

# Machine Abstractions and Locality Issues in Studying Parallel Systems\*

*Anand Sivasubramaniam  
Aman Singla  
Umakishore Ramachandran  
H. Venkateswaran*

Technical Report GIT-CC-93/63  
October 1993

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

## **Abstract**

We define a set of overhead functions that capture the salient artifacts representing the interaction between parallel application characteristics and architectural features. An execution-driven simulation testbed is used to separate these overheads in a parallel system. Using this testbed and a set of applications, we address two important issues. The first concerns the use of machine abstractions for performance studies of parallel systems. The second deals with quantifying the impact of locality on the performance of applications. The key conclusions from this study are that the newly proposed model LogP is an effective one for abstracting the network, and that ignoring locality can significantly affect the application performance.

**Key words:** Parallel Systems, Machine Abstractions, Locality, Execution-driven Simulation, Application-driven Studies

---

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

# 1 Introduction

Our study attempts to relate the application characteristics with the architectural features by identifying, isolating, and quantifying the different overheads that account for deviation from ideal behavior in a parallel system.<sup>1</sup> Such an isolation has a wide applicability: predicting the scalability of parallel systems by identifying performance bottlenecks; validating performance prediction models as well as models of computation used for algorithm development; and identifying key system artifacts that need to be incorporated in any model. Such an isolation of overheads has not been attempted before to the best of our knowledge. We have used an execution-driven simulation approach in our study to separate the overheads in a parallel system.

The key contributions of our work can be summarized as follows: We define the overhead functions and develop a framework for the separation of these functions. Using this framework we explore two important questions: (a) can machine abstractions be used in an execution-driven simulator for the performance studies of parallel systems?, and (b) can locality be ignored in the performance prediction or algorithm development without significantly affecting the validity of the results? We show that LogP [12] is an effective model for abstracting the network, and that ignoring locality can significantly affect the application performance.

Section 2 defines the overhead functions; Section 3 describes the simulation framework for measuring the overheads; Section 4 gives the motivation for addressing the above two questions and discusses the use of the framework for exploring them. We use a set of applications (Section 5) and a set of architectures (Section 6) as the basis to address these questions. Performance results are presented in Section 7 and we conclude with a discussion of the implications of the results in Section 8.

## 2 Overhead Functions

It is desirable to see a performance improvement (speedup) that is linear with the increase in the number of processors (as shown by the curve for linear behavior in Figure 1). With increasing number of processors, overheads in the parallel system increase (as shown by the curve for real execution in Figure 1) causing deviation from linear behavior. The overheads may even dominate the added computing power after a certain stage resulting in potential slow-downs. Parallel system overheads may be broadly classified into a purely algorithmic component (*algorithmic overhead*), and a component arising due to the interaction of the algorithm and the architecture (*interaction overhead*). The algorithmic overhead is due to the inherent *serial* part [6] and the *work-imbalance* in the algorithm, and is independent of the architectural

---

<sup>1</sup>The term, parallel system, is used to denote an algorithm-architecture combination.

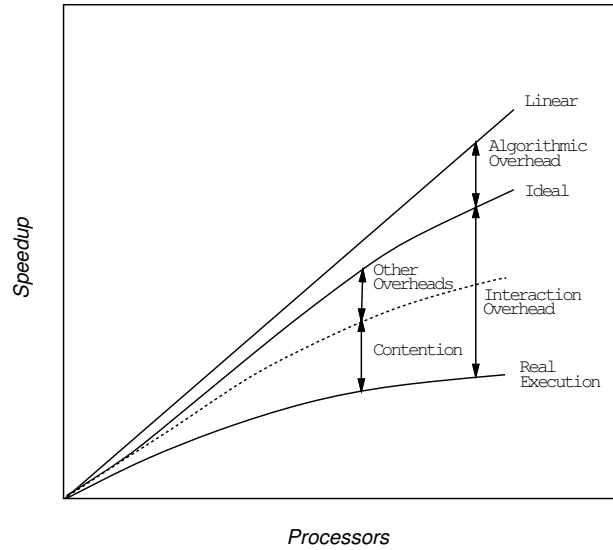


Figure 1: Overheads in a Parallel System

characteristics. Work-imbalance could result if in a parallel phase, there is a differential amount of work done in the different threads. 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 PRAM (Parallel Random Access Machine) [27]. Such a machine idealizes the parallel architecture by assuming an infinite number of processors, and unit costs for communication and synchronization. A real execution could deviate significantly from the ideal execution due to the overheads such as latency, contention, synchronization, scheduling and cache effects. These overheads are lumped together as the interaction overhead. As we observed in Section 1, isolating the influence of each component of the interaction overhead on the overall performance has wide applicability. 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 between the ideal curve and the real execution curve (see Figure 1).

We associate an *overhead function* with each such overhead that results in non-ideal behavior of the system. 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 [27] 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 algorithmic overheads are not germane to the questions

raised in Section 1, and hence we do not address it in the rest of the paper.

As we mentioned earlier, the interaction overhead should be separated into its component parts. We currently do not address scheduling overheads.<sup>2</sup> Processes in a parallel program often need to communicate during execution. This communication is explicit in message-passing systems via messages, while it is implicit on a shared memory system via accesses to shared variables. But regardless of the programming paradigm, communication involves network accesses and the physical limitations of the network tend to contribute to the overheads in the execution. These overheads may be broadly classified as latency and contention, and we associate an overhead function with each. The *Latency Overhead Function* ( $f_L(p)$ ) is thus defined as the total amount of time spent by a processor waiting for messages due to the transmission time on the links and the switching overhead in the network assuming that the messages did not have to contend for any link. Likewise, the *Contention Overhead Function* ( $f_C(p)$ ) is the total amount of time incurred by a processor due to the time spent waiting for links to become free by the messages. Synchronization and communication are intertwined in a message-passing system. On the other hand, in a shared memory system it is interesting to separate these two artifacts. Such systems normally provide some synchronization support which may either be as simple as an atomic read-modify-write operation, or may provide special hardware for more complicated operations like barriers and queue-based locks. While the latter may save execution time for complicated synchronization operations, the former is more flexible for implementing a variety of such operations. For reasons of generality, we assume that only the test&set operation is supported by shared memory systems. We also assume that the memory module (at which the operation is performed), is intelligent enough to perform the necessary operation in unit time. With such an assumption, the only network overhead due to the synchronization operation (test&set) is a roundtrip message, and the overheads for such a message are accounted for in the latency and contention overhead functions described earlier. The waiting time in a processor during synchronization operations is accounted for in the CPU time which would manifest itself as an algorithmic (serial or work imbalance) overhead. Hence, for the rest of this paper, we confine ourselves to the the only two aspects of the interaction overhead that are germane to this study, namely, latency and contention.

### 3 The Framework

We have taken an application-driven approach to conduct our study. A similar approach has also been used by other researchers in studying the impact of application characteristics on architectural requirements

---

<sup>2</sup>We do not distinguish between the terms, *process*, *processor* and *thread*, and use them synonymously in this paper.

[14, 22].

### 3.1 Approaches for Measuring Overheads

Experimentation, simulation, and analytical models are techniques that can be used for measuring overheads. But each has its own limitations. Experimentation is useful for understanding and evaluating existing architectures but the underlying hardware is fixed making it impossible to study the effect of changing individual architectural parameters; also it is difficult if not impossible to separate the effects of different architectural artifacts on the performance since we are constrained by the 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 [18, 26, 13] are mainly applicable to algorithms with regular communication structures that can be predetermined before the execution of the algorithm. Madala and Sinclair [18] confine their studies to synchronous algorithms while Vrsalovic et al. [26] and Cvetanovic [13] develop models for regular iterative algorithms. However, there exist several applications [22] 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 [19] ignores data inconsistency that can arise in a cache-based multiprocessor during the course of execution of an application.

In this study we use execution-driven simulation for separating and quantifying the overhead functions. Several recent studies have also stressed the importance of using simulation with realistic workloads as the appropriate method of studying parallel systems [22]. However, simulation also has its limitations. For example, it may not always be possible to simulate large systems owing to resource (time and space) constraints in using architectural simulators. But simulation can be viewed as complementing the analytical models in that simulation can be used to obtain several datapoints for a parallel system, which can then be used as a feedback to refine the existing models. Further, simulation can also be used to validate existing

models using real applications.

## 3.2 SPASM

SPASM (Simulator for Parallel Architectural Scalability Measurements) is an execution-driven simulator written in CSIM that we have developed for measuring the overhead functions. The reader is referred to [24] for a detailed description of the implementation of SPASM. It provides us with a wide range of input parameters and output statistics.

### 3.2.1 Parameters

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

### 3.2.2 Metrics

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

*Ideal time* is the total time taken by a parallel program to execute on an ideal machine such as the PRAM. It includes the algorithmic overhead but does not include the interaction overhead. SPASM simulates an ideal machine to provide this metric. As we mentioned in Section 2, the difference between the linear time and the ideal time gives the algorithmic overhead.

SPASM quantifies both the latency overhead function ( $f_L(p)$ ) as well as the contention overhead function ( $f_C(p)$ ) seen by a processor as described in Section 2. This is done by time-stamping messages when they are sent. At the time a message is received, the time that the message would have taken in a contention free environment is charged to the latency overhead function while the rest of the time is accounted for in the contention overhead function. Though not relevant to this study, it is worthwhile to mention that SPASM provides the latency and contention incurred by a message as well as the latency and contention that a processor may choose to see. Even though a message may incur a certain latency and contention, a processor may choose to hide all or part of it by overlapping computation with communication. Such a

scenario may arise with a non-blocking message operation on a message-passing machine or with a prefetch operation on a shared memory machine. But for the rest of this paper (since we deal with blocking load/store shared memory operations), we assume that a processor sees all of the network latency and contention.

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

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

- **NORMAL:** A program is in the NORMAL mode if it is not in any of the other modes. An application programmer may further define sub-modes if necessary.
- **BARRIER:** Phase corresponding to a barrier synchronization operation.
- **MUTEX:** Even though the simulated hardware provides only a test&set operation, mutual exclusion *lock* (implemented using test-test&set [7]) is available as a library function in SPASM. A program enters this mode during lock operations. With this mechanism, we can separate the overheads due to the synchronization operations from the rest of the program execution.
- **PGM\_SYNC :** Parallel programs may use Signal-Wait semantics for pairwise synchronization. A lock is unnecessary for the Signal variable since only 1 processor writes into it and the other reads from it. This mode is used to differentiate such accesses from normal load/store accesses.

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

## **4 Using the Framework**

There are several applications for the framework described in this paper. In [24], we use it to study the scalability of parallel systems. In this paper, we use the framework to explore the following two questions:

- Can machine abstractions be used in an execution-driven simulator for the performance studies of parallel systems?

Performance analysis of parallel systems is complex due to the several degrees of freedom that exist in them. Similarly, developing algorithms for parallel architectures is also hard if one has to grapple with all these aspects. There have been several attempts to abstract artifacts of a parallel machine both from the point of view of performance analysis as well as program development. For instance, some simulators [9, 11, 21] (including ours) have abstracted out the instruction-set of the processors since a detailed simulation of the instruction-set is not likely to contribute significantly to the performance analysis of parallel systems. Similarly, Agarwal [1] develops mathematical models for abstracting the network and studying network properties. Theoretical models such as the PRAM [15], and LogP [12] are abstractions of parallel machines used for algorithm development and analysis.

There is a growing awareness for studying parallel systems with real applications due to the dynamic nature of the interaction between applications and architectures. Execution-driven simulation is an important technique for enabling such studies. However, simulating every artifact of a parallel machine can considerably slow down the speed of simulation. In fact, as we observed in Section 3, it may be difficult to simulate large systems due to resource constraints. Therefore, it is worth investigating this question of using machine abstractions for speeding up execution-driven simulations in the context of real applications. In particular, since the interconnection network is the single-most important component of a parallel architecture, we investigate if it can be abstracted without sacrificing the accuracy of the performance analysis. The impact of network abstraction on performance analysis in the context of real applications has not been adequately addressed in earlier studies.

- Can locality be ignored in the performance prediction or algorithm development without significantly affecting the validity of the results?

Locality (both spatial and temporal) is an application characteristic and its importance in application design is well-recognized. Parallel machines facilitate exploiting locality for performance reasons in various ways: private caches in shared memory multiprocessors, and local memories in distributed memory machines. Application design has to ensure that the hardware facility is appropriately used to exploit the locality in the application to enhance performance. The performance impact of locality is determined by the interaction between the application characteristic and the hardware facility. However there have not been many attempts in quantifying this impact on application performance. Architectural studies [5, 2] have explored hardware facilities that would help exploit locality in applications. Gupta, et al. [22] have investigated the appropriate memory and cache sizes in a

parallel system based on the characteristics (such as the working set) of a set of applications. Many applications (like CG and CHOLESKY described in the next section) exhibit dynamic behavior, both in the degree of sharing as well as in their communication requirements. Such a dynamic behavior coupled with a complex interaction with the underlying hardware makes it difficult to capture the effect of locality by a static analysis of the application and/or the architecture.

It is clear that it is important to quantify the impact of both network abstraction and locality on the performance of real applications to address the above two questions. To study the first question it is sufficient if we simulate the details of a parallel system with an appropriate abstraction for the network. Abstracting machine characteristics via a few simple parameters have been traditionally addressed by theoretical models of computation. The PRAM model assumes conflict-free accesses to shared memory (assigning unit cost for memory accesses) and zero cost for synchronization. The PRAM model has been augmented with additional parameters to account for memory access latency [3], memory access conflicts [4], and cost of synchronization [16, 10]. Valiant's Bulk Synchronous Parallel (BSP) model [25] is a departure from the PRAM models, and is one of the first attempt to realistically bridge the gap between theory and practice. The *LogP* model recently developed by Culler et al. [12], inspired by BSP, improves on the BSP model and is considered a more realistic one since it incorporates the two important properties of any network, namely, latency and contention. Therefore, we use this model in our work to abstract the network.

The LogP model assumes a collection of processing nodes executing asynchronously, communicating with each other by small fixed-size messages incurring constant latencies on a network with a finite bandwidth. The model defines the following set of parameters that are independent of network topology:

- $L$ : the *latency*, is the maximum time spent in the network by a message from a source to any destination.
- $o$ : the *overhead*, is the time spent by a processor in the transmission/reception of a message.
- $g$ : the *communication gap*, is the minimum time interval between consecutive message transmissions/receptions from/to a given processor.
- $P$ : is the number of processors in the system.

The  $L$ -parameter captures the actual network transmission time for a message in the absence of any contention, while the  $g$ -parameter corresponds to the available per-processor bandwidth. By ensuring that a processor does not exceed the per-processor bandwidth of the network (by maintaining a gap of at least  $g$

between consecutive transmissions/receptions), a message is guaranteed to encounter no contention in the network.

We use the  $L$  and  $g$  parameters of the model to abstract the network in the simulator. Since we are considering a shared memory platform (where the ‘message overhead’ is incurred in the hardware) the contribution of the  $o$ -parameter is insignificant compared to  $L$  and  $g$ . Therefore we do not discuss it in the rest of this paper.

In relation to the locality question, we confine ourselves to shared memory style parallel programs. Locality in such programs manifests itself implicitly through the memory access pattern for shared data. Most modern shared memory multiprocessors have sufficiently large-size caches, capable of holding the working set of most parallel applications [22]. Therefore, we use a LogP machine augmented with a cache, referred to as a *cLogP* machine, in our simulator to answer the two questions. Shared memory machines with private caches usually employ a protocol to maintain cache coherence. With a diverse range of cache coherence protocols, it would become very specific if cLogP were to simulate any particular protocol. Moreover, the locality characteristics of an application does not depend on any specific protocol. Hence, in our simulation we assume that cLogP only captures the locality aspect of the caches by taking the appropriate coherence actions and not accounting for the overhead involved in maintaining the coherence.

The approach to answering the first question is as follows: we simulate a real parallel system exactly in the context of an application; we then simulate a cLogP abstraction of the same parallel system. If the results of the two simulations agree then we have answered the first question affirmatively, and also have shown that LogP is a good abstraction for the network. The approach we take to answer the second question is as follows: we simulate LogP and cLogP characterizations of a parallel system in the context of real applications. The difference in results between the two simulations quantifies the impact of locality on performance. The applications and the architectures that are used to answer these two questions are described in the next two sections.

## 5 Application Characteristics

This section briefly describes the characteristics of five applications used in this study. Three of them (EP, IS and CG) are from the NAS parallel benchmark suite [8]; CHOLESKY is from the SPLASH benchmark suite [23]; and FFT is the well-known Fast Fourier Transform algorithm. The characteristics include the data access pattern, the synchronization pattern, the communication pattern, the computation granularity (the amount of work done) and data granularity (the amount of data communicated) for each phase of the

program. EP and FFT are well-structured applications with regular communication patterns determinable at compile-time, with the difference that EP has a higher computation to communication ratio. IS also has a regular communication pattern, but in addition it uses locks for mutual exclusion during the execution. CG and CHOLESKY are different from the other applications in that their communication patterns are not regular (both use sparse matrices) and cannot be determined at compile time. While a certain number of rows of the matrix in CG is assigned to a processor at compile time (static scheduling), CHOLESKY uses a dynamically maintained queue of runnable tasks.

## EP

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

Table 1: Characteristics of EP

EP is the “Embarrassingly Parallel” application 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. It is computation bound and has little communication between processors. A large number of floating point random numbers is generated which are then subject to a series of operations. The computation granularity of this section of the code is considerably large and is linear in the number of random numbers (the problem size) calculated. A data size of 64K pairs of random numbers has been chosen in this study. The operation performed on a computed random number is completely independent of the other random numbers. The processor assigned to a random number can thus execute all the operations for that number without any external data. Hence the data granularity is meaningless for this phase of the program. Towards the end of this phase, a few global sums are calculated by using a logarithmic reduce operation. In step  $i$  of the reduction, a processor receives an integer from another which is a distance  $2^i$  away and performs an addition of the received value with a local value. The data that it receives (data granularity) resides in a cache block in the other processor, along with the synchronization variable which indicates that the data is ready (synchronization is combined with data transfer to exploit spatial locality). Since only 1 processor writes into this variable, and the other spins on the value of the synchronization variable (the PGM\_SYNC mode described in Section 3.2), no locks are used. Every processor reads the global sum from the cache block of processor 0 when the last addition is complete. The computation granularity between these communication steps can lead to work imbalance since the number of participating processors halves

after each step of the logarithmic reduction. However since the computation is a simple addition it does not cause any significant imbalance for this application. The amount of local computation in the initial computation phase overshadows the communication performed by a processor. Table 1 summarizes the characteristics of EP.

## IS

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

Table 2: Characteristics of IS

IS is the “Integer Sort” application that uses bucket sort to rank a list of integers which is an important operation in “particle method” codes. A list of 64K integers with 2K buckets is chosen for this study. An implementation of the algorithm is described in [20] and Table 2 summarizes its characteristics. The input list is equally partitioned among the processors. Each processor maintains two sets of buckets. One set of buckets (of size 2K) is used to maintain the information for the portion of the list local to it. The other set (of size  $chunk = 2K/p$  where  $p$  is the number of processors) maintains the information for the entire list. A processor first updates the local buckets for the portion of the list allotted to it, which is an entirely local operation (phase 1). Each list element would require an update (integer addition) of its corresponding bucket. A barrier is used to ensure the completion of this phase. The implementation of the barrier is similar to the implementation of the logarithmic global sum operation discussed in EP, except that no computation need be performed. A processor then uses the local buckets of every other processor to calculate the bucket values for the  $chunk$  of the global buckets allotted to it (phase 3). The phase would thus require  $chunk * (p - 1)$  remote bucket values per processor. During this calculation, the processor also maintains the sum of all the global bucket values in its  $chunk$ . These sums are then involved in a logarithmic reduce operation (phase 4) to obtain the partial sum for each processor. Each processor uses this partial sum in calculating the partial sums for the  $chunk$  of global buckets allotted to it (phase 5) which is again a local operation. At the completion of this phase, a processor sets a lock (test-test&set lock [7])

for each global bucket, subtracts the value found in the corresponding local bucket, updates the local bucket with this new value in the global bucket, and unlocks the bucket (phase 7). The memory allocation for the global buckets and its locks is done in such a way that a bucket and its corresponding lock fall in the same cache block and the rest of the cache block is unused. Synchronization is thus combined with data transfer and false sharing is avoided. The final list ranking phase (phase 8) is a completely local operation using the local buckets in each processor and is similar to phase 1 in its characteristics.

## FFT

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

Table 3: Characteristics of FFT

FFT is the one dimensional complex Fast Fourier Transform of  $N$  (64K for this study) points.  $N$  is a power of 2 and greater than or equal to the square of the number of processors  $P$ . The application is implemented in 3 main phases. In phases 1 and 5, processors perform the radix-2 butterfly computation on  $N/P$  local points. Phase 3 is the only communication phase in which the *cyclic* layout of data is changed to a *blocked* layout as described in [12]. It involves an all-to-all communication step where each processor distributes its local data equally among the  $P$  processors. The communication in this step is *staggered* with processor  $i$  starting with reading data ( $\frac{N}{P^2}$  points) from processor  $i + 1$  and ending with reading data from processor  $i - 1$  in  $P - 1$  substeps. This communication schedule minimizes contention both in the network and at the processor ends. These three phases are separated by barriers.

## CG

CG is the ‘‘Conjugate Gradient’’ application which uses the Conjugate Gradient method to estimate the smallest eigenvalue of a symmetric positive-definite sparse matrix with a random pattern of non-zeroes that is typical of unstructured grid computations. A sparse matrix of size 1400X1400 containing 100,300 non-zeroes has been used in the study. The sparse matrix and the vectors are partitioned by rows assigning an equal number of contiguous rows to each processor (static scheduling). We present the results for five iterations of the Conjugate Gradient Method in trying to approximate the solution of a system of linear

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

Table 4: Characteristics of CG

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

## CHOLESKY

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

Table 5: Characteristics of CHOLESKY

This application performs a Cholesky factorization of a sparse positive definite matrix. The sparse nature of the input matrix results in an algorithm with a data dependent dynamic access pattern. The algorithm requires an initial symbolic factorization of the input matrix which is done sequentially because it requires only a small fraction of the total compute time. Only numerical factorization [23] is parallelized and analyzed. Sets of columns having similar non-zero structure are combined into supernodes at the end of symbolic factorization. Processors get tasks from a central task queue. Each supernode is a potential task which is used to modify subsequent supernodes. A *modifications\_due* counter is maintained with each supernode. Thus each task involves fetching the associated supernode, modifying it and using it to modify other supernodes, thereby decreasing the *modifications\_due* counters of supernodes. Communication is involved in fetching all the required columns to the processor working on a given task. When the counter for a supernode reaches 0 it is added to the task queue. Synchronization occurs in locking the task queue when fetching or adding tasks, and locking columns when they are being modified. A 1806-by-1806 matrix with 30,824 floating point non-zeros in the matrix and 110,461 in the factor with 503 distinct supernodes is used for the study.

## 6 Architectural Characteristics

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

We use three shared memory platforms with different interconnection topologies: the *fully connected network*, the *binary hypercube* and the *2-D mesh*. All three networks use serial (1-bit wide) unidirectional links with a link bandwidth of 20 MBytes/sec. The fully connected network models two links (one in each direction) between every pair of processors in the system. The cube platform connects the processors in a binary hypercube topology. Each edge of the cube has a link in each direction. The 2-D mesh resembles the Intel Touchstone Delta system. Links in the North, South, East and West directions, enable a processor in the middle of the mesh to communicate with its four immediate neighbors. Processors at corners and along an edge have only two and three neighbors respectively. Equal number of rows and columns is assumed when the number of processors is an even power of 2. Otherwise, the number of columns is twice the number of rows (we restrict the number of processors to a power of 2 in this study). Messages are circuit-switched and

use a transmission scheme similar to the one used on the Intel iPSC/860 [17]. A circuit is set up between the source and the destination, and the message is then sent in a single packet. Message-sizes can vary upto 32 bytes. We assume that the switching time for setting up a circuit (in a contention free scenario) is negligible.

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

As outlined in section 4, to answer the two questions raised there we need a LogP and cLogP versions of the above three architectural platforms. In the LogP model, a processor makes a network access for each remote load/store operation, in effect not utilizing its private cache. Thus the LogP characterization is equivalent to an ordinary NUMA machine such as the BBN Butterfly GP1000. The  $L$  parameter for a message is chosen to be 1.6 microseconds assuming 32-byte messages. Similar to the method used in [12], the  $g$  parameter is calculated using the cross-section bandwidth available per processor for each of the above network configurations. The resulting  $g$  parameters for the full, cube and mesh networks are respectively,  $3.2/p$ , 1.6 and  $0.8 * p_x$  microseconds (where  $p$  is the number of processors and  $p_x$  is the number of columns in the mesh). In the cLogP model, a load/store operation would need a network access only when it cannot be satisfied by the private cache or the processor's piece of the shared memory. As observed in Section 4, a large enough private cache is a good abstraction for locality in a simulation environment. Further, with a large enough cache the details of the physical organization of the cache may not be important for the purposes of quantifying the effect of locality. Therefore, we use a 2-way set-associative cache (64KBytes) in keeping with the organization used in most modern day shared memory multiprocessors. As we mentioned earlier (see Section 4), we take the appropriate cache coherence actions without accounting for the their overhead.

## 7 Performance Results

The simulation results of the five parallel applications on the actual machine, and the LogP and cLogP models of the actual machine are discussed in this section. The results presented include the execution times, latency overheads, and the contention overheads for each execution mode of the applications. We confine our discussion to the specific results that are relevant to the questions raised in Section 4.

## 7.1 Importance of Locality

As we observed in Section 5, EP, FFT, and IS are applications with statically determinable memory reference patterns. Thus, in implementing these applications we ensured that the amount of communication (due to the lack of locality in these applications) is minimized. On the other hand, CG and CHOLESKY preclude any such optimization owing to their dynamic memory reference patterns. The number of messages generated on the network due to lack of locality in an application is the same regardless of the network topology. Even though the number of messages stays the same, the contention is expected to increase when the connectivity in the network decreases. Therefore, the impact of locality is expected to be more for a cube network compared to a full; and for a mesh network compared to a cube.

To address the locality question, we consider the results for the LogP and the cLogP models. While all three applications are static, EP has the highest computation to communication ratio, followed by FFT, and IS. For EP regardless of the network topology there is agreement in the results for LogP, cLogP, and the actual machine. On a fully connected and cube networks there is little difference in the results for FFT as well, whereas for the mesh interconnect there is a difference between LogP and cLogP (Figure 3). The difference is due to the fact that FFT has more communication (due to non-local references) compared to EP, and with increased  $g$ -parameter (which models contention) the effect of non-local references is amplified. For IS (see Figure 4), which has even more communication than FFT, there is a more pronounced difference between LogP and cLogP for all three networks.

Figures 5 and 6 show the total execution times for CG and CHOLESKY on a fully connected network. The difference in the results between LogP and cLogP curves in each of these figures quantifies the impact of locality for each of these applications. As can be observed, the impact for these two applications is much more pronounced than for the static ones since the LogP implementation cannot be optimized to exploit locality. We observe that the cLogP curve is very close to the actual execution curve for all applications and on all networks. Further, the LogP execution curves for the CG and CHOLESKY on both the cube and the mesh (see Figures 7, 8, 9, and 10) do not even follow the shape of the cLogP execution. This significant deviation of LogP from cLogP execution is due to the amplified effect that the large amount of communication in these applications has with increased contention in lower connectivity networks (see Figures 25, 26, 27, and 28).

Isolating the contention and latency overheads in the context of the execution modes identified in Section 3.2 enables discovering and quantifying locality effects. The Figures 11, 12, and 13, illustrate some of these effects for FFT, CG, and EP respectively. In FFT, during the communication phase a processor reads 4

consecutive data items displaying spatial locality. In either a cLogP or the actual execution, a cache-miss on the first data item brings in the whole cache block (which is 4 data items). On the other hand, in a LogP execution all four data items result in network accesses. Thus the LogP execution for FFT incurs a latency (Figure 11) that is approximately four times that of the other two. Similarly, ignoring spatial and temporal locality in CG (Figure 12) results in a significant disparity for the latency overhead in the LogP execution compared to the other two.

Ignoring locality can manifest itself as other subtle effects, which may affect performance depending on application characteristics. In EP, there is a PGM\_SYNC mode (see Section 3.2) where a processor waits on a condition variable. In a cLogP execution of EP, only the first and last accesses to the condition variable use the network while a LogP execution would incur a network access for each reference to the condition variable as is illustrated in Figure 13. Similarly, a MUTEX mode in an application which uses the test-test&set primitive [7], would behave like an ordinary test&set operation in the LogP execution thus resulting in an increase of network accesses. As can be seen in Figure 2, these effects do not impact the total execution time of EP since computation dominates for this particular application.

## 7.2 Abstracting the Network

It is sufficient to consider the cLogP and the actual machine executions for answering the question related to network abstractions. From Figures 11, 12, 13, 14, and 15, we observe that the latency overhead curves for the cLogP execution display a trend very similar to the actual execution thus validating the use of the  $L$ -parameter of the LogP model for abstracting the network latency. Since LogP model abstracts the network latency independent of the topology, the other two network platforms (cube and mesh) also display a similar agreement between cLogP and actual executions. Despite exhibiting the same trend, there is a difference in the absolute values for the latency overhead between cLogP and the actual executions. cLogP models  $L$  as the time taken for a cache-block (32 bytes) transfer. But some messages may actually be shorter (for instance memory requests), thus making  $L$  pessimistic with respect to the actual execution. On the other hand, cLogP does not model coherence traffic thereby incurring fewer network messages than the actual execution, which can have the effect of making  $L$  more optimistic than the actual. The impact of these two counter-acting effects on the overall performance depends on the application characteristics. The pessimism is responsible for cLogP displaying a higher latency overhead than the actual for FFT (Figure 11) and CG (Figure 12) since there is very little coherence related activity in these two applications; while the optimism favors cLogP in IS (Figure 14) and CHOLESKY (Figure 15) where coherence related activity is dominant.

From the Figures 16, 17, 18, 19, and 20, we observe that the contention overhead curves for the cLogP execution display a trend very similar to the actual execution. The difference that can be observed in the absolute values can be explained as follows. The  $g$ -parameter in cLogP is estimated using the bisection bandwidth of the network as suggested in [12], leading to a pessimistic estimate because of not accounting for communication locality. This pessimism increases as the diameter of the network increases as can be seen in Figures 16, 17, and 18. This pessimism is amplified further for applications such as EP that display a significant amount of communication locality as can be seen in Figures 21, 22, and 23. In fact, this amplified effect changes the very trend of the cLogP contention curves compared to the actual. It is worth mentioning that for a non-square mesh estimating the  $g$ -parameter using the bisection bandwidth makes it more pessimistic. This is illustrated by the jagged contention curve for the cLogP execution of FFT (Figure 24). Despite such differences in the contention overhead predicted by cLogP and the actual executions, the concept of abstracting the network contention using the  $g$ -parameter of LogP is valid. This is because the impact of this difference on the overall execution times may not be significant. Further, it may be possible to estimate the  $g$ -parameter taking into account the communication properties of the application.

## 8 Discussion

We raised two questions in this paper to be addressed through our framework. The results presented in the previous section show that the network can be abstracted out of an execution-driven simulator, and that locality plays a significant role in the performance of parallel systems to be ignored in any machine model. There are several observations from our study that are summarized below:

- *Abstracting the Network:* The overhead functions have helped us validate the use of the  $L$  and  $g$  parameters of the LogP model for abstracting the network. In most cases, the overheads due to the  $L$  and  $g$  factors in the cLogP model closely resemble the corresponding latency and contention overhead functions on the actual machine. Such an abstraction of the network can be incorporated into a simulation platform to enhance the speed of simulation. This fact is confirmed by our observation that the elapsed time for simulating cLogP execution in SPASM is significantly shorter than for simulating the actual execution. As we observed in the previous section, estimation of the  $g$ -parameter can be pessimistic if the network contention is treated as a pure architectural artifact. The communication properties of the application should be taken into consideration as well.

- *Abstracting Locality*: Locality is an important factor in determining the performance of parallel programs. As we observed earlier (see Section 4), a private cache is the hardware feature that facilitates exploiting locality in shared memory multiprocessors. A machine model such as LogP that does not account for a cache is therefore expected to be pessimistic for predicting the performance of shared memory style parallel programs. A theoretical model serves two purposes: (a) to aid performance-conscious program development; and (b) performance prediction of programs written for the model. If LogP is used to develop shared memory parallel programs then its performance as predicted by LogP can be significantly different from the actual as was observed in the previous section for several applications. One might expect that the LogP model augmented with a cache may serve as a good performance predictor. In fact, we use such a model (cLogP) to answer the locality question in this study. Our performance results show that the cLogP predictions are close to the actual executions. Though these results were obtained using simulation, it is reasonable to hypothesize that cLogP would serve as an accurate performance predictor for shared memory applications whose memory reference patterns are statically determinable, just as LogP is for message passing style applications.

However, it may not be easy to analytically predict performance using theoretical models since locality in a parallel computation is much more complex due to the additional degrees of freedom compared to a sequential computation. Even for static applications, data alignment (several variables falling in the same cache block as observed in FFT) leading to true and false sharing, and scheduling are two factors that make abstracting locality complex. In dynamic applications, this problem is exacerbated owing to factors such as dynamic scheduling (both the overhead of scheduling as well as process migration as in CHOLESKY), and synchronization (implicit synchronization using condition variables and explicit synchronizers such as locks and barriers). While incorporating a cache in a model is worthwhile (as we have seen with cLogP), it may be difficult to use such a model analytically in either program development or performance prediction of shared memory applications. On the other hand, an execution-driven simulation platform which uses such a model has proven useful in this study to quantify the impact of locality.

- *The Power of Overhead Functions*: The overhead functions are the key to addressing both of the above issues. For instance, even when total execution time curves were similar, the latency and contention overhead curves helped us determine whether the model parameters were accurate in

capturing the intended machine abstractions. Thus we were able to show that the  $g$ -parameter is pessimistic for calculating the contention overhead for several applications, and that the  $L$ -parameter can be optimistic or pessimistic depending on the application characteristics. The overhead functions can also play an important role in identifying performance bottlenecks in an implementation. This observation can be illustrated with our experiences in implementing the IS application. An initial implementation of IS on the LogP model exhibited abnormal contention overheads (Figure 29). A re-examination of the implementation, suggested the skewing the memory accesses for the global buckets which reduced this contention considerably (Figure 16).

Our framework using the overhead functions has been useful in validating the LogP model. One can experimentally determine the accuracy of the performance predicted by the LogP model as is done in [12] using CM-5. However, this approach does not validate the individual parameters abstracted using the model. Using the overhead functions in the framework of an execution-driven allows validation of the individual parameters. simulator addresses both of these deficiencies. Further, we have shown that using these parameters for abstracting the network in an execution-driven simulator helps to identify performance bottlenecks and serves as a program restructuring guide. This also avoids a potential danger in using LogP alone as a performance conscious programming model due to the pessimism that may be there in the estimation of the  $L$  and  $g$  parameters.

## References

- [1] Anant Agarwal. Limits on Interconnection Network Performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–412, October 1991.
- [2] Anant Agarwal, David Chaiken, Kirk Johnson, David Kranz, John Kubiawicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, and Dan Nussbaum. The MIT Alewife machine : A large scale Distributed-Memory Multiprocessor. In *Scalable shared memory multiprocessors*. Kluwer Academic Publishers, 1991.
- [3] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. On Communication Latency in PRAM Computations. In *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 11–21, 1989.
- [4] H. Alt, T. Hagerup, K. Mehlhorn, and F. P. Preparata. Deterministic Simulation of Idealized Parallel Computers on More Realistic Ones. *SIAM Journal of Computing*, 16(5):808–835, 1987.

- [5] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera Computer System. In *Proceedings of the ACM 1990 International Conference on Supercomputing*, pages 1–6, Amsterdam, Netherlands, 1990.
- [6] 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.
- [7] 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.
- [8] D. Bailey et al. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications*, 5(3):63–73, 1991.
- [9] 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.
- [10] Richard Cole and Ofer Zajicek. The APRAM: Incorporating Asynchrony into the PRAM Model. In *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, 1989.
- [11] 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.
- [12] David Culler et al. LogP : Towards a realistic model of parallel computation. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, May 1993.
- [13] Zarka Cvetanovic. The effects of problem partitioning, allocation, and granularity on the performance of multiple-processor systems. *IEEE Transactions on Computer Systems*, 36(4):421–432, April 1987.
- [14] 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.

- [15] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10th Annual Symposium on Theory of Computing*, pages 114–118, 1978.
- [16] Phillip B. Gibbons. A More Practical PRAM Model. In *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 158–168, 1989.
- [17] Intel Corporation, Oregon. *Intel iPSC/2 and iPSC/860 User's Guide*, 1989.
- [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] Janak H. Patel. Analysis of multiprocessors with private cache memories. *IEEE Transactions on Computer Systems*, 31(4):296–304, April 1982.
- [20] U. Ramachandran, G. Shah, S. Ravikumar, and J. Muthukumarasamy. Scalability study of the KSR-1. In *Proceedings of the 1993 International Conference on Parallel Processing*, pages I–237–240, August 1993.
- [21] 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.
- [22] Edward Rothberg, Jaswinder Pal Singh, and Anoop Gupta. Working sets, cache sizes and node granularity issues for large-scale multiprocessors. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 14–25, May 1993.
- [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, Aman Singla, Umakishore Ramachandran, and H. Venkateswaran. A Simulation-based Scalability Study of Parallel Systems. Technical Report GIT-CC-93/27, College of Computing, Georgia Institute of Technology, April 1993.
- [25] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, August 1990.

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

# Execution Time Graphs

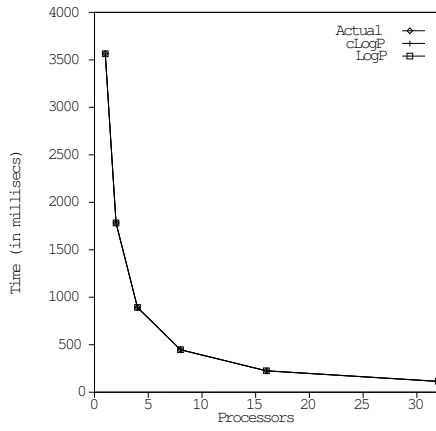


Figure 2: EP on Full

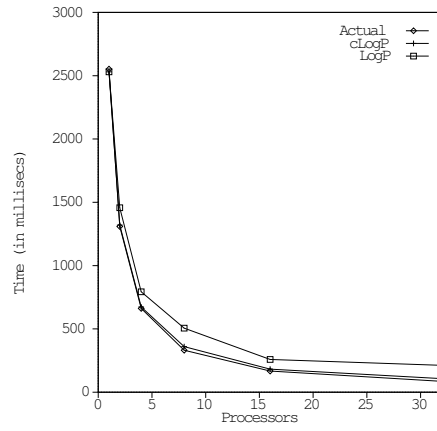


Figure 3: FFT on Mesh

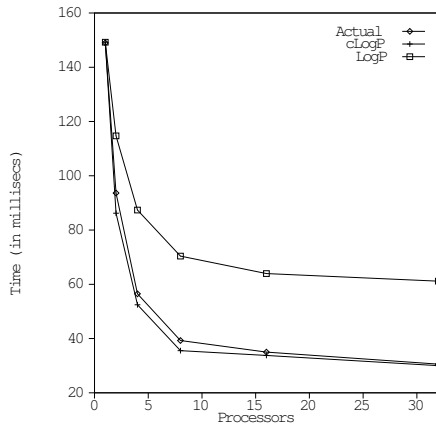


Figure 4: IS on Full

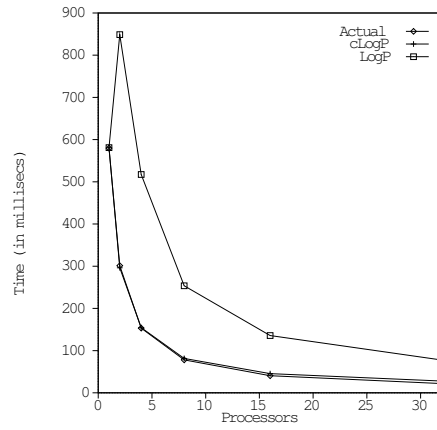


Figure 5: CG on Full

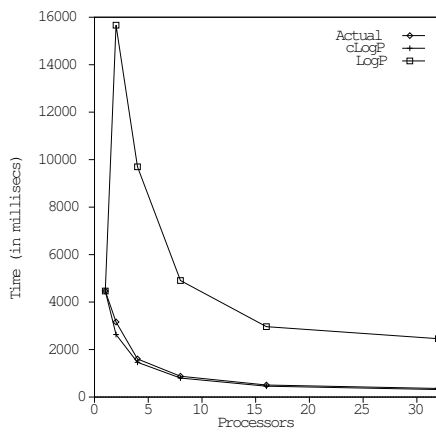


Figure 6: CHOLESKY on Full

# Execution Time Graphs (contd)

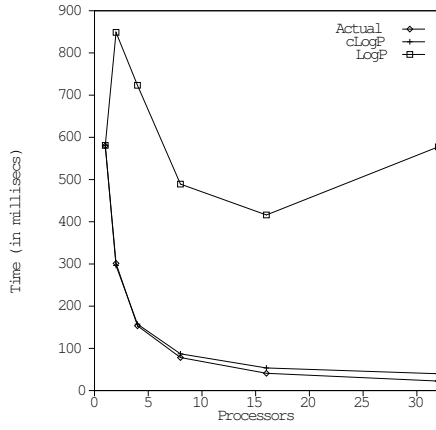


Figure 7: CG on Cube

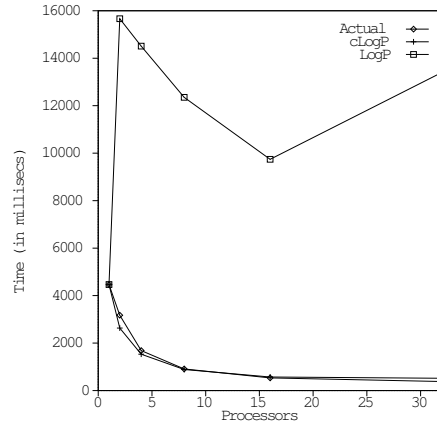


Figure 8: CHOLESKY on Cube

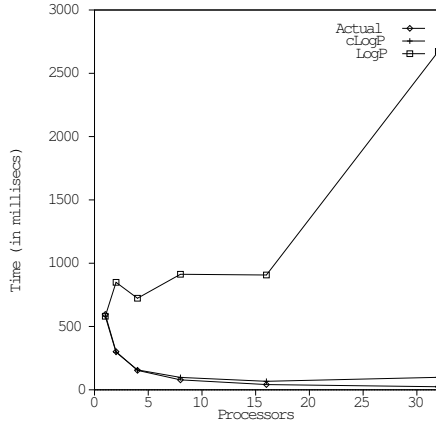


Figure 9: CG on Mesh

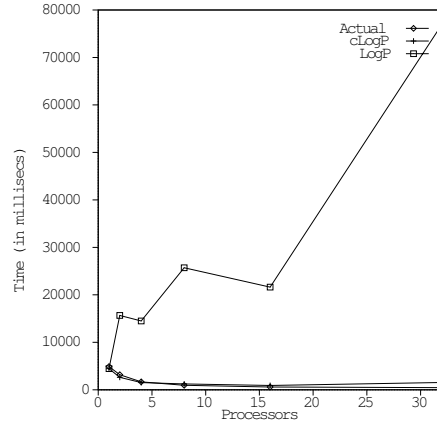


Figure 10: CHOLESKY on Mesh

# Latency Graphs

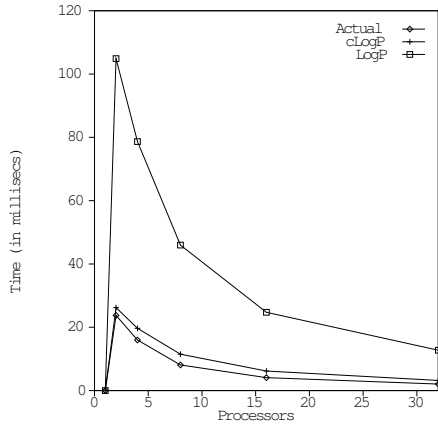


Figure 11: FFT on Full : Latency

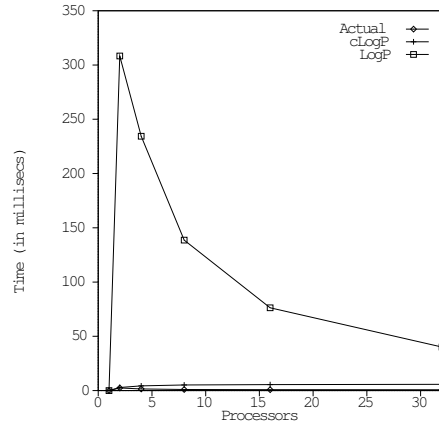


Figure 12: CG on Full : Latency

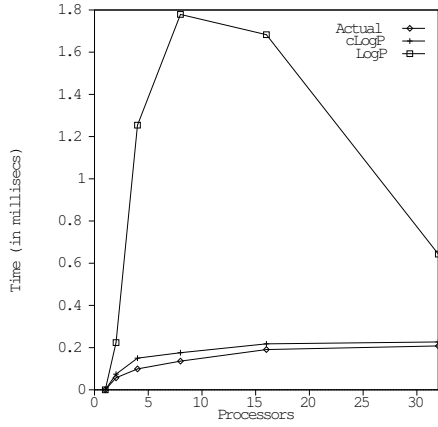


Figure 13: EP on Full : Latency

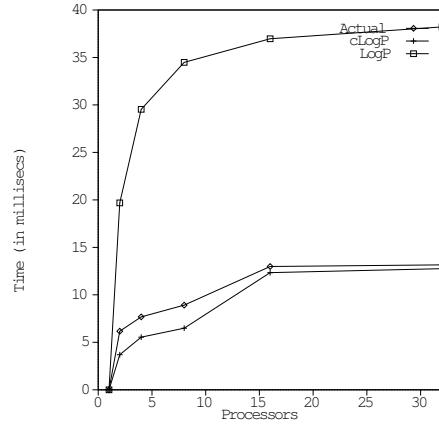


Figure 14: IS on Full : Latency

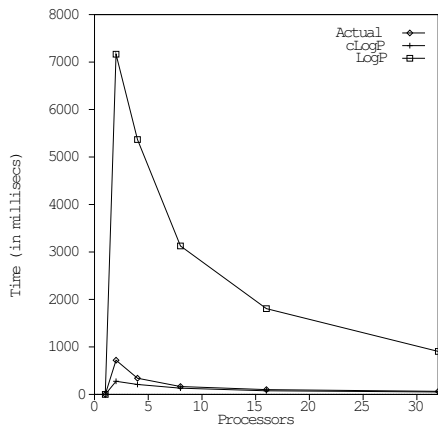


Figure 15: CHOLESKY on Full : Latency

# Contention Graphs

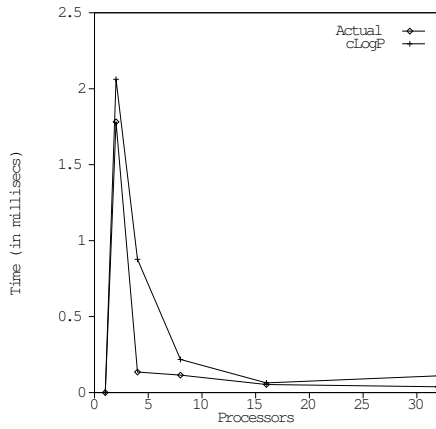


Figure 16: IS on Full : Contention

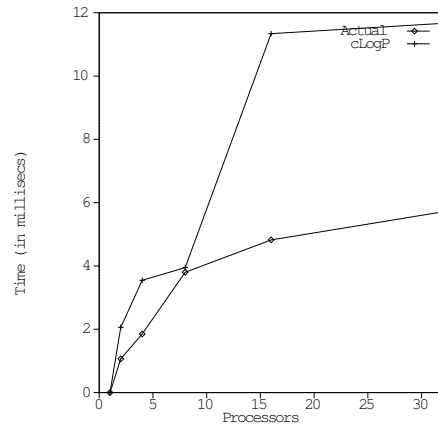


Figure 17: IS on Cube : Contention

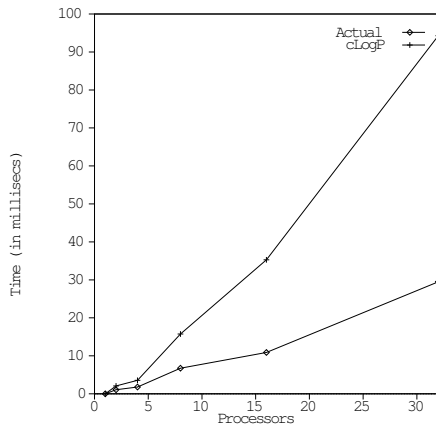


Figure 18: IS on Mesh : Contention

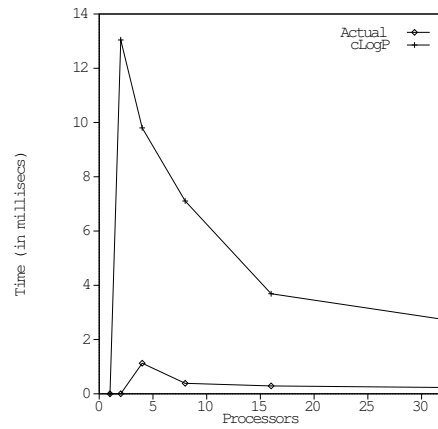


Figure 19: FFT on Cube : Contention

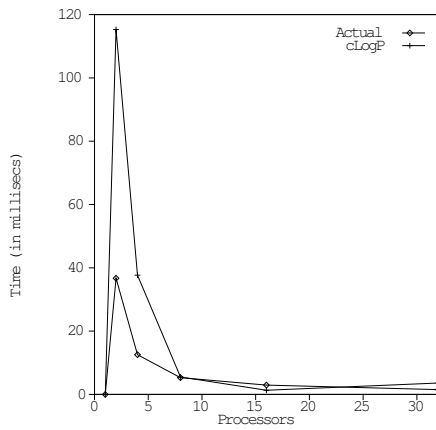


Figure 20: CHOLESKY on Full : Contention

# Contention Graphs (contd)

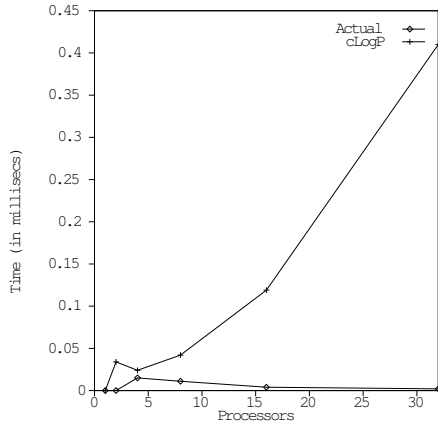


Figure 21: EP on Full : Contention

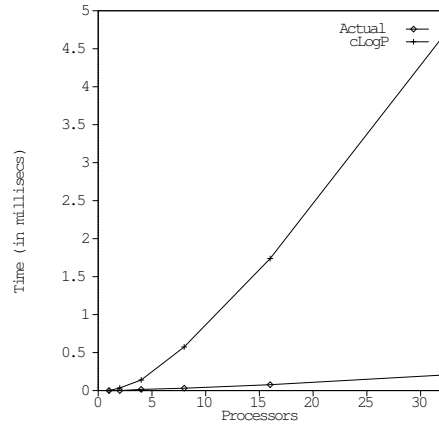


Figure 22: EP on Cube : Contention

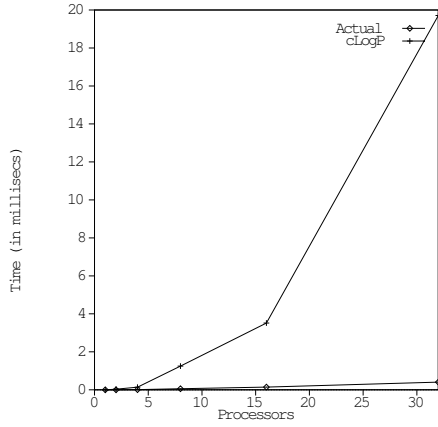


Figure 23: EP on Mesh : Contention

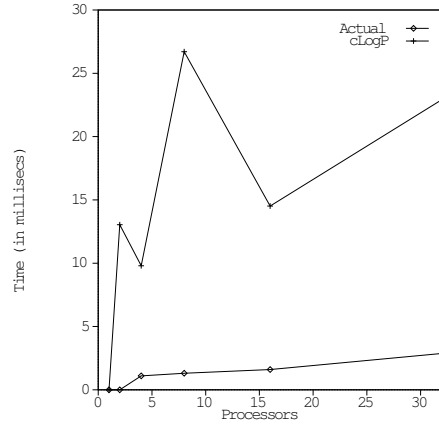


Figure 24: FFT on Mesh : Contention

# Contention Graphs (contd)

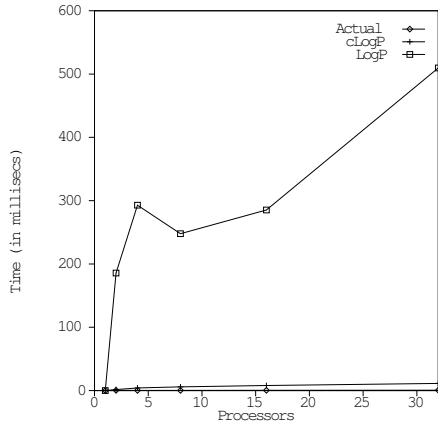


Figure 25: CG on Cube : Contention

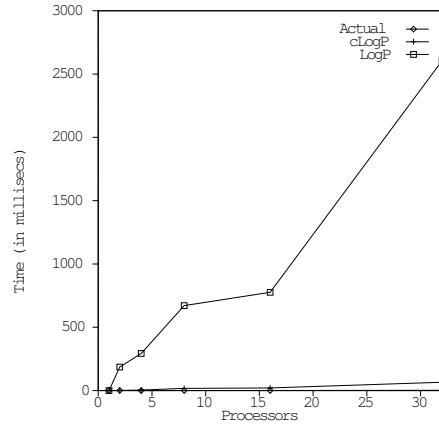


Figure 26: CG on Mesh : Contention

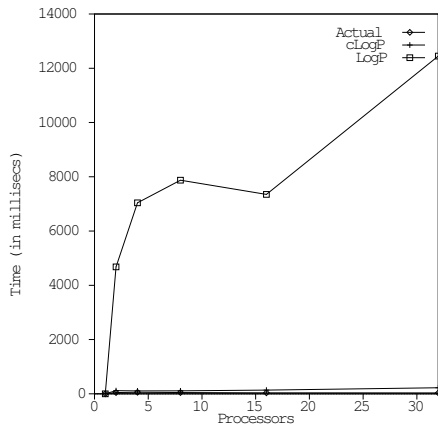


Figure 27: CHOLESKY on Cube : Contention

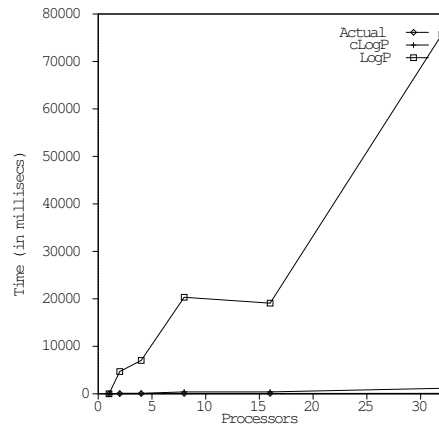


Figure 28: CHOLESKY on Mesh : Contention

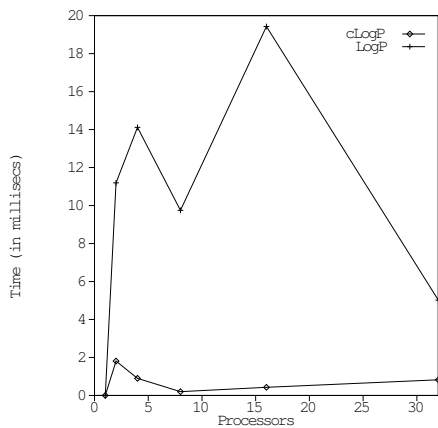


Figure 29: IS (Initial Implementation) on Full : Contention