially the size of the messages is equal to the number of rows present on each processor ($11200/p$ bytes for $1400/p$ double precision numbers where p is the number of processors). After each step, the size of this message doubles since a processor needs to send the data that it receives along with its own local data to a processor that is at a distance a power of 2 away. Figure 5 summarizes the characteristics for each iteration of the CG kernel.

5 Architectural Characteristics

A uniprocessor architecture is characterized by: processing power as indicated by clock speed, instruction sets, clocks per instruction, floating point capabilities, pipelining, on-chip caches; memory size and bandwidth; and input-output capabilities. Parallel architectures have many more degrees of freedom making it difficult to study each artifact. Since uniprocessor architecture is getting standardized with the advent of RISC technology, we fix most of the processor characteristics by using the 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.

To illustrate the top-down approach, we use two message-passing architectures with different interconnection topologies: the *bus* and the *binary hypercube*. The bus platform consists of a number of nodes that are connected by a single 64-bit wide bus. Each processor in a node consists of a SPARC processor with local memory. The bus is an asynchronous Sequent-like bus (split transaction) with a cycle time of 150 nanoseconds. The cube platform closely resembles an iPSC/860 in terms of its communication capabilities. The nodes are connected by serial links with a bandwidth of 2.8 MBytes/sec in a binary hypercube topology. Message transmission uses a circuit-switched wormhole routing strategy. We have chosen these two platforms because they provide very different communication characteristics. The bus provides a much higher bandwidth compared to a single link of the cube, but the latter is expected to provide more contention free transmission due to its multiple links. The software overhead incurred is 100 microseconds per message which is keeping in trend with existing message-passing machines.

Both platforms provide an identical message-passing interface to the programmer. They support blocking and non-blocking modes of message transfer. The semantics of these modes are the same as those available on an iPSC/860 [12]. A blocking send blocks the sender until the message has left the sending buffer. Such a send does not necessarily imply that the message has reached the destination processor or even entered the network. A blocking receive blocks until the message from a corresponding send is completely in the receiving buffer. A non-blocking send does not guarantee that the message has even left the user buffer and a non-blocking receive returns immediately to the user program even if the message has not been received.

Many message-passing parallel programs are easier to write if the underlying system provides *typed*-messages and selective blocking on *typed*-messages. Typed-messages make it easier to order messages instead of leaving the burden to the programmer. Both our platforms support this elegant facility. On a message receive, the processor picks up messages from its mailbox and queues them up until it finds a message of the type that it needs.

6 Performance Results

The simulations that have been carried out include execution of the three NAS kernels on the two message-passing platforms. We report results for two different processor speeds: one at the native SPARC speed and the second at 10 times the native SPARC speed.

Figures 8, 9 and 10 show the speedups of the three NAS kernels on the two hardware platforms. The curves labeled "ideal" in these Figures have been calculated using the ideal time given by SPASM. The curves show the maximum possible speedup that could be obtained for the given parallel program (a purely algorithmic characteristic). As explained by the characteristics of these kernels in section 4, the "ideal" curve is observed to be almost linear for the EP kernel (Figure 8) and slightly deviates from being linear for the CG kernel (Figure 10) up to 64 processors. For the IS kernel (Figure 9) with the given problem size, the work imbalance in the program dominates, yielding maximum performance at around 30 processors. Further increase in number of processors results in a slowdown. The architectural overheads arise due to the communication in the problem and result in a deviation from the algorithmically predicted speedup curve (labeled "ideal"). EP has a high computation to communication ratio thus yielding speedup curve (for both bus and cube) close to the ideal speedup with the processor running at SPARC speed (1X). CG is more communication bound showing speedup curves that are significantly worse than the ideal

speedup curve. The deviations from the ideal curve for IS lie in-between that for EP and CG. For this problem, the speedup curves are limited more by the algorithm than by the architectural overheads. Increasing the processing speed to 10 times the SPARC speed (10X), progressively reduces the computation to communication ratio for all the kernels, thus yielding worse speedup curves (see corresponding 1X and 10X curves in Figures 8, 9 and 10). The EP kernel, which uses short messages (4 bytes) for its prefix computations, shows practically no difference in speedups between the bus and cube platforms. On the other hand, the poor point-to-point bandwidth of the cube compared to the bus plays an important role in degrading the performance of the other two kernels which send messages of much larger lengths (see Bus 1X and Cube 1X curves in Figures 8 and 9).

Figures 11, 12, and 13 show the latency overhead of the architecture on the respective kernels with the processor running at the native SPARC speed. These curves have been drawn for the processor that observes the maximum latency in each case. In all the kernels, the network latency ($f_l(p)$) of a processor is almost identical to the latency overhead function ($f_L(p)$) observed by a processor indicating that there is minimal overlap of computation with communication. The communication in all three kernels occurs only in the logarithmic reduce operations. The difference among them is in the size of the messages exchanged in this operation and the bandwidth of the interconnect. Since the number of messages received by a processor grows logarithmically with the number of processors, all the curves show a logarithmic behavior. The curves for latencies on the bus and the cube (see Figure 11) are almost identical for EP. This is due to the short messages (4 bytes) used by EP for its data exchanges. The software overhead of 100 microseconds per message on both platforms is the more dominating factor obviating the difference in the two hardware bandwidths. On the other hand, for IS and CG which send longer messages, there is a considerable disparity between the bus and cube for network latency (see Figures 12 and 13).

Figures 14, 15 and 16 show the contention overhead of the architectures on the respective kernels with the processor running at the native SPARC speed. A logarithmic reduce operation exchanges messages between processors that are at a distance a power of two apart. Such an operation can be elegantly mapped on to the cube to be entirely contention free. On the other hand, all the messages have to be sequentially handled on the bus giving rise to growing contention with increasing number of processors. As with latency, the network contention curves ($f_c(p)$) and the contention overhead curves ($f_C(p)$) are almost identical for the three kernels. There is negligible hiding of contention

due to overlap of computation with communication. Only IS exhibits any hidden contention for the 64 processor case (around 5% of the overall contention). The shape of the curves shows that the contention overhead on the bus grows faster than linear for all three kernels. Latency, which is a logarithmically growing function, is soon overtaken by the faster than linear growing contention function (at around 40 processors for IS and at around 12 processors for CG).

Figures 17, 18 and 19 show the breakup of the times due to the algorithmic and interaction overheads for the three kernels on the bus. Figures 20, 21 and 22 depict the same information for the cube. The timings shown are for a representative processor that executes the workload that is characteristic of the specific kernel. Note that this may not necessarily be the one that takes the longest time nor the one that experiences the maximum overheads. It is the processor that spends maximum time for computation among all the parallel processors. For EP, the overheads are marginal and the bulk of the time is largely due to the computation in the algorithm. For the other two kernels on the bus, contention becomes a bigger problem than latency with increasing number of processors as explained earlier. For large number of processors a considerable wait time is seen. The kernels consist of computation phases and communication phases. All the computation phases are load balanced among the processors and they arrive at a communication phase around the same time. The work-imbalance overhead (W_w) is mainly due to the logarithmic reduce operation where the number of processors participating is halved at each step. This intuitively suggests that the most of the wait time is due to latency (W_l) and contention (W_c). The measurements (see Figures 6 and 7) confirm this intuition. These measurements are for 64-node bus and cube systems for all the three kernels.

Kernel	W_w	W_l	W_c
EP	0.0%	100.0%	0.0%
IS	0.0%	31.1%	68.9%
CG	0.4%	34.4%	65.2%

Figure 6: Wait Times on the Bus

Figures 17, 18 and 19 also show the relative impact of the latency and contention overhead functions on performance. For smaller number of processors, latency is a more dominant factor than contention in limiting performance. But as mentioned earlier, the latency grows logarithmically (because of the structure of the algorithms) and is soon superseded by the faster than linear growing

Kernel	W_w	W_l	W_c
EP	0.0%	100.0%	0.0%
IS	0.0%	100.0%	0.0%
CG	0.2%	99.8%	0.0%

Figure 7: Wait Times on the Cube

contention overhead function. This transition occurs at around 40 processors for IS and at around 12 processors for CG on the bus platform. Latency and contention overhead have very little effect on the performance of EP.

To understand the effect of varying the latency on the contention overhead function, we simulate a 64-node bus platform and study the three kernels. Figure 23 shows the result of this simulation. the overheads are given in seconds for CG, in milliseconds for IS and in microseconds for EP. An interesting observation from this graph is that the contention overhead seen by a processor increases linearly with an increase in the latency of the underlying hardware. The contention overhead is affected the least for CG even though the net latency seen by a processor is the maximum of the three kernels (see Figure 23). IS exhibits the maximum change in contention overhead while EP falls in between.

6.1 Validation

We validate our simulation by executing the kernels on comparable parallel machines, and present sample validation results in this subsection. Figures 24, 25 and 26 compare the execution times for the EP, IS and CG kernels respectively on an iPSC/860 and on SPASM simulating an iPSC/860. The curves are identical for EP while there is around a 10-15% deviation for CG and around 15-20% deviation for IS. However, the shapes of the real and simulated curves are very similar indicating that trends predicted by the simulation are accurate within a constant factor. The deviation is largely due to inaccuracies in our estimation of the time taken for execution of the processor instruction streams. As mentioned in section 3, we use the special (simulated) instructions to update the simulation clock of a processor for the instructions that are executed at the speed of the native processor. If we are to use UNIX system calls to measure this time interval, then we are limited by the least count of the UNIX timers. The least count of the UNIX timer calls on the SPARCstation is in milliseconds and this can severely impact our measurements. Hence, we have

resorted to calculating these time quanta manually and introducing the appropriate instrumentation code in our source programs. These manual measurements may have contributed to the inaccuracies in the estimation. We propose to use the augmentation technique used in other similar simulation studies [6, 8] to overcome these inaccuracies.

7 Concluding Remarks

Theoretical and analytical models for studying the scalability of parallel systems have their limitations. We have proposed a new top-down approach to identify and quantify the different overheads in a parallel system that affects its scalability. We have used a combination of execution-driven simulation and experimentation to implement this approach. We use experimentation to understand the performance implications of real applications on real architectures, and to identify interesting kernels occurring in such applications. The kernels are then used in our simulation to separate the different overheads that cause non-ideal behavior. We have developed a simulation platform (SPASM) to conduct this study. SPASM provides an elegant way of isolating the algorithmic overhead and interaction overhead in a parallel system, and further separating them into their respective components.

We illustrate our approach by simulating a bus-based and a hypercube-based message passing platforms on SPASM. Using the NAS parallel kernels we isolate the algorithmic effects such as serial and work-imbalance overheads and the interaction effects such as latency and contention.

8 Future Work

Currently the physical memory available on the SPARCstations limits the system size as well as the problem size that can be simulated using SPASM. We are exploring alternative techniques, such as virtual memory and parallel simulation, for structuring SPASM to overcome this limitation. There are several interesting directions for extending this work. One is to identify and quantify other overheads in a parallel system such as scheduling, synchronization, task granularity (computation and data granularity) and caching. Another direction is to include shared memory style implementations, and incorporate different interconnection network topologies into SPASM.

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] M. Berry et al. The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. *International Journal of Supercomputer Applications*, 3(3):5–40, 1989.

[6] 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.

[7] 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.

[8] 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.

[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*, May 1993. To appear.

[10] 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.

[11] 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.

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

[13] Leah H. Jamieson. Characterizing Parallel Algorithms. In L. H. Jamieson, D. B. Gannon, and R. J. Douglas, editors, *The Characteristics of Parallel Algorithms*, pages 65–100. MIT Press, 1987.

[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] H. T. Kung. The Structure of Parallel Algorithms. *Advances in Computers*, 19:65–112, 1980. Edited by Marshall C. Yovits and Published by Academic Press, New York.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[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] 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*, Santa Clara, CA, May 1993. To appear.

[23] 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.

[24] 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.

[25] Anand Sivasubramaniam, Umakishore Ramachandran, and H. Venkateswaran. A Computational Model for Message-Passing. In *Proceedings of the Sixth International Parallel Processing Symposium*, pages 358–361, Beverly Hills, California, March 1992.

[26] 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.

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

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

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

[30] 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.

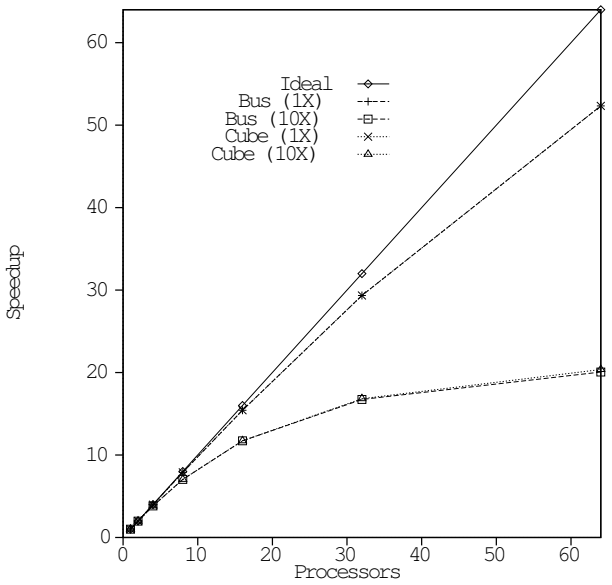

Figure 8: EP : Speedup

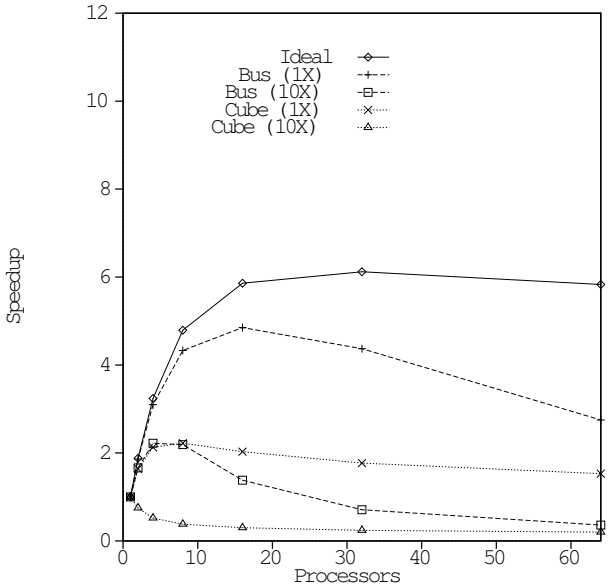

Figure 9: IS : Speedup

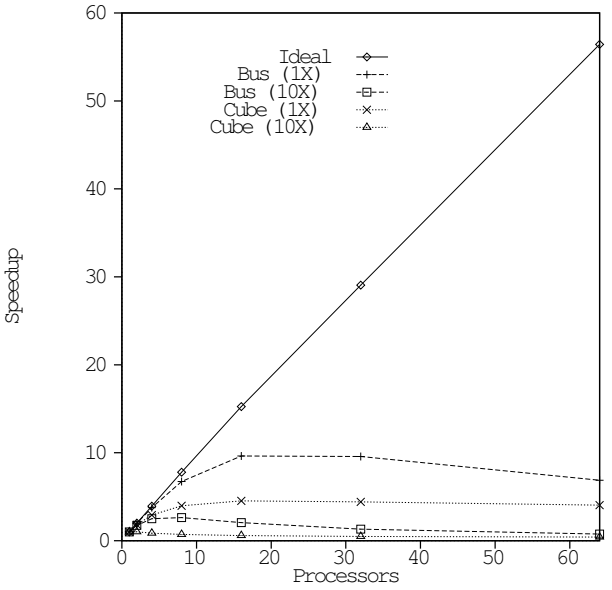

Figure 10: CG : Speedup

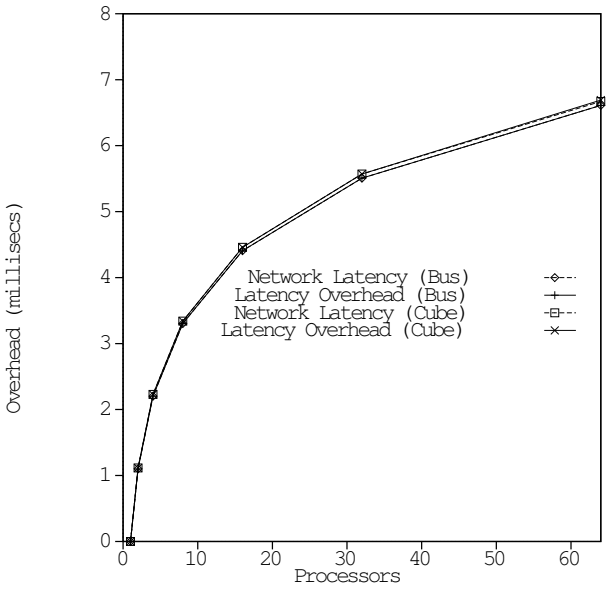

Figure 11: EP : Latency

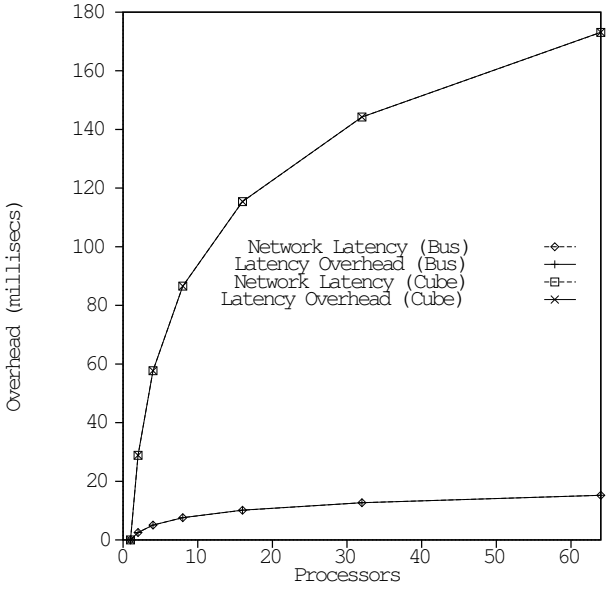

Figure 12: IS : Latency

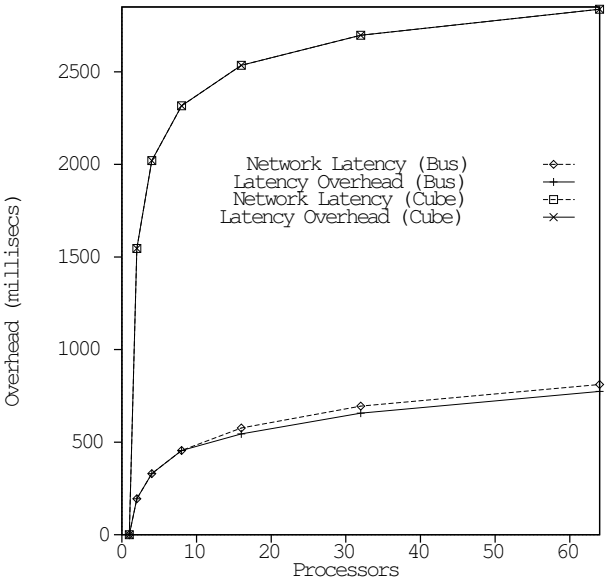

Figure 13: CG : Latency

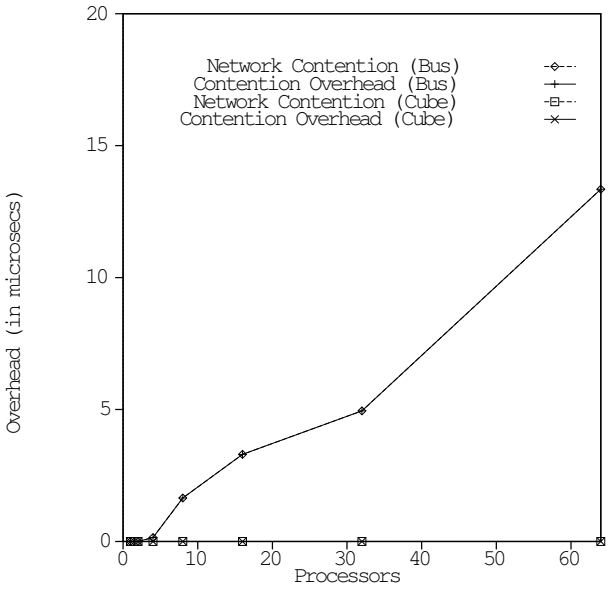

Figure 14: EP : Contention

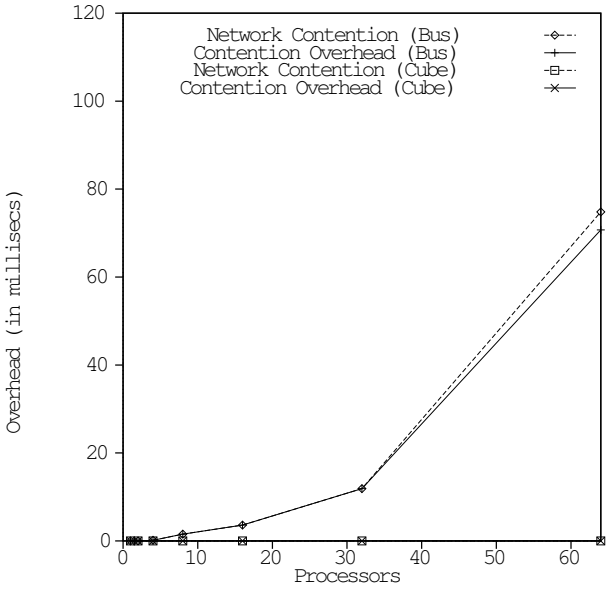

Figure 15: IS : Contention

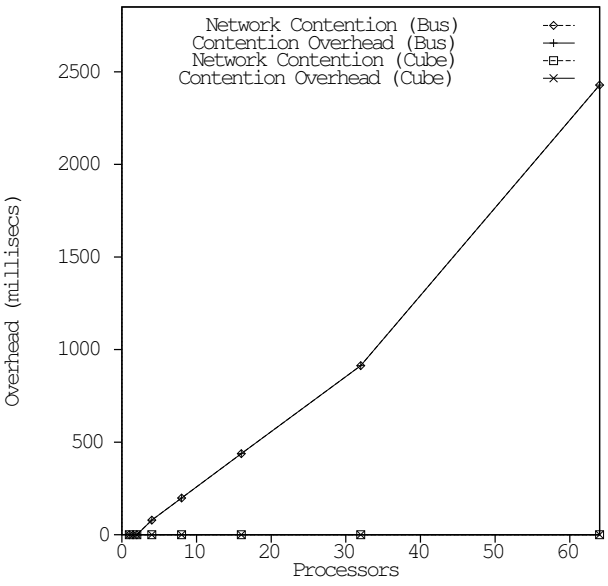

Figure 16: CG : Contention

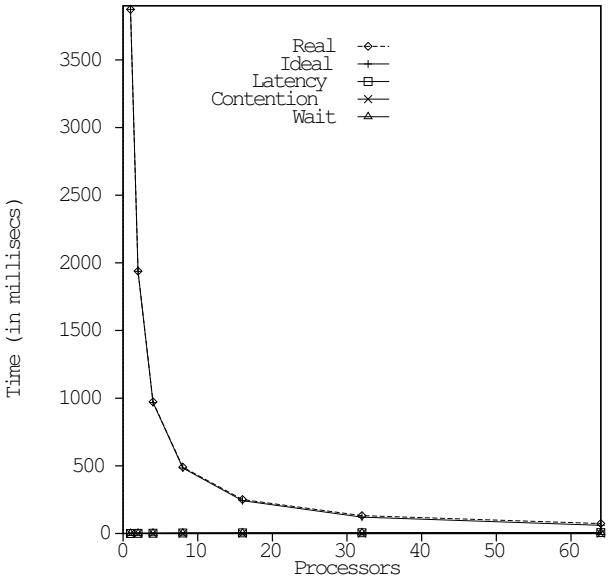

Figure 17: EP : Overheads on Bus

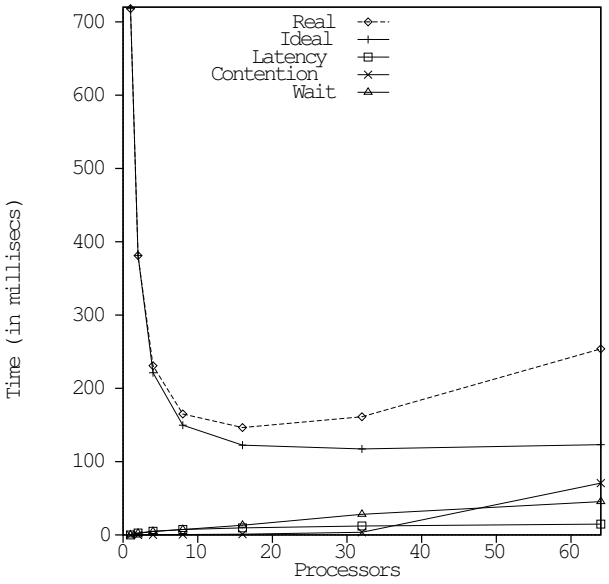

Figure 18: IS : Overheads on Bus

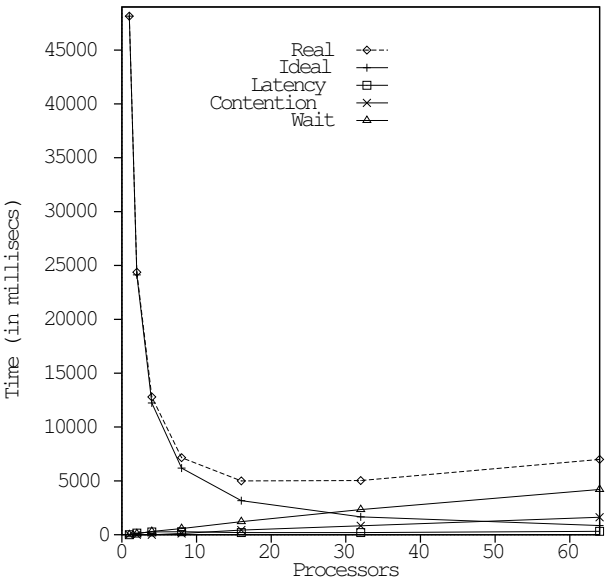

Figure 19: CG : Overheads on Bus

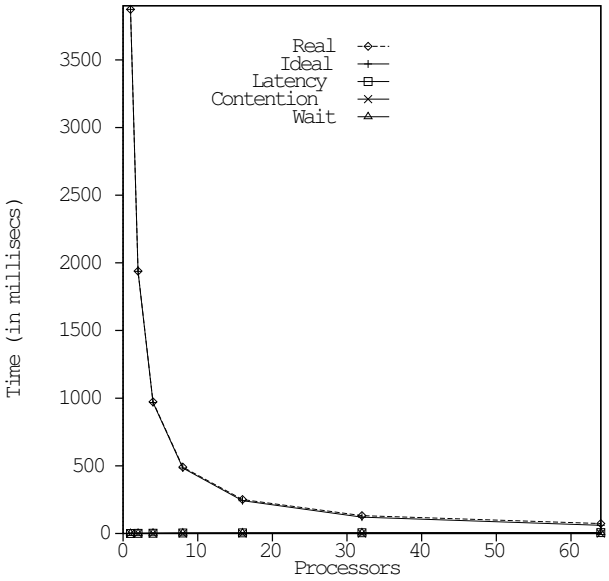

Figure 20: EP : Overheads on Cube

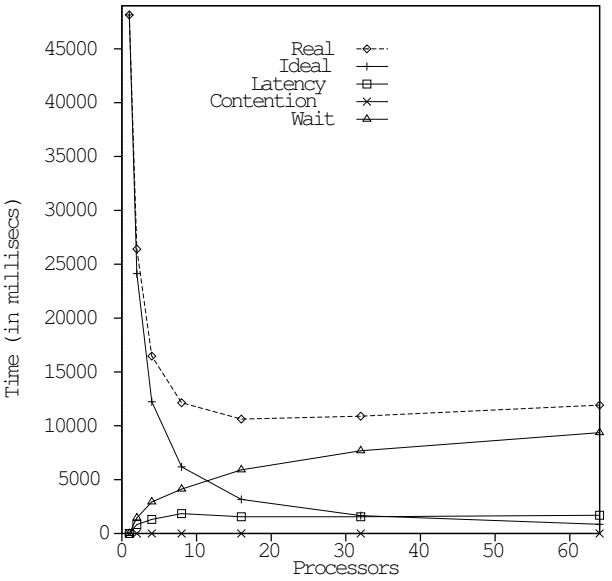

Figure 22: CG : Overheads on Cube

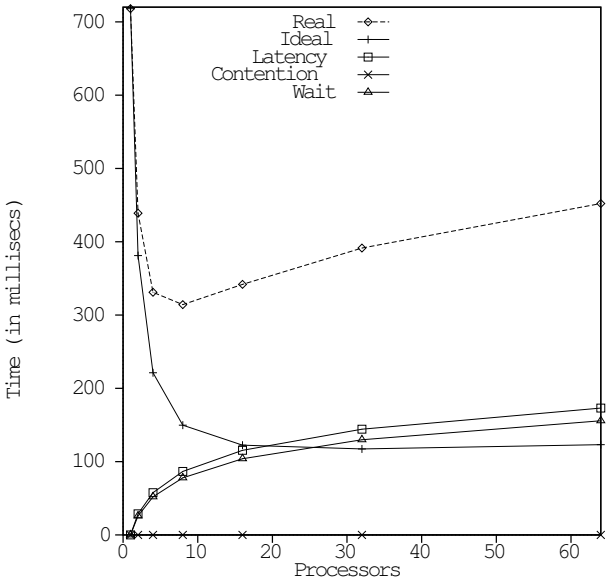

Figure 21: IS : Overheads on Cube

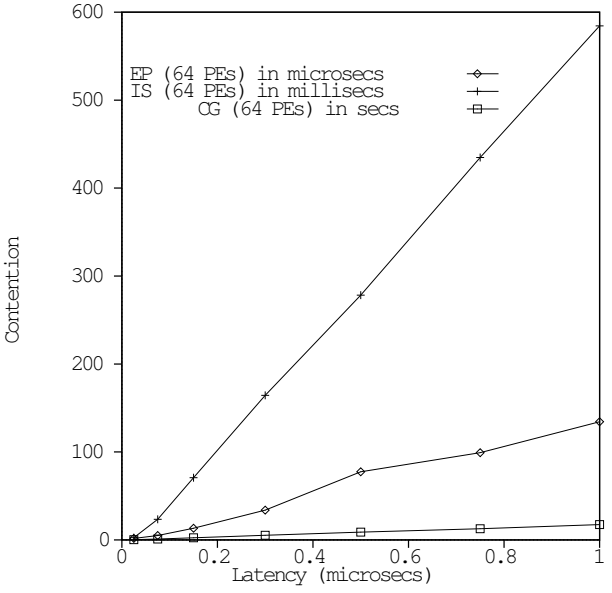

Figure 23: Effect of Latency on Contention

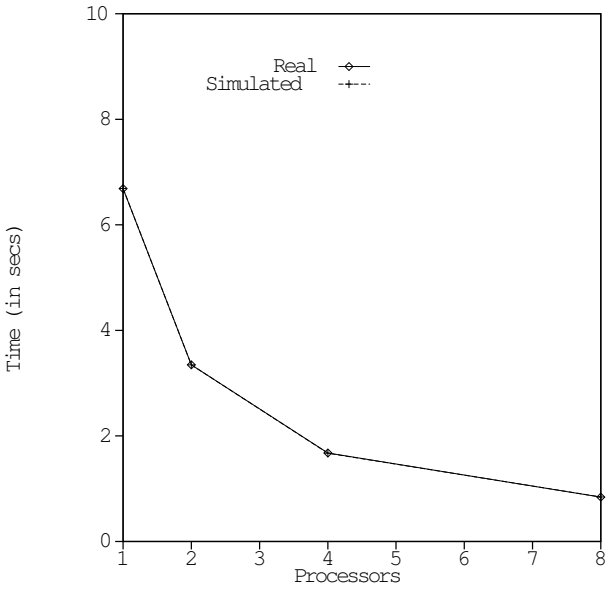

Figure 24: EP on Cube : Validation

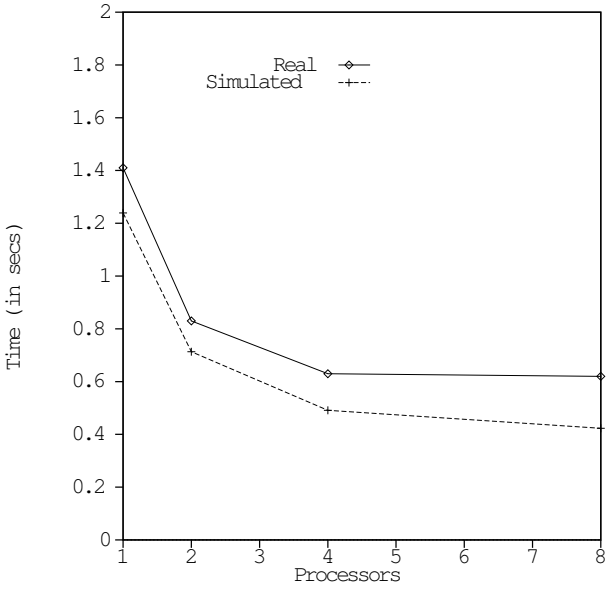

Figure 25: IS on Cube : Validation

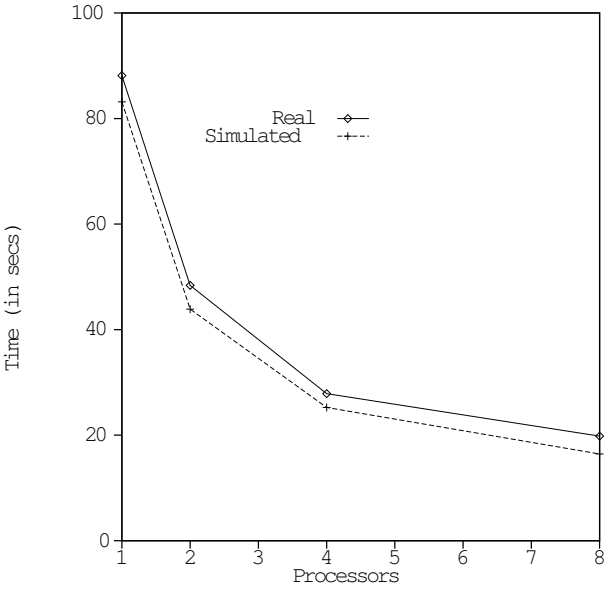

Figure 26: CG on Cube : Validation