In *Journal of Parallel and Distributed Computing*, Vol 22(3):411-426, September 1994.

A Simulation-based Scalability Study of Parallel Systems[*]

Anand Sivasubramaniam
Aman Singla
Umakishore Ramachandran
H. Venkateswaran

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 each artifact of the parallel system that limits its scalability. 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 using real applications. 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 a proliferation 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. 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 due to additional degrees of freedom. 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 tractable. *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, identify algorithmic and architectural bottlenecks, 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 understand the scalability of other algorithm-architecture pairs.

Performance metrics such as speedup [2], scaled speedup [11], sizeup [28], experimentally determined serial fraction [14], and isoefficiency function [15] have been proposed for quantifying the scalability of parallel systems. While these metrics are extremely useful for tracking performance trends, they do not provide adequate information needed to understand the reason why an algorithm does not scale well on an architecture. 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.

Several performance studies address issues such as latency, contention and synchronization. The limits on interconnection network performance [1, 21] and the scalability of synchronization primitives supported by the hardware [3, 19] are examples of such studies undertaken over the years. 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

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

system performance [6]. Cypher et al. [9], identify the architectural requirements such as floating point operations, communications, and input/output for message-passing scientific applications. Rothberg et al. [23] 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.

Since real-life applications set 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 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 Architectural 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 separate the algorithmic overhead into a *serial* component and a *work-imbalance* component. We also isolate the overheads due to *network latency* (the actual hardware transmission time of a message 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 in the context of real applications, 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.

This work is part of a larger project which aims at understanding the significant issues in the design of scalable parallel systems using the above-mentioned top-down approach. In our earlier work, we studied issues such as task granularity, data distribution, scheduling, and synchronization, by implementing frequently used parallel algorithms on shared memory [25] and message-passing [24] platforms. In [27], we illustrate the top-down approach for the scalability study of shared memory systems. In this paper, we conduct a similar study for message-passing systems.

In a related paper [26] we evaluate the use of abstractions for the network and locality in the context of simulating cache-coherent shared memory multiprocessors.

We illustrate the top-down approach through a case study, implementing three 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. Based on these results we present the scalability implications for these two systems with respect to these kernels in Section 7. Concluding remarks and directions for future research are given in Sections 8 and 9.

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 [18]) and evaluating their performance. As one goes lower in the hierarchy, 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, in particular the NAS parallel kernels [4] that have been derived from a large number of Computational Fluid Dynamics applications.

Parallel system overheads 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 due to the inherent *serial* part [2] and the *work-imbalance* in the algorithm. 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 by executing the algorithm on an ideal machine such as the PRAM [30] (the “ideal” curve in Figure 1). 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 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 devoid of network contention, the communication pattern of the application would dictate the latency overhead, and its performance may lie between the ideal curve and the real execution curve (see Figure 1).

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

- 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 exist in them. Experimentation, simulation, and analytical models are three techniques that have been commonly used for such studies. But each has its own limitations. We adopted the first technique in our earlier work by experimenting with frequently used parallel algorithms on shared memory [25] and message-passing [24] platforms. Experimentation is important and useful for scalability studies of existing architectures, but has certain limitations. The underlying hardware is fixed making it impossible to study the effect of changing individual architectural parameters, and it is difficult if not impossible to separate the effects of different architectural artifacts on the performance since we are constrained by the performance monitoring support provided by the parallel system. Further, monitoring program behavior via instrumentation can become intrusive yielding inaccurate results.

Analytical models have often been used to give gross estimates for the performance of large parallel systems. In general, such models tend to make simplistic assumptions about program behavior and architectural characteristics to make the analysis using the model tractable. These assumptions restrict their applicability for capturing complex interactions between algorithms and architectures. For instance, models developed in [17, 29, 8] are mainly applicable to algorithms with regular communication structures that can be predetermined before execution of the algorithm. Madala and Sinclair [17] confine their studies to synchronous algorithms while [29] and [8] develop models for regular iterative algorithms. However, there exist several applications [23] with irregular data access, communication, and synchronization characteristics which cannot always be captured by such simple parameters. Further, an application may be structured to hide a particular overhead such as latency by overlapping computation with communication. It may be difficult to capture such dynamic program behavior using analytical models. Similarly, several other models make assumptions about architectural characteristics. For instance, the model developed in [20] ignores data inconsistency that can arise in a cache-based multiprocessor during the execution of an application and thus does not consider the coherence traffic on the network.

The main focus of our top-down approach is to quantify the overheads that arise from the interaction between the kernels and architecture and their impact on the overall execution of the application. It is not clear that these overheads can be easily modeled by a few parameters. Therefore, we use simulation for quantifying and separating the overheads. 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. Simulation also has its limitations. It may not always be possible to predict system scalability with simulation owing to resource (time and space) constraints. But we believe that simulation can complement the analytical technique by using the datapoints obtained from simulation, which are closer to reality, as a feedback to refine existing models for predicting the scalability of larger systems. Further, our simulator can also be used to validate existing analytical models using real applications.

Our simulation platform (SPASM) provides an elegant set of mechanisms for quantifying the different overheads 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 [30] and measuring its deviation from a linear speedup curve. Further, we separate this overhead into that due to the serial part (*serial overhead*) and that due to work imbalance (*work-imbalance overhead*). The interaction overhead is also 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* and the *contention*

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

overhead function 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 such issues are beyond the scope of this paper. For the rest of this paper, we confine ourselves to 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 input to the simulator are parallel applications written in C, which are compiled and linked with the simulator libraries.

Architectural simulators have normally tended to be very slow, making it tedious to get a wide range of data points on realistic problems. Simulating the entire instruction set of a processor can slow down the simulation considerably. Since the thrust of this study is to understand 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. 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 and updates the simulation clock of the processor. This strategy has considerably lowered the simulation time which is at most a factor of two compared to the real time for the applications considered. This strategy has also been used in other recent simulation studies [5, 7, 22].

The novel feature of our simulation study is the separation of overheads in a parallel system. Providing the functionality that is needed for such a separation requires a considerable amount of instrumentation. There exist simulators in the public domain (such as Proteus [5]) that provide certain basic functionality. We have used CSIM as the starting point since the programming effort to provide the needed functionality is comparable whether we used a basic process-based simulation package such as CSIM or any other architecture simulator.

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 (NI). These three entities are implemented as CSIM processes. The CSIM process representing a processor executes the code associated with the corresponding thread of control in the parallel program. These processes execute at the speed of the native processor until they encounter an instruction that needs to be trapped by the simulator like a load/store on a shared memory platform or a send/receive on a message-passing platform. On a shared memory platform, the processor issues load/store requests to its cache module. The cache module services the request, invoking the network interface 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 interacts directly with the network interface. On a send, the processor creates a message (a data structure) and places it in a mailbox. The network interface 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 On a synchronous receive, the processor blocks until a message appears in its mailbox.

The system parameters that can be specified to SPASM are: the *number of processors (p),* the *clock speed* of the processor[3], the *hardware bandwidth* of the links in the network, the *switching delay*, the *software latency* for transmission of a message and the sustained *software bandwidth*.

3.1 Metrics

SPASM provides a wide range of statistical information about the program execution. 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.

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. If a receive is posted prior to the matching send, then the difference between the two times is due to skews

[3]Even though we assume that each processor is a SPARC chip, we can vary the clock speed of the simulated chip which provides us with a convenient mechanism for varying the computation to communication ratio and studying the scalability of future systems built with faster processors.

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. In a network with no contention, 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 when all other conditions are ideal. The sum of all these overheads incurred by a processor is called the *Network Latency* ($f_l(p)$). 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. 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 earlier 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 contention-free 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 for comparing the merits of different networks. 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 programming style 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.

EP is the “Embarrassingly 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 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). Table I 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). Table II 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 obtaining the intermediate dot products for the elements in the vectors local to a processor. This is again a local operation with a computation granularity linear in the number of rows assigned to a processor with a floating point multiplication and addition performed for each row. A global sum of the intermediate dot products is calculated by a logarithmic reduce operation (as in EP) yielding the final dot product. 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. Table III 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 and uses the e-cube routing algorithm. 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 have been carried out for the 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. 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.

Figures 3, 4 and 5 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 can 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 3) and close to linear for the CG kernel (Figure 5) up to 64 processors. For the IS kernel (Figure 4) 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 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 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 kernels, thus yielding worse speedup curves (see corresponding 1X and 10X curves in Figures 3, 4 and 5). 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 larger lengths (see Bus 1X and Cube 1X curves in Figures 3 and 4).

Figures 6, 7, and 8 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 between them is in the size of 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 curves show a logarithmic behavior. The curves for latencies on the bus and the cube (see Figure 6) 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 7 and 8).

Figures 9, 10 and 11 show the contention overhead of the architectures for the respective kernels with the processor running at the native SPARC speed. A logarithmic reduce operation exchanges messages between processors that are separated by a distance that is a power of two. Such an operation can be elegantly mapped on 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 12, 13 and 14 show the breakup of times due to algorithmic and interaction overheads for the three kernels on the bus. Figures 15, 16 and 17 depict the same information for the cube. The timings shown are for a representative processor which 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 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) incurred in previous messages. The measurements (see Tables IV and V) confirm this intuition. These measurements are for 64-node bus and cube systems for all three kernels.

Figures 12, 13 and 14 also show the relative impact of 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 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 overheads have very little effect on the performance of EP.

To understand the effect of varying the bus bandwidth on the contention overhead function, we simulate a 64-node bus platform for the three kernels and vary the cycle time of the bus. Figure 18 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 bus cycle time. The contention overhead is affected the least for CG even though the net contention seen by a processor is the maximum of the three kernels (see Figure 18). 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 19, 20 and 21 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 [5, 7] to overcome these inaccuracies.

7 Scalability Implications

In this section we illustrate how the top-down approach, in particular the overhead functions defined by the simulator, can be used for drawing conclusions regarding the scalability of a parallel system. It should be noted that these scalability projections are purely in the context of the specific algorithm-architecture pairs. For each of the three kernels, we discuss how the overhead functions are expected to grow with system size (both the number of processors as well as the problem size) based on the datapoints collected using the simulator. The computation time and the overhead functions are determined by interpolating these datapoints for each kernel. In reality, these functions are dependent on all aspects of a parallel system. However, for the sake of simplicity of analysis we express these as functions of the number of processors in the parallel system. The coefficients of these functions are constants for a given problem size and a given set of system parameters (such as link bandwidth).

7.1 EP Kernel

Table VI gives the computation time and the overhead functions for the EP kernel. Based on these functions, we can make the following observations:

- The computation time scales down linearly with the number of processors. In addition, it outweighs all the overheads and hence is the dominant factor in the execution time as can be seen from the coefficients associated with these functions.
- Since the communication is confined to the logarithmic global sum operation, the latency overhead grows logarithmically with the number of processors. The coefficients for the latency overhead on the bus and cube are the same since EP uses small sized messages (4 bytes) for this operation.
- On the cube, there is no contention overhead for the logarithmic communication operation while it grows linearly with the number of processors on the bus. However, the associated coefficient is so small that its effect is negligible in absolute terms as well as relative to the latency overhead. The contention overhead becomes dominant compared to the latency only beyond several thousand processors.
- As is shown in Tables IV and V, the wait time is more dependent on the latency overhead than contention for this kernel, thus growing logarithmically with system size.

From these observations, we can conclude that the bus and cube systems scale well with the number of processors for the EP kernel. Even for a relatively small problem size (64K in this case), it would take nearly a 1000 processors for the overheads to start dominating. While an increase in the problem size would increase the coefficient associated with the computation time, it does not have any effect on the coefficients of the other overhead functions. Therefore the bus and cube systems scale well with the problem size as well for this kernel.

7.2 IS Kernel

Table VII gives the computation time and the overhead functions for the IS kernel. Based on these functions, we can make the following observations:

- In the ideal execution of the IS kernel, the computation time scales down logarithmically with the number of processors.

- As with EP, the latency overhead function is logarithmic for IS. The difference being that the coefficients for this function are different for the bus and the cube. Since the individual link bandwidths on the cube are lower than the bus bandwidth, the longer messages used by IS incur a larger latency on the cube.

- The logarithmic communication does not incur any contention on the cube. But on the bus, the contention grows quadratically with the number of processors and exceeds the latency overhead beyond 40 processors.

- The wait time on the cube is purely dependent on the latency overhead (Table V) and is thus logarithmic. On the other hand, the wait time on the bus depends on the latency and contention overheads (Table IV) and the resulting function thus lies in between these two overhead functions.

From the above observations, we can conclude that the bus and cube systems are not scalable with respect to the IS kernel. For the bus system, the contention function (being quadratic) starts dominating the computation time with increasing system size. On the cube, despite the lack of contention, the coefficient associated with the latency overhead is significant compared to that associated with the computation time and becomes the limiting factor with increasing number of processors.

An increase in the problem size will increase the coefficients associated with the computation time as well as the latency overhead (since the messages in IS are dependent on the problem size). For the bus system, Figure 18 indicates that the contention increases linearly with the link latency. Therefore, increasing the problem size is likely to make both the cube and the bus systems less scalable for this kernel.

7.3 CG Kernel

Table VIII gives the computation time and the overhead functions for the CG kernel. Based on these functions, we can make the following observations:

- The computation time scales down linearly with the number of processors.

- As with the previous two kernels, the latency overhead is logarithmic in the number of processors.

- The contention overhead on the bus increases linearly with the number of processors and becomes the limiting factor beyond 10 processors. On the cube, contention is non-existent.

- The wait time on the cube is a logarithmic function (since there is no contention), while it is a linear function on the bus (since contention dominates latency in this case).

Even though the computation time for the CG kernel scales down linearly with the number of processors, the hardware overheads limit the scalability of this parallel system. However, increasing the problem size has a favorable impact on scalability of this kernel for both platforms. While the coefficient associated with the computation time increases quadratically with the problem size, those associated with latency and contention are likely to grow only linearly. Therefore, increasing the problem size is likely to make both the cube and the bus systems more scalable for this kernel.

8 Concluding Remarks

We have proposed a top-down approach to separate and quantify the different overheads in a parallel system that limit 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 illustrated our approach by simulating the NAS parallel kernels on a bus-based and a hypercube-based message-passing platform on SPASM, and isolated the algorithmic effects such as serial and work-imbalance overheads and the interaction effects such as latency and contention.

9 Future Work

Currently, we are limited by the resource constraints inherent in sequential simulation for simulating large parallel systems. We are exploring the viability of both conservative and optimistic methods of parallel simulation on different hardware platforms for overcoming 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, and caching. Another direction is to include a wider range of hardware platforms and a broader application domain.

Acknowledgements

The authors would like to thank the anonymous referees for their comments, which helped us put the results of this work in the proper perspective in addition to improving the quality of the presentation.

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

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

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

[15] V. Kumar and V. N. 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] S. Madala and J. B. Sinclair. Performance of Synchronous Parallel Algorithms with Regular Structures. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):105–116, January 1991.

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

[19] 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.

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

[21] 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.

[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*, pages 48–60, Santa Clara, CA, May 1993.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. An Approach to Scalability Study of Shared Memory Parallel Systems. In *Proceedings of the ACM SIGMETRICS 1994 Conference on Measurement and Modeling of Computer Systems*, 1994. To appear.

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

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

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

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

Table I: Characteristics of EP

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

Table II: Characteristics of IS

Phase	Description	Comp. Gran.	Data Gran.
1	Local Floating Pt. Opns	Medium	N/A
2	Matrix-Vector Product		
	a) Global Vector Merge	N/A	$(11200/p) * 2^i$ in step i
	b) Local Matrix-Vector Product	Medium	N/A
3	Vector-vector dot product		
	a) Local vector-vector dot product	Small	N/A
	b) 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

Table III: Characteristics of CG

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%

Table IV: Wait Times on the Bus

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%

Table V: Wait Times on the Cube

EP	Bus	Cube
Comp. Time	$3873/p$	$3873/p$
Latency	$1.1 \log p$	$1.1 \log p$
Contention	$0.15 * 10^{-3} \lfloor p/2 \rfloor$	0
Wait	$\log p$	$\log p$

Table VI: EP : Overhead Functions (in millisecs)

IS	Bus	Cube
Comp. Time	$717/(1 + \log p)$	$717/(1 + \log p)$
Latency	$2.5 \log p$	$28.9 \log p$
Contention	$0.01 * p^2$	0
Wait	$2.28 * \lfloor p/2 \rfloor$	$26 \log p$

Table VII: IS : Overhead Functions (in millisecs)

CG	Bus	Cube
Comp. Time	$48146/p$	$48146/p$
Latency	$65 \log p$	$500 \log p$
Contention	$45 * \lfloor p/2 \rfloor$	0
Wait	$75 * \lfloor p/2 \rfloor$	$1400 \log p$

Table VIII: CG : Overhead Functions (in millisecs)

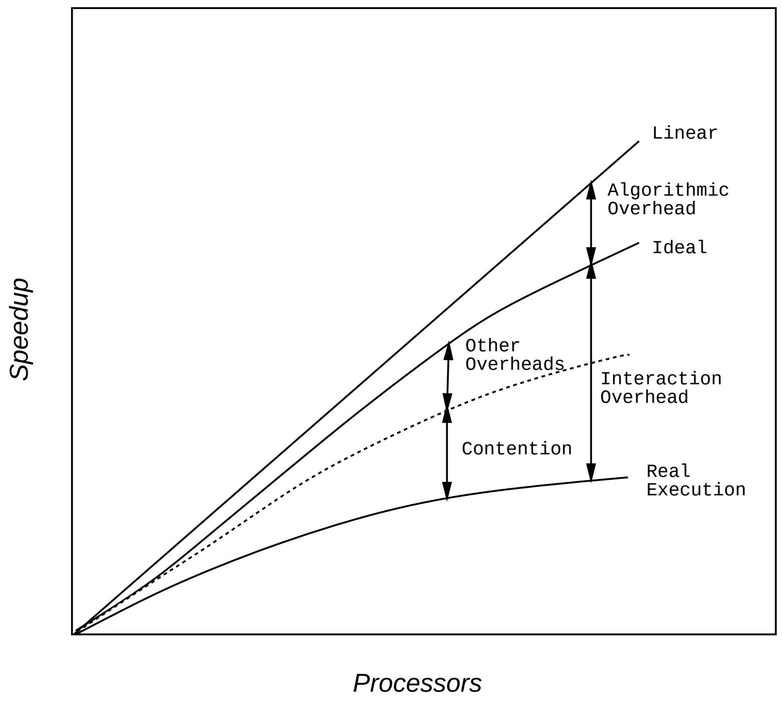

Figure 1: Top-down Approach to Scalability Study

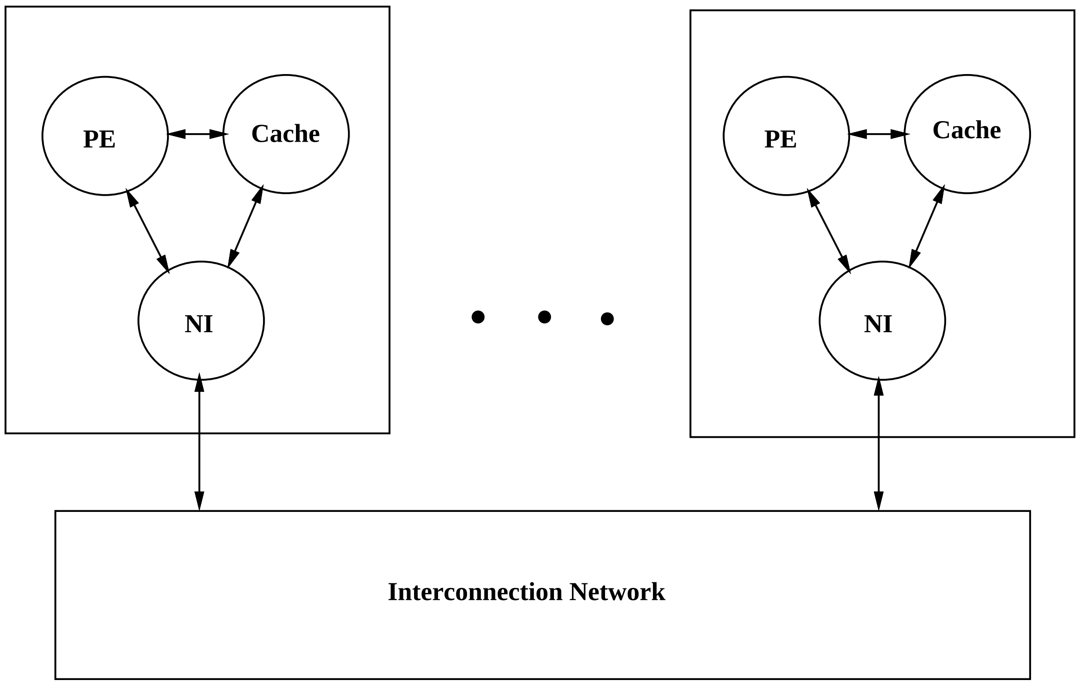

Figure 2: Architecture of SPASM

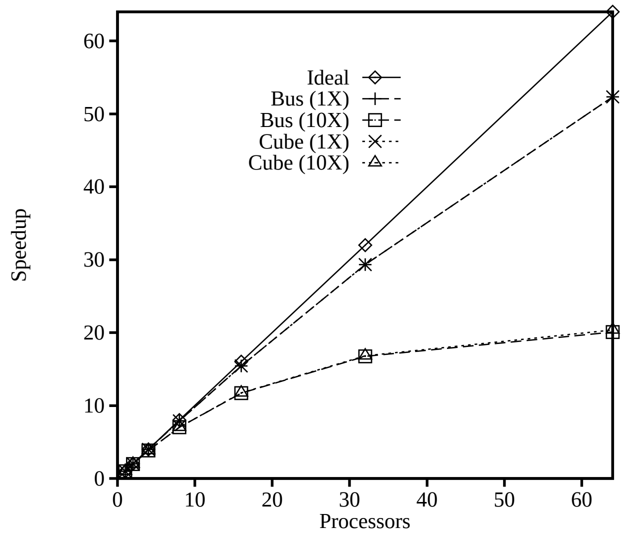

Figure 3: EP: Speedup

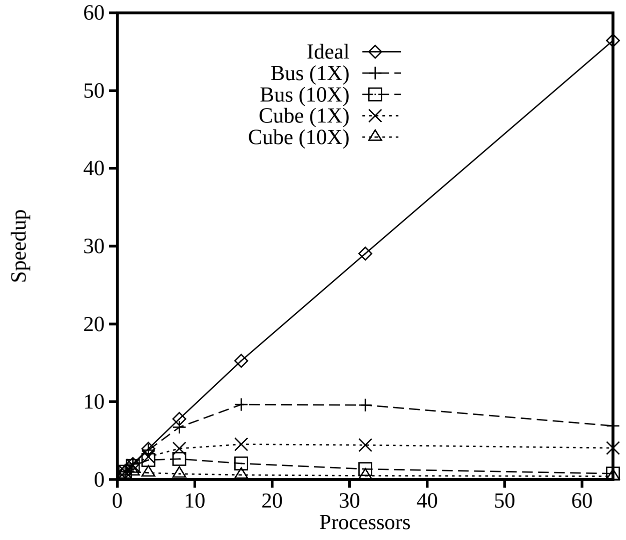

Figure 5: CG: Speedup

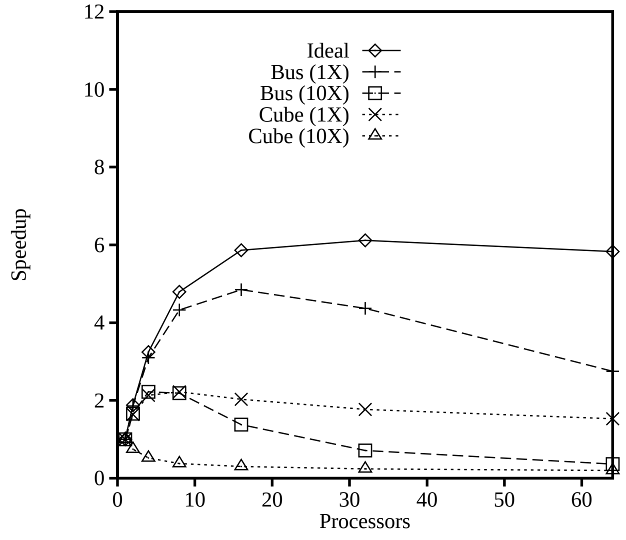

Figure 4: IS: Speedup

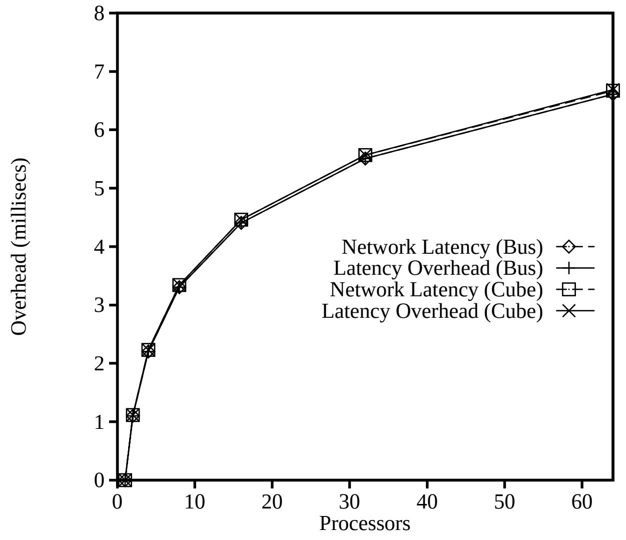

Figure 6: EP: Latency

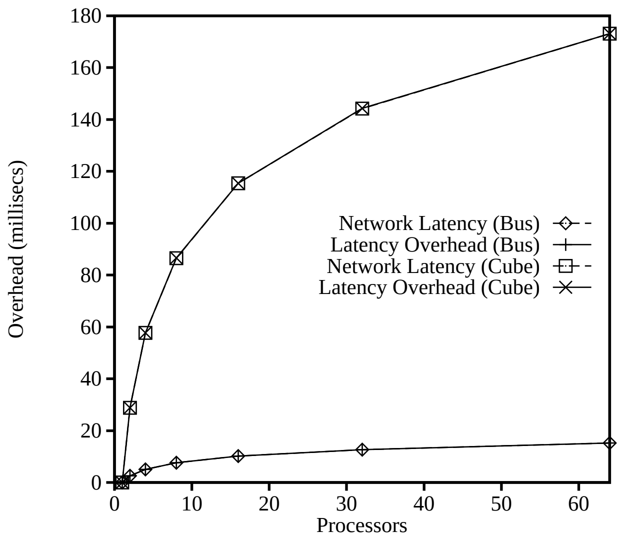

Figure 7: IS: Latency

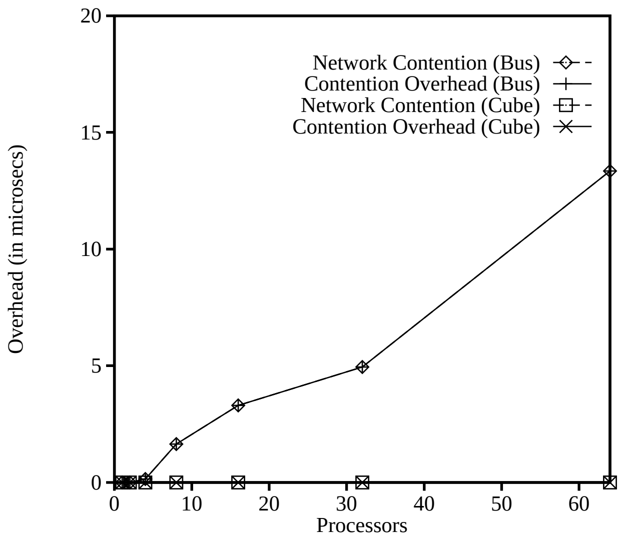

Figure 9: EP: Contention

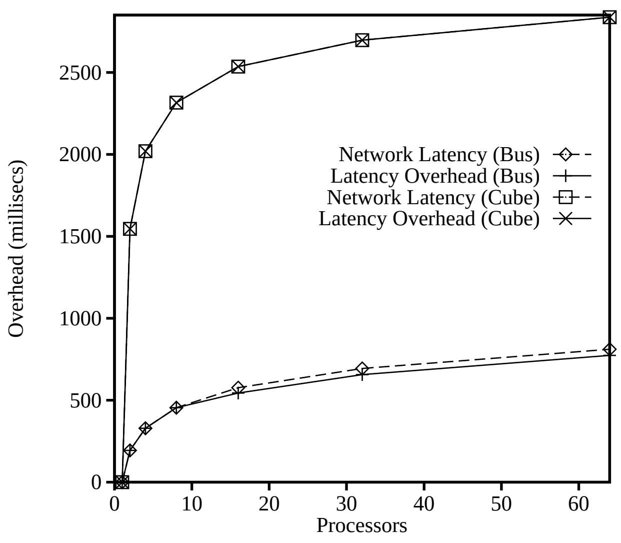

Figure 8: CG: Latency

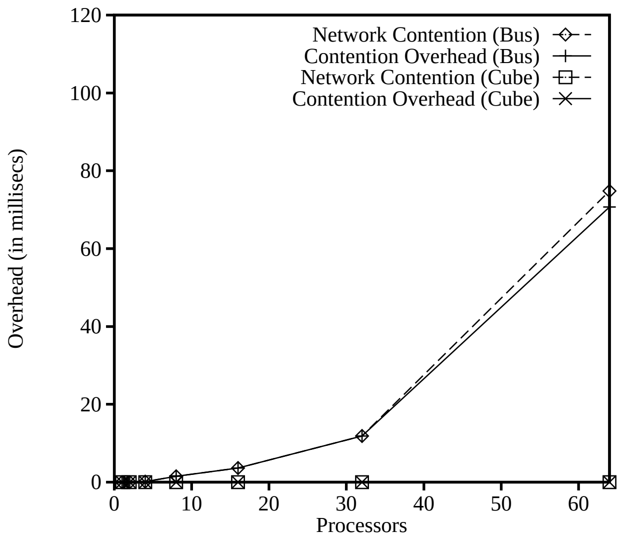

Figure 10: IS: Contention

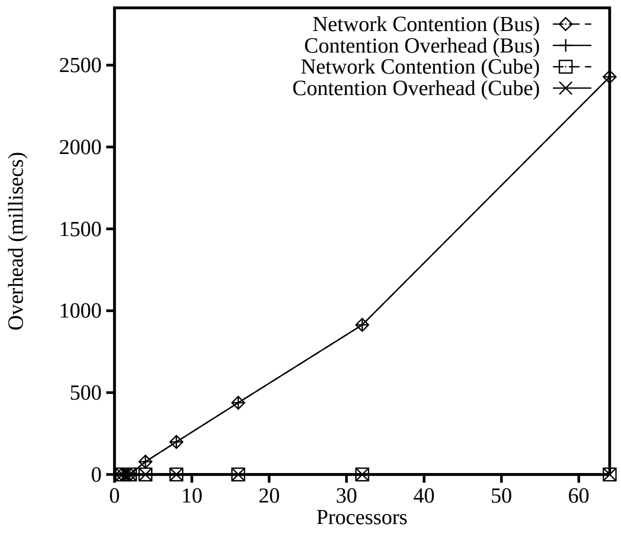

Figure 11: CG: Contention

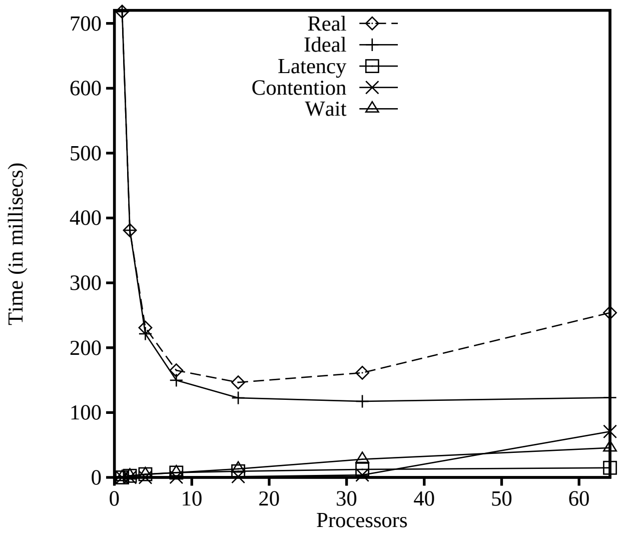

Figure 13: IS: Overheads on Bus

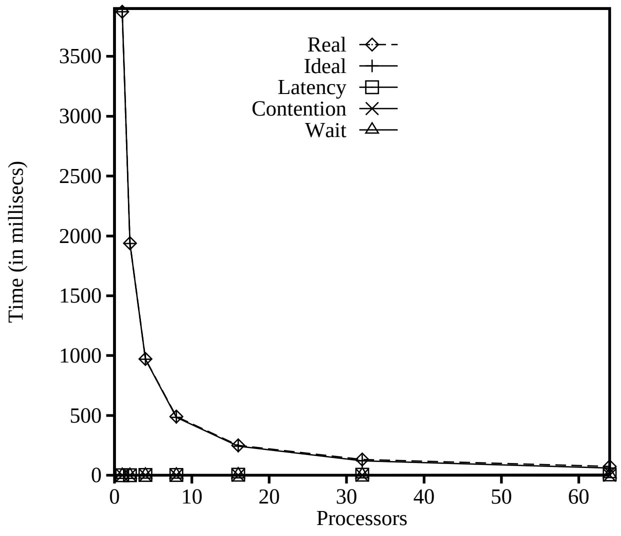

Figure 12: EP: Overheads on Bus

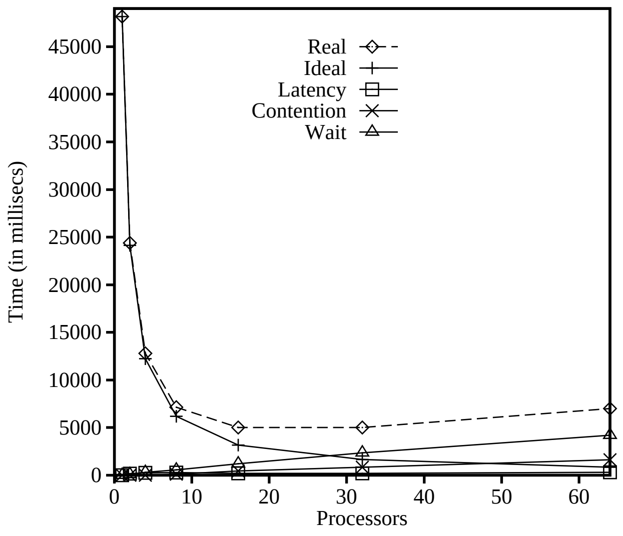

Figure 14: CG: Overheads on Bus

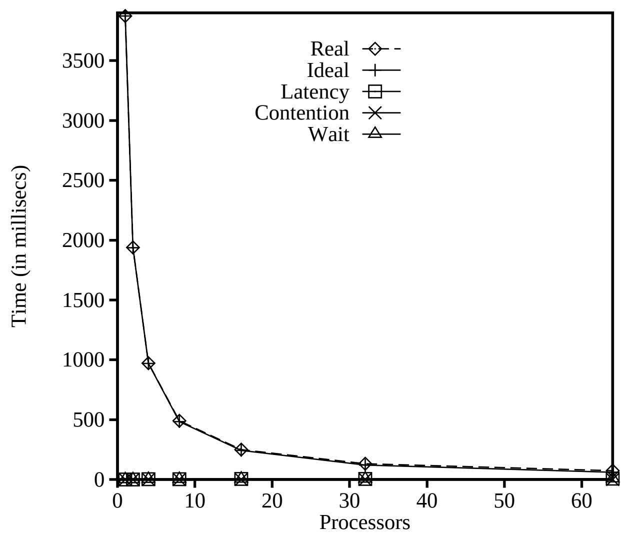

Figure 15: EP: Overheads on Cube

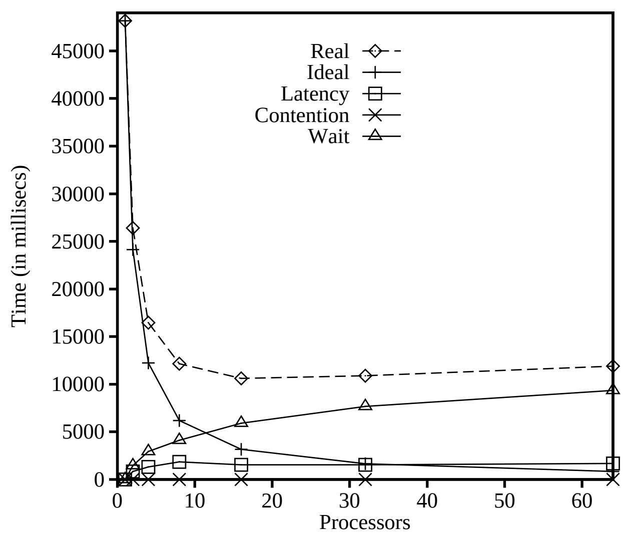

Figure 17: CG: Overheads on Cube

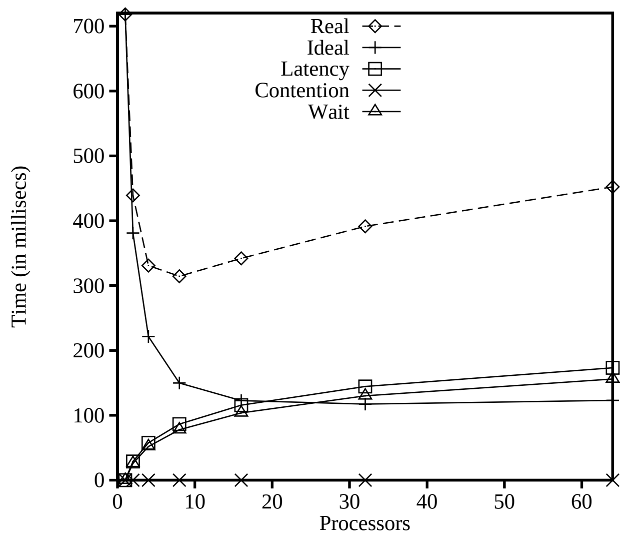

Figure 16: IS: Overheads on Cube

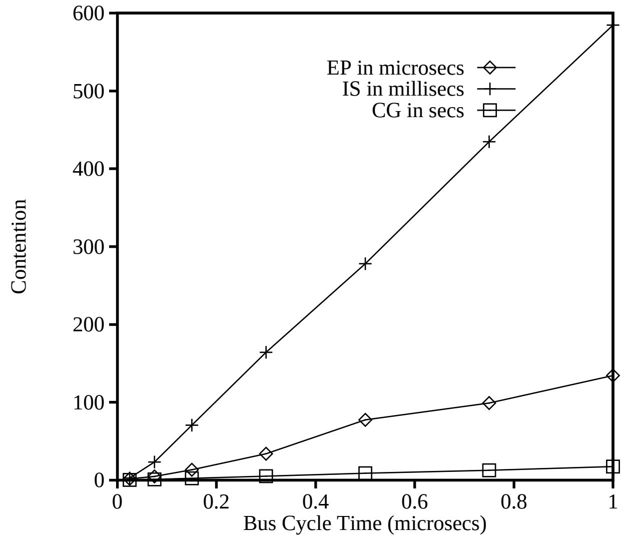

Figure 18: Effect of Bus Cycle Time on Contention (64 PEs)

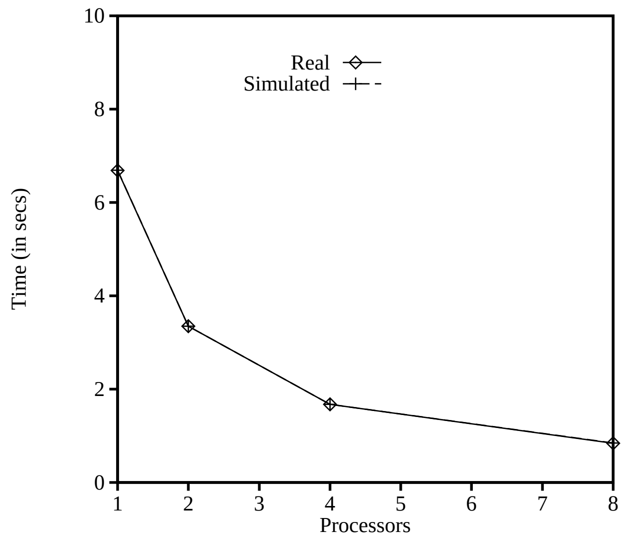

Figure 19: EP on Cube: Validation

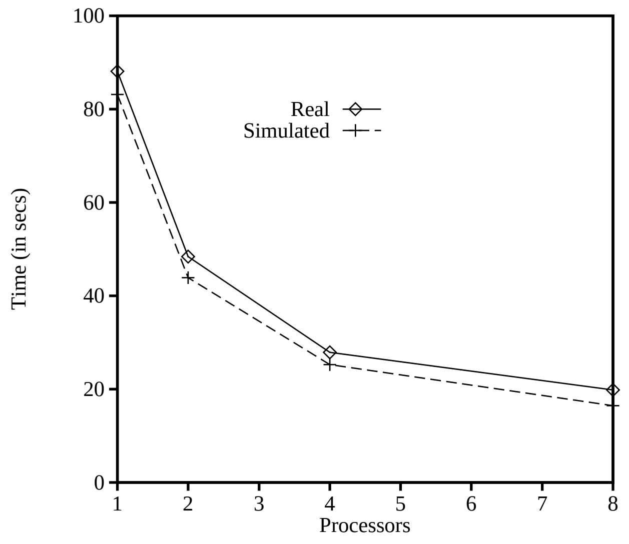

Figure 21: CG on Cube: Validation

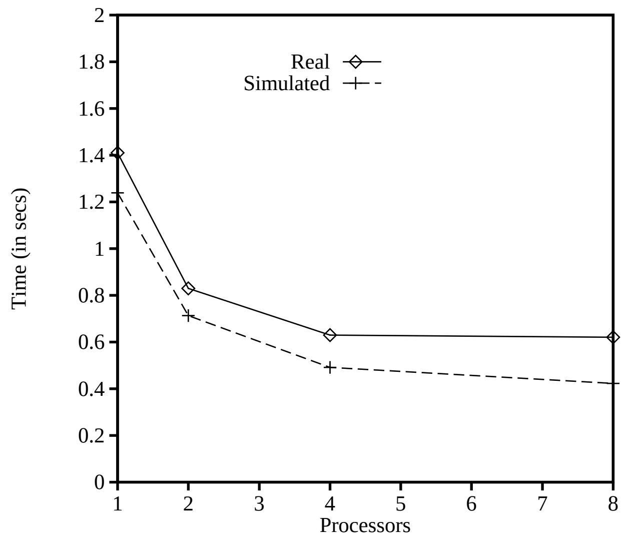

Figure 20: IS on Cube: Validation