Abstracting Network Characteristics and Locality Properties of 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

Abstracting features of parallel systems is a technique that has been traditionally used in theoretical and analytical models for program development and performance evaluation. In this paper, we explore the use of abstractions in execution-driven simulators in order to speed up simulation. In particular, we evaluate abstractions for the interconnection network and locality properties of parallel systems in the context of simulating cache-coherent shared memory (CC-NUMA) multiprocessors. We use the recently proposed LogP model to abstract the network. We abstract locality by modeling a cache at each processing node in the system which is maintained coherent, without modeling the overheads associated with coherence maintenance. Such an abstraction tries to capture the true communication characteristics of the application without modeling any hardware induced artifacts. Using a suite of applications and three network topologies simulated on a novel simulation platform, we show that the latency overhead modeled by LogP is fairly accurate. On the other hand, the contention overhead can become pessimistic when the applications display sufficient communication locality. Our abstraction for data locality closely models the behavior of the actual system over the chosen range of applications. The simulation model which incorporated these abstractions was around 250-300% faster than the simulation of the actual machine.

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 Motivation

Performance analysis of parallel systems[1] is complex due to the several degrees of freedom that exist in them. Developing algorithms for parallel architectures is also hard if one has to grapple with all parallel system artifacts. Abstracting features of parallel systems is a technique often employed to address both of these issues. For instance, abstracting parallel machines by theoretical models like the PRAM [14] has facilitated algorithm development and analysis. Such models try to hide hardware details from the programmer, providing a simplified view of the machine. Similarly, analytical models used in performance evaluation abstract complex system interactions with simple mathematical formulae, parameterized by a limited number of degrees of freedom that are tractable.

There is a growing awareness for evaluating parallel systems using applications due to the dynamic nature of the interaction between applications and architectures. Execution-driven simulation is becoming an increasingly popular vehicle for performance prediction because of its ability to accurately capture such complex interactions in parallel systems [25, 23]. However, simulating every artifact of a parallel system places tremendous requirements on resource usage, both in terms of space and time. A sufficiently abstract simulation model which does not compromise on accuracy can help in easing this problem. Hence, it is interesting to investigate the use of abstractions for speeding up execution-driven simulations which is the focus of this study. In particular, we address the issues of abstracting the *interconnection network* and *locality* properties of parallel systems.

Interprocess communication (both explicit via messages or implicit via shared memory), and locality are two main characteristics of a parallel application. The interconnection network is the hardware artifact that facilitates communication and an interesting question to be addressed is if it can be abstracted without sacrificing the accuracy of the performance analysis. Since latency and contention are the two key attributes of an interconnection network that impacts the application performance, any model for the network should capture these two attributes. There are two aspects to locality as seen from an application: communication locality and data locality. The properties of the interconnection network determine the extent to which communication locality is exploited. In this sense, the abstraction for the interconnection network subsumes the effect of communication locality. Exploiting data locality is facilitated either by private caches in shared memory multiprocessors, or local memories in distributed memory machines. Focussing only on shared memory multiprocessors, an important question that arises is to what extent caches can be abstracted

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

and still be useful in program design and performance prediction. It is common for most shared memory multiprocessors to have coherent caches, and the cache plays an important role in reducing network traffic. Hence, it is clear that any abstraction of such a machine has to model a cache at each node. On the other hand, it is not apparent if a simple abstraction can accurately capture the important behavior of caches in reducing network traffic.

We explore these two issues in the context of simulating Cache Coherent Non-Uniform Memory Access (CC-NUMA) shared memory machines. For abstracting the interconnection network, we use the recently proposed *LogP* [11] model that incorporates the two defining characteristics of a network, namely, latency and contention. For abstracting the locality properties of a parallel system, we model a private cache at each processing node in the system to capture data locality[2]. Shared memory machines with private caches usually employ a protocol to maintain coherence. With a diverse range of cache coherence protocols, it would become very specific if our abstraction were to model any particular protocol. Further, memory references (locality) are largely dictated by application characteristics and are relatively independent of cache coherence protocols. Hence, instead of modeling any particular protocol, we choose to maintain the caches coherent in our abstraction but do not model the overheads associated with maintaining the coherence. Such an abstraction would represent an ideal coherent cache that captures the true inherent locality in an application.

The study uses an execution-driven simulation framework which identifies, isolates, and quantifies the different overheads that arise in a parallel system. Using this framework, we simulate the execution of five parallel applications on three different machine characterizations: an *actual* machine, a *LogP* machine and a *cLogP* machine. The actual machine simulates the pertinent details of the hardware. The LogP machine does not model private caches at processing nodes, and abstracts the interconnection network using the LogP model. The cLogP machine abstracts the locality properties using the above mentioned scheme, and abstracts the interconnection network using the LogP model. To answer the first question, we compare the simulation of the actual machine to the simulation of the cLogP machine. If the network overheads of the two simulations agree then we have shown that LogP is a good abstraction for the network. To answer the second question, we compare the network traffic generated by the actual and cLogP machines. If they agree, then it shows that our abstraction of the cache is sufficient to model locality. Incidentally, the difference in results between the actual and LogP simulations would quantify the impact of locality on performance. If the difference is substantial (as we would expect it to be), then it shows that locality cannot be abstracted

[2]Note that the communication locality is subsumed in the abstraction for the interconnection network. Thus in the rest of the paper (unless explicitly stated otherwise) we use the term 'locality' to simply mean data locality.

out entirely.

Our results show that the latency overhead modeled by LogP is fairly accurate. On the other hand, the contention overhead modeled by LogP can become pessimistic for some applications due to failure of the model to capture communication locality. The pessimism gets amplified as we move to networks with lower connectivity. With regard to the data locality question, results show that our ideal cache, which does not model any coherence protocol overheads, is a good abstraction for capturing locality over the chosen range of applications. Abstracting the network and cache behavior also helped lower the cost of simulation by a factor of 250-300%. Given that execution-driven simulations of real applications can take an inordinate amount of time (some of the simulations in this study take between 8-10 hours), this factor can represent a substantial saving in simulation time.

Section 2 addresses related work and section 3 gives details on the framework that has been used to conduct this study. We use a set of applications (Section 4) and a set of architectures (Section 5) as the basis to address these questions. Performance results are presented in Section 6 and a discussion of the implication of the results is given in Section 7. Section 8 presents concluding remarks.

2 Related Work

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 [4], memory access conflicts [5], and cost of synchronization [15, 9]. The Bulk Synchronous Parallel (BSP) model [28] and the LogP model [11] are departures from the PRAM models, and attempt to realistically bridge the gap between theory and practice. Similarly, considerable effort has been expended in the area of performance evaluation in developing simple analytical abstractions to model the complex behavior of parallel systems. For instance, Agarwal [2] and Dally [12] develop mathematical models for abstracting the network and studying network properties. Patel [19] analyzes the impact of caches on multiprocessor performance. But many of these models make simplifying assumptions about the hardware and/or the applications, restricting their ability to model the behavior of real parallel systems.

Execution-driven simulation is becoming increasingly popular for capturing the dynamic behavior of parallel systems [25, 8, 10, 13, 21]. Some of these simulators 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. Researchers have tried to use other abstractions for the workload as well as the simulated hardware in order to speed up the simulation. In [29] a Petri net model is used for the application and the hardware. Mehra et al. [17] use application knowledge in abstracting out phases of the execution.

The issue of locality has been well investigated in the architecture community. Several studies [3, 16] have explored hardware facilities that would help exploit locality in applications, and have clearly illustrated the use of caches in reducing network traffic. There have also been application-driven studies which try to synthesize cache requirements from the application viewpoint. For instance, Gupta et al. [22] show that a small-sized cache of around 64KB can accommodate the important working set of many applications. Similarly, Wood et al. [30] show that the performance of a suite of applications is not very sensitive to different cache coherence protocols. But from the performance evaluation viewpoint, there has been little work done in developing suitable abstractions for modeling the locality properties of a parallel system which can be used in an execution-driven simulator.

3 The Framework

In this section, we present the framework that is used to answer the questions raised earlier. We give details of the three simulated machine characterizations and the simulator that has been used in this study.

The "actual" machine is a CC-NUMA shared memory multiprocessor. Each node in the system has a piece of the globally shared memory and a private cache that is maintained sequentially consistent using an invalidation-based (Berkeley protocol) fully-mapped directory-based cache coherence scheme. All the details of the interconnection network and coherence maintenance are exactly modeled.

3.1 The LogP Machine

The LogP model proposed by Culler et al. [11] 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 not likely to encounter contention.

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, and we do not discuss it in the rest of this paper. Our LogP machine is thus a collection of processors, each with a piece of the globally shared memory, connected by a network which is abstracted by the L and g parameters. Due to the absence of caches, any non-local memory reference would need to traverse the network as in a NUMA machine like the Butterfly GP-1000. In our simulation of this machine, each message in the network incurs a latency L that accounts for the actual transmission time of the message. In addition, a message may incur a waiting time at the sending/receiving node as dictated by the g parameter. For instance, when a node tries to send a message, it is ensured that at least g time units have elapsed since the last network access at that node. If not the message is delayed appropriately. A similar delay may be experienced by the message at the receiving node. These delays are expected to model the contention that such a message would encounter on an actual network.

3.2 The cLogP Machine

The LogP machine augmented with an abstraction for a cache at each processing node is referred to as a cLogP machine. A network access is thus incurred only when the memory request cannot be satisfied by the cache or local memory. The caches are maintained coherent conforming to a sequentially consistent memory model. With a diverse number of cache coherence protocols that exist, it would become very specific if cLogP were to model any particular protocol. Further, the purpose of the cLogP model is to verify if a simple minded abstraction for the cache can closely model the behavior of the corresponding "actual" machine without having to model the details of any specific cache coherence protocol, since it is not the intent of this study to compare different cache coherence protocols. In the cLogP model the caches are maintained consistent using an invalidation based protocol (Berkeley protocol), but the overhead

for maintaining the coherence is not modeled. For instance, consider the case where a block is present in a valid state in the caches of two processors. When a processor writes into the block, an invalidation message would be generated on the "actual" machine, but there would not be any network access for this operation on the cLogP machine. The block would still change to 'invalid' state on both machines after this operation. A read by the other processor after this operation, would incur a network access on both machines. cLogP thus tries to capture the true communication characteristics of the application, ignoring overheads that may have been induced by hardware artifacts, representing the minimum number of network messages that any coherence protocol may hope to achieve. If the network accesses incurred in the cLogP model are significantly lower than the accesses on the "actual" machine, then we would need to make our cLogP abstraction more realistic. But our results (to be presented in section 6) show that the two agree very closely over the chosen range of applications, confirming our choice for the cLogP abstraction in this study. Furthermore, if the actual machine implements a fancier cache coherence protocol (which would reduce the network accesses even further), that would only enhance the agreement between the results for the cLogP and actual machines.

3.3 SPASM

In this study, we use an execution-driven simulator called SPASM (Simulator for Parallel Architectural Scalability Measurements) that enables us to accurately model the behavior of applications on a number of simulated hardware platforms. SPASM has been written using CSIM [18], a process oriented sequential simulation package, and currently runs on SPARCstations. The input to the simulator are parallel applications written in C. These programs are pre-processed (to label shared memory accesses), the compiled assembly code is augmented with cycle counting instructions, and the assembled binary is linked with the simulator code. As with other recent simulators [8, 13, 10, 21], bulk of the instructions is executed at the speed of the native processor (the SPARC in this case) and only instructions (such as LOADs and STOREs on a shared memory platform or SENDs and RECEIVEs on a message-passing platform) that may potentially involve a network access are simulated. The reader is referred to [26, 25] for a detailed description of SPASM where we illustrated its use in studying the scalability of a number of parallel applications on different shared memory [25] and message-passing [26] platforms. The input parameters that may be specified to SPASM are the number of processors, the CPU clock speed, the network topology, the link bandwidth and switching delays.

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. The profiling capabilities of SPASM (outlined in [25]) provide a novel isolation and quantification of different overheads in a parallel system that contribute to the performance of the parallel system. These overheads may be broadly separated into a purely algorithmic component, and an interaction component arising from the interaction of the algorithm with the architecture. The algorithmic overhead arises from factors such as the serial part and work-imbalance in the algorithm, and is captured by the *ideal time* metric provided by SPASM. Ideal time is the time taken by the parallel program to execute on an ideal machine such as the PRAM [31]. This metric includes the algorithmic overheads but does not include any overheads arising from architectural limitations. Of the interaction component, the *latency* and *contention* introduced by network limitations are the important overheads that are of relevance to this study. The time that a message would have taken for transmission in a contention free environment is charged to the latency overhead, while the rest of the time spent by a message in the network waiting for links to become free is charged to the contention overhead.

The separation of overheads provided by SPASM plays a crucial role in this study. For instance, even in cases where the overall execution times may agree, the latency and contention overheads provided by SPASM may be used to validate the corresponding estimates provided by the L and g parameters in LogP. Similarly, the latency overhead (which is an indication of the number of network messages) in the actual and cLogP machine may be used to validate our locality abstraction in the cLogP model. In related studies, we have illustrated the importance of separating parallel system overheads in scalability studies of parallel systems [25], identifying parallel system (both algorithmic and architectural) bottlenecks [25], and synthesizing architectural requirements from an application viewpoint [27].

4 Application Characteristics

Three of the applications (EP, IS and CG) used in this study are from the NAS parallel benchmark suite [7]; CHOLESKY is from the SPLASH benchmark suite [24]; and FFT is the well-known Fast Fourier Transform algorithm. EP and FFT are well-structured applications with regular communication patterns determinable at compile-time, with the difference that EP has a higher computation to communication ratio. IS also has a regular communication pattern, but in addition it uses locks for mutual exclusion during the execution. CG and CHOLESKY are different from the other applications in that their communication patterns are not regular (both use sparse matrices) and cannot be determined at compile time. While a certain number of

rows of the matrix in CG is assigned to a processor at compile time (static scheduling), CHOLESKY uses a dynamically maintained queue of runnable tasks. The appendix gives further details of the applications.

5 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.

The study is conducted for the following 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 wormhole routing strategy. Message-sizes can vary upto 32 bytes. The switching delay is assumed to be negligible compared to the transmission time and we ignore it in this study.

Each node in the simulated CC-NUMA hierarchy is assumed to have 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. The private cache modeled in the "actual" and the "cLogP" machines is a 2-way set-associative cache (64KBytes with 32 byte blocks) that is maintained sequentially consistent using an invalidation-based (Berkeley protocol) fully-mapped directory-based cache coherence scheme. The L parameter for a message on the LogP and cLogP models is chosen to be 1.6 microseconds assuming 32-byte messages and a link bandwidth of 20 MBytes/sec. Similar to the method used in [11], 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).

6 Performance Results

The simulation results for 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 contention overheads for the execution of the applications on the three network topologies. We confine our discussion to the specific results that are relevant to the questions raised earlier. EP, FFT, and IS are applications with statically determinable memory reference patterns (see the appendix). Thus, in implementing these applications we ensured that the amount of communication (due to non-local references) is minimized. On the other hand, CG and CHOLESKY preclude any such optimization owing to their dynamic memory reference patterns.

6.1 Abstracting the Network

For answering the question related to network abstractions, we compare the results obtained using the cLogP and the actual machines. From Figures 1, 2, 3, 4, and 5, we observe that the latency overhead curves for the cLogP machine display a trend (shape of the curve) very similar to the actual machine thus validating the use of the L-parameter of the LogP model for abstracting the network latency. For the chosen parallel systems, there is negligible difference in latency overhead across network platforms since the size of the messages and transmission time dominate over the number of hops traversed. 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 the results for the cLogP and actual machines. Therefore, we show the results for only the fully connected network. Despite this similar trend, there is a difference in the absolute values for the latency overheads. cLogP models L as the time taken for a cache-block (32 bytes) transfer. But some messages may actually be shorter making L pessimistic with respect to the actual machine. On the other hand, cLogP does not model coherence traffic thereby incurring fewer network messages than the actual machine, 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 1) and CG (Figure 2) since there is very little coherence related activity in these two applications; while the optimism favors cLogP in IS (Figure 4) and CHOLESKY (Figure 5) where coherence related activity is more prevalent. However, it should be noted that these differences in absolute values are quite small implying that the L parameter pretty closely models the latency attribute.

Figures 6, 7, 8, 9, and 10, show that the contention overhead curves for the cLogP machine display a

trend (shape of the curves) similar to the actual machine. But there is a difference in the absolute values. The g-parameter in cLogP is estimated using the bisection bandwidth of the network as suggested in [11]. Such an estimate assumes that every message in the system traverses the bisection and can become very pessimistic when the application displays sufficient communication locality [1, 2]. This pessimism increases as the connectivity of the network decreases (as can be seen in Figures 6, 7, and 8) since the impact of communication locality increases. This pessimism is amplified further for applications such as EP that display a significant amount of communication locality. This effect can be seen in Figures 11, 12, and 13 which show a significant disparity between the contention on the cLogP and actual machines. 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 FFT on the cLogP machine (Figure 14). These results indicate that the contention estimated by the g parameter can turn out to be pessimistic, especially when the application displays sufficient communication locality. Hence, we need to find a better parameter for estimating the contention overhead, or we would at least need to find a better way of estimating g that incorporates application characteristics.

6.2 Abstracting Locality

Recall that our abstraction for locality attempts to capture the inherent data locality in an application. The number of messages generated on the network due to non-local references 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 compared to a cube.

The impact of ignoring locality in a performance model is illustrated by comparing the execution time curves for the LogP and cLogP machines. Of the three static applications (EP, FFT, IS), EP has the highest computation to communication ratio, followed by FFT, and IS. Since the amount of communication in EP is minimal, there is agreement in the results for the LogP, the cLogP, and the actual machines (Figure 20), regardless of network topology. On the fully connected and cube networks there is little difference in the results for FFT as well, whereas for the mesh interconnect the results are different between LogP and cLogP (Figure 21). The difference is due to the fact that FFT has more communication compared to EP, and the effect of non-local references is amplified for networks with lower connectivity. For IS (see Figure 22), which has even more communication than FFT, there is a more pronounced difference between

LogP and cLogP on all three networks. For applications like CG and CHOLESKY which exhibit dynamic communication behavior, the difference between LogP and cLogP curves is more significant (see Figures 23 and 24) since the LogP implementation cannot be optimized statically to exploit locality. Further, as we move to networks with lower connectivity, the LogP execution curves for CG and CHOLESKY (see Figures 25, 26, 27, and 28) do not even follow the shape of the cLogP execution curves. This significant deviation of LogP from cLogP execution is due to the amplified effect of the large amount of communication stemming from the increased contention in lower connectivity networks (see Figures 15, 16, 17, and 18).

Isolating the latency and contention overheads from the total execution time (see section 3) helps us identify and quantify locality effects. Figures 1, 2, and 3, illustrate some of these effects for FFT, CG, and EP respectively. During the communication phase in FFT, a processor reads consecutive data items from an array displaying spatial locality. In either the cLogP or the actual machine, a cache-miss on the first data item brings in the whole cache block (which is 4 data items). On the other hand, in the LogP machine all four data items result in network accesses. Thus FFT on the LogP machine incurs a latency (Figure 1) which is approximately four times that of the other two. Similarly, ignoring spatial and temporal locality in CG (Figure 2) results in a significant disparity for the latency overhead in the LogP machine compared to the other two. In EP, a processor waits on a condition variable to be signalled by another (see the appendix). For EP on a cLogP machine, only the first and last accesses to the condition variable use the network, while on the LogP machine a network access would be incurred for each reference to the condition variable as is reflected in Figure 3. Similarly, a test-test&set primitive [6], would behave like an ordinary test&set operation in the LogP machine thus resulting in an increase of network accesses. As can be seen in Figure 20, these effects do not impact the total execution time of EP since computation dominates for this particular application.

The above results confirm the well known fact that locality cannot be ignored in a performance prediction model or in program development. On the other hand, the results answer the more interesting question of whether the simple abstraction we have chosen for modeling locality in cLogP is adequate, or if we have to look for a more accurate model. cLogP does a fairly good job of modeling the cache behavior of the actual machine. The above results clearly show that the execution curves of cLogP and the actual machine are in close agreement across all application-architecture combinations. Further, the latency overhead curves (which are indicative of the number of messages exchanged between processors) of cLogP and the actual machine are also in close agreement. This suggests that our simple abstraction for locality in cLogP, an ideal coherent cache with no overhead associated with coherence maintenance, is sufficient to model the

locality properties over the chosen range of applications.

7 Discussion

We considered the issues pertaining to abstracting network characteristics and locality in this study in the context of five parallel scientific applications with different characteristics. The interprocess communication and locality behavior of three of these applications can be determined statically, but they have different computation to communication ratios. For the other two applications, the locality and the interprocess communication are dependent on the input data and are not determinable statically. The applications thus span a diverse range of characteristics. The network topologies (full, cube, mesh) also have diverse connectivities. The observations from our study are summarized below:

On Network Abstractions

The separation of overheads provided by SPASM has helped us evaluate the use of L and g parameters of the LogP model for abstracting the network. In all the considered cases the latency overhead from the model and the actual network closely agree. The pessimism in the model of assuming L to be the latency for the maximum size message on the network does not seem to have a significant impact on the accuracy of the latency overhead. Incidentally, we made a conscious decision in the cLogP simulation to abstract the specifics of the coherence protocol by ignoring the overheads associated with the coherence actions. The results show that the ensuing optimism does not impact the accuracy of the latency overhead either.

On the other hand, there is a disparity between the model and the actual network for the contention overhead in many cases. The two sources of disparity are (a) the way g is computed, and (b) the way g is to be used as defined by the model. Since g is computed using only the bisection bandwidth of the network (as is suggested in [11]), it fails to capture any communication locality resulting from mapping the application on to a specific network topology. The ensuing pessimism in the observed contention overhead would increase with decreasing connectivity in the network as we have seen in the previous section. There is also a potential for the model to be optimistic with respect to the contention overhead when two distinct source-destination pairs share a common link. The second source of disparity leads purely to a pessimistic estimate of the contention overhead. The node architecture may have several ports that gives the potential for simultaneous network activity from a given node. However, the model definition precludes even simultaneous “sends" and “receives" from a given node.

As can be seen from our results, the pessimistic effects in computing and using g dominates the observed

contention overheads. While it may be difficult to change the way g is computed within the confines of the LogP model, at least the way it is used should be modified to lessen the pessimism. For example, we conducted a simple experiment for FFT on the cube allowing for the g gap only between identical communication events (such as sends for instance). As can be seen in Figure 19, the resulting contention overhead (the curve for cLogP send/receive) is much closer to the real network.

The disparity in the contention prediction suggests that we need to incorporate application characteristics in computing g. For static applications like EP, IS and FFT, we may be able to use the computation and communication pattern in determining g. But for applications like CG and CHOLESKY, dynamism precludes such an analysis. On the other hand, since we are using these models in an execution driven simulation, we may be able get a better handle on calculating g. For instance, we may be able to maintain a history of the execution and use it to calculate g. It would be interesting to investigate such issues in arriving at a better estimate.

On Locality Abstraction

As we expected, locality is an important factor in determining the performance of parallel programs and cannot be totally abstracted away for performance prediction or performance-conscious program development. But locality in parallel computation is much more difficult to model due to the additional degrees of freedom compared to sequential computation. Even for static applications, data alignment (several variables falling in the same cache block as observed in FFT) and temporal interleaving of memory accesses across processors, are two factors that make abstracting locality complex. In dynamic applications, this problem is exacerbated owing to factors such as dynamic scheduling and synchronization (implicit synchronization using condition variables and explicit synchronizers such as locks and barriers). It is thus difficult to abstract locality properties of parallel systems by a static theoretical or analytical model. Hence, in this study we explored the issue of using an abstraction for locality in a dynamic execution-driven simulation environment. In particular, we wanted to verify if a simple abstraction of a cache at each processing node that is maintained coherent without modeling the overheads for coherence maintenance would suffice to capture the locality properties of the system. Such an abstraction would try to capture the true communication characteristics of the application without modeling any hardware induced artifacts. Our results show that such an abstraction does indeed capture the locality of the system, closely modeling the communication in the actual machine.

The network messages incurred in our abstraction for locality is representative of the minimum overhead that any cache coherence protocol may hope to achieve. We compared the performance of such an abstraction

with a machine that incorporates an invalidation based protocol. Even for this simple protocol, the results of the two agree closely over the chosen range of applications. The performance of a fancier cache coherence protocol that reduces network traffic on the actual machine is expected to agree even closer with the chosen abstraction. This result suggests that cache coherence overhead is insignificant at least for the set of applications considered, and hence the associated coherence-related network activity can be abstracted out of the simulation. The applications that have been considered in this study employ the data parallel paradigm which is representative of a large class of scientific applications. In this paradigm, each processor works with a different portion of the data space, leading to lower coherence related traffic compared to applications where there is a more active sharing of the data space. It may be noted that Wood et al. [30] also present simulation results showing that the performance of a suite of applications is not very sensitive to different cache coherence protocols. However, further study with a wider suite of applications is required to validate this claim.

Importance of Separating Parallel System Overheads

The isolation and quantification of parallel system overheads provided by SPASM helped us address 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. One can experimentally determine the accuracy of the performance predicted by the LogP model as is done in [11] using the CM-5. However, this approach does not validate the individual parameters abstracted using the model. On the other hand, 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.

Speed of Simulation

Our main reason in studying the accuracy of abstractions is so that they may be used to speed up execution-driven simulations. Intuitively, one would think that the LogP machine described in this paper would execute the fastest since it is the most abstract of the three. But, our simulations of the LogP machine took a longer time to complete than those of the actual machine. This is because such a model is very pessimistic due to ignoring data locality and the way it accounts for network contention. Hence, the simulation encountered considerably more events (non-local accesses which are cache 'hits' in the actual and cLogP machines result in network accesses in the LogP machine) making it execute slower. On the other hand, the simulation of

cLogP, which is less pessimistic, is indeed around 250-300% faster than the simulation of the actual machine. This factor can represent a substantial saving given that execution-driven simulation of real applications can take an inordinate amount of time. For instance, the simulation of some of the data points for CHOLESKY take between 8-10 hours for the actual machine. If we can reduce the pessimism in cLogP in modeling contention, we may be able to reduce the time for simulation even further.

8 Concluding Remarks

Abstractions of machine artifacts are useful in a number of settings. Execution-driven simulation is one such setting. This simulation technique is a popular vehicle for performance prediction because of its ability to capture the dynamic behavior of parallel systems. However, simulating every aspect of a parallel system in the context of real applications places a tremendous requirement on resource usage, both in terms of space and time. In this paper, we explored the use of abstractions in alleviating this problem. In particular, we explored the use of abstractions in modeling the interconnection network and locality properties of parallel systems in an execution-driven simulator. We used the recently proposed LogP model to abstract the interconnection network. We abstracted the locality in the system by modeling a coherent private cache without accounting for the overheads associated with coherence maintenance. We used five parallel scientific applications and hardware platforms with three different network topologies to evaluate the chosen abstractions. The results of our study show that the network latency overhead modeled by LogP is fairly accurate. On the other hand, the network contention estimate can become very pessimistic, especially in applications which exhibit communication locality. With regard to the data locality issue, the chosen simple abstraction for the cache does a good job in closely modeling the locality of the actual machine over the chosen range of applications. The simulation speed of the model which incorporated these two abstractions was around 250-300% faster than the simulation of the actual hardware, which can represent a substantial saving given that simulation of real parallel systems can take an inordinate amount of time.

References

[1] V. S. Adve and M. K. Vernon. Performance analysis of mesh interconnection networks with deterministic routing. *IEEE Transactions on Parallel and Distributed Systems,* 5(3):225–246, March 1994.

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

[3] A. Agarwal et al. The MIT Alewife machine : A large scale Distributed-Memory Multiprocessor. In *Scalable shared memory multiprocessors*. Kluwer Academic Publishers, 1991.

[4] A. Aggarwal, A. K. Chandra, and M. Snir. On Communication Latency in PRAM Computations. In *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 11–21, 1989.

[5] 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.

[6] 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.

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

[8] 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.

[9] R. Cole and O. 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.

[10] 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.

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

[12] W. J. Dally. Performance analysis of *k*-ary *n*-cube interconnection networks. *IEEE Transactions on Computer Systems*, 39(6):775–785, June 1990.

[13] H. Davis, S. R. Goldschmidt, and J. L. Hennessy. Multiprocessor Simulation and Tracing Using Tango. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages II 99–107, 1991.

[14] 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.

[15] P. B. Gibbons. A More Practical PRAM Model. In *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 158–168, 1989.

[16] D. Lenoski, J. Laudon, K. Gharachorloo, W-D Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[17] P. Mehra, C. H. Schulbach, and J. C. Yan. A comparison of two model-based performance-prediction techniques for message-passing parallel programs. In *Proceedings of the ACM SIGMETRICS 1994 Conference on Measurement and Modeling of Computer Systems*, pages 181–190, May 1994.

[18] Microelectronics and Computer Technology Corporation, Austin, TX 78759. *CSIM User's Guide*, 1990.

[19] J. 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] 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.

[23] J. P. Singh, E. Rothberg, and A. Gupta. Modeling communication in parallel algorithms: A fruitful interaction between theory and systems? In *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*, 1994.

[24] J. P. Singh, W-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-91-469, Computer Systems Laboratory, Stanford University, 1991.

[25] 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*, pages 171–180, May 1994.

[26] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. A Simulation-based Scalability Study of Parallel Systems. *Journal of Parallel and Distributed Computing*, 1994. To appear.

[27] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. Synthesizing network requirements using parallel scientific applications. Technical Report GIT-CC-94/31, College of Computing, Georgia Institute of Technology, July 1994.

[28] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[29] H. Wabnig and G. Haring. PAPS - The Parallel Program Performance Prediction Toolset. In *Proceedings of the 7th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, Vienna, Austria, May 1994.

[30] D. A. Wood et al. Mechanisms for cooperative shared memory. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 156–167, May 1993.

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

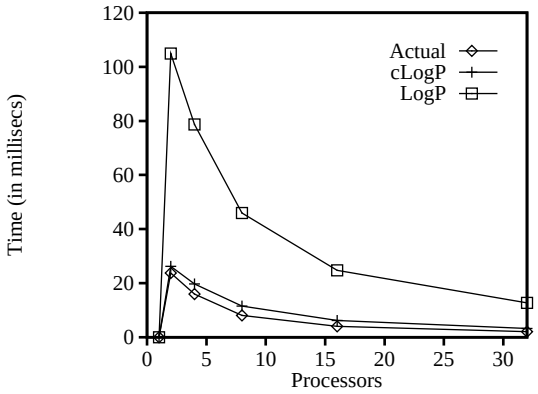

Figure 1: FFT on Full: Latency

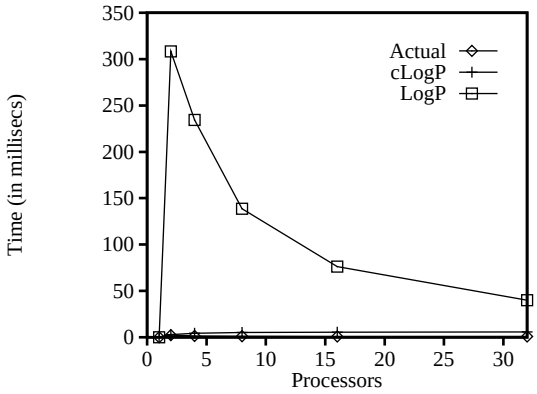

Figure 2: CG on Full: Latency

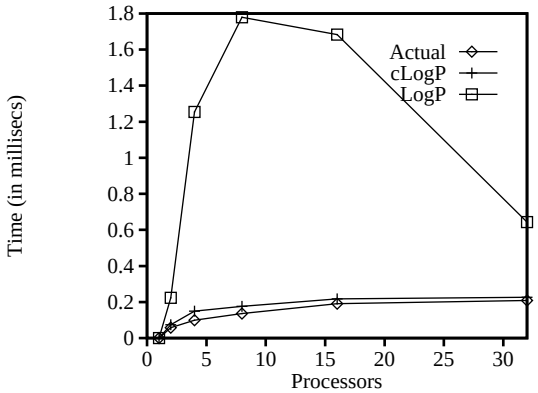

Figure 3: EP on Full: Latency

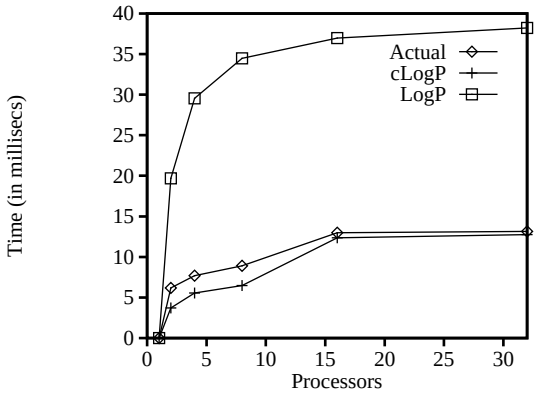

Figure 4: IS on Full: Latency

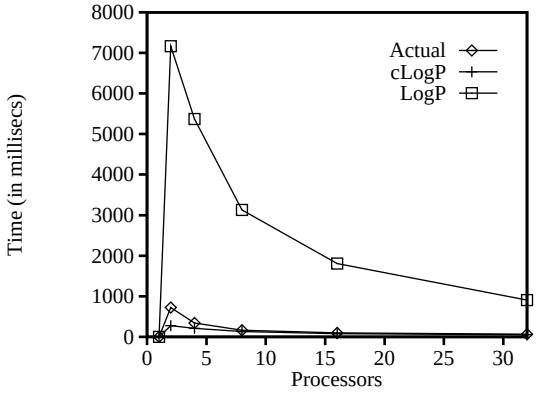

Figure 5: CHOLESKY on Full: Latency

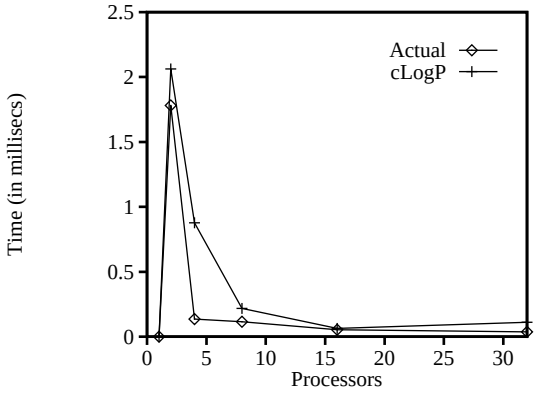

Figure 6: IS on Full: Contention

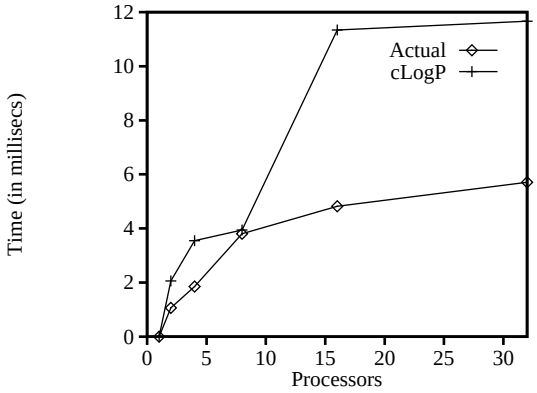

Figure 7: IS on Cube: Contention

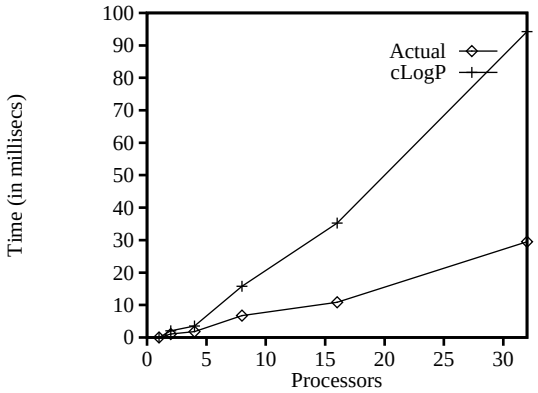

Figure 8: IS on Mesh: Contention

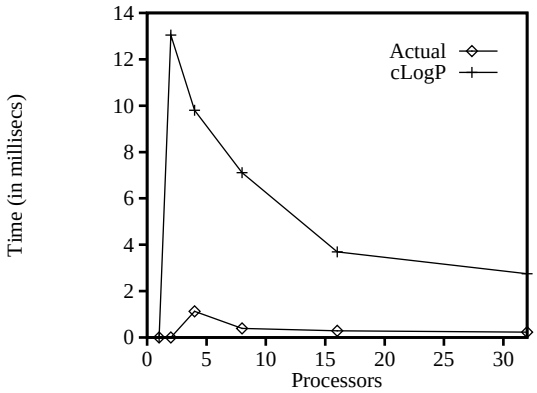

Figure 9: FFT on Cube: Contention

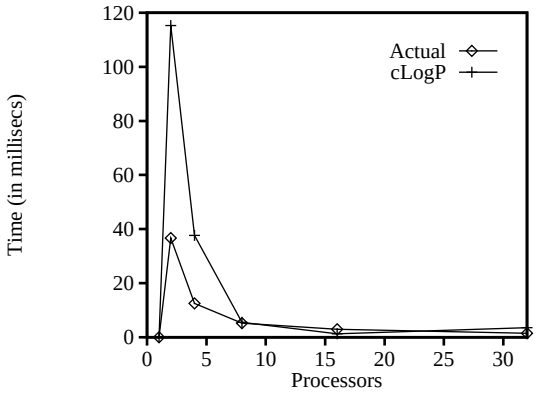

Figure 10: CHOLESKY on Full: Contention

Contention Overhead (contd)

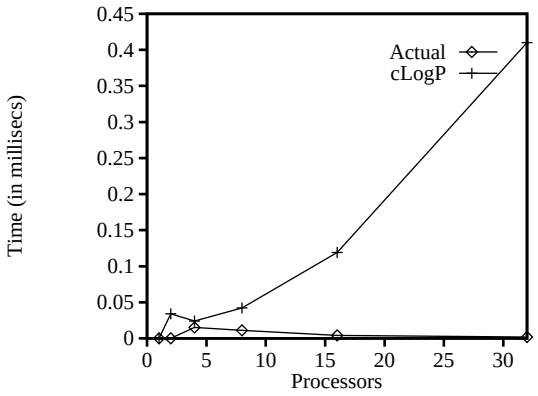

Figure 11: EP on Full: Contention

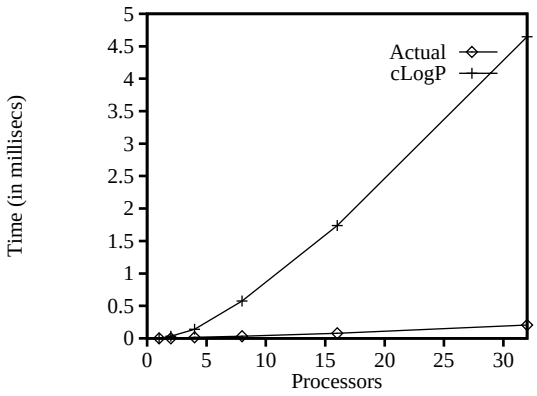

Figure 12: EP on Cube: Contention

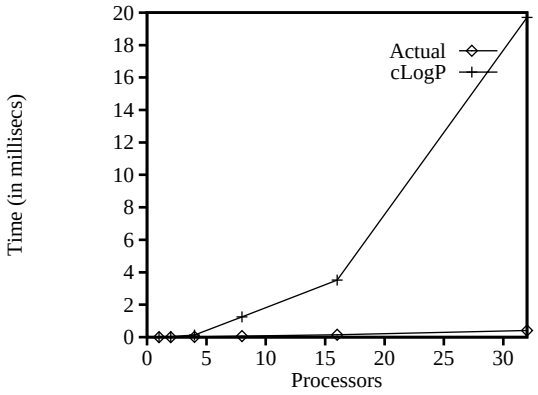

Figure 13: EP on Mesh: Contention

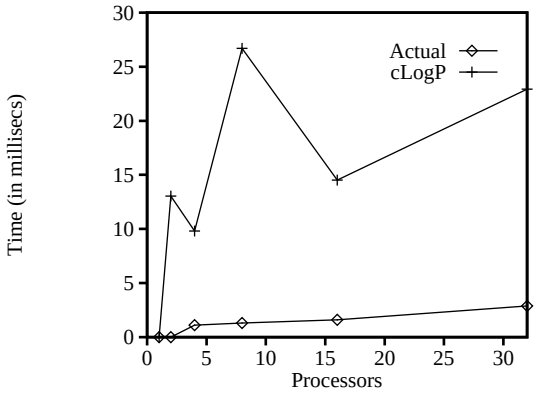

Figure 14: FFT on Mesh: Contention

Contention Overhead (contd)

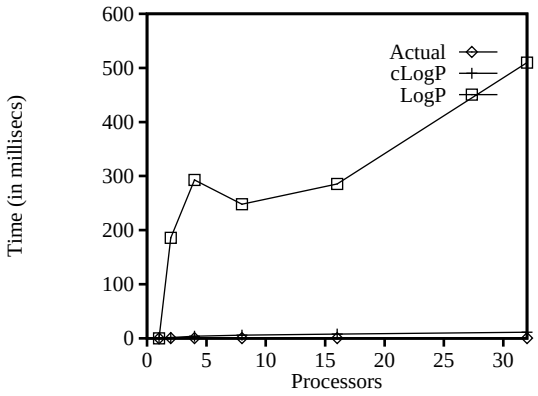

Figure 15: CG on Cube: Contention

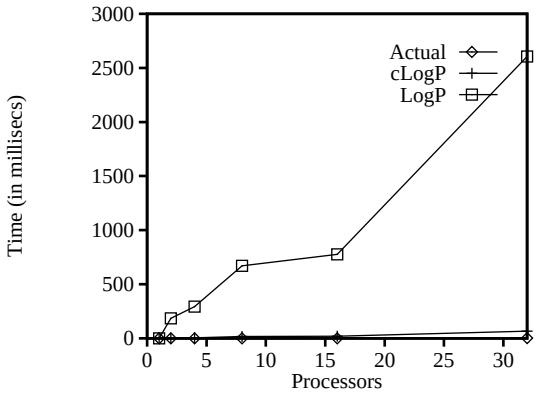

Figure 16: CG on Mesh: Contention

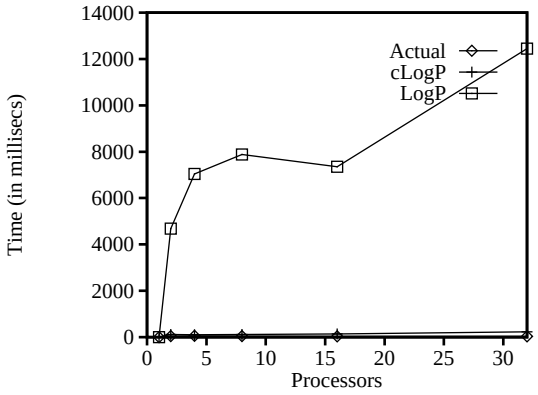

Figure 17: CHOLESKY on Cube: Contention

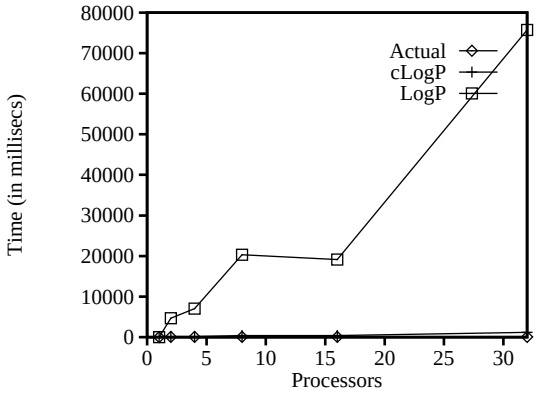

Figure 18: CHOLESKY on Mesh: Contention

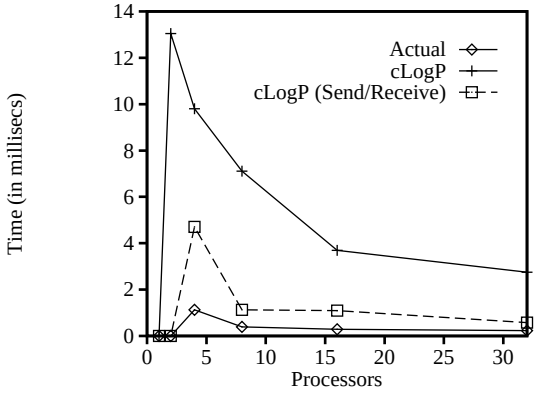

Figure 19: FFT on Cube: Contention (Send/Receive Separation)

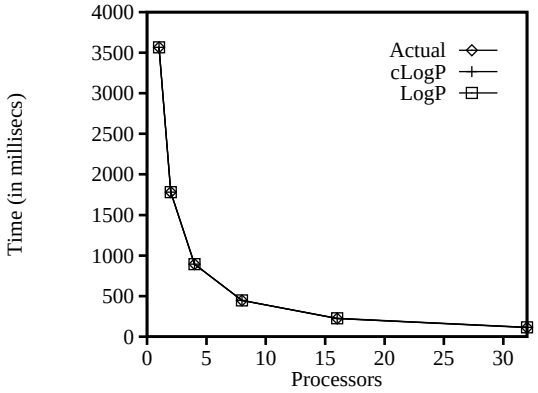

Figure 20: EP on Full

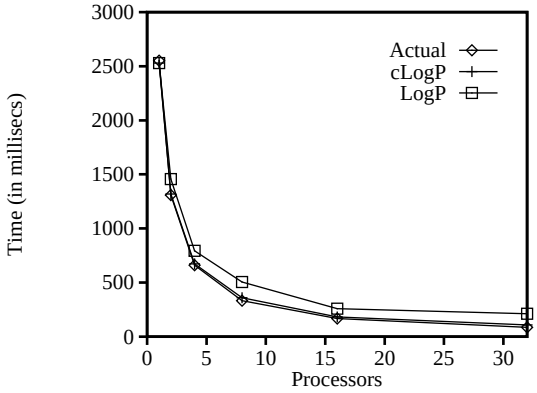

Figure 21: FFT on Mesh

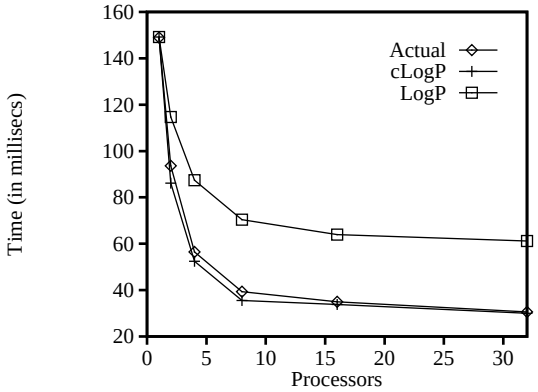

Figure 22: IS on Full

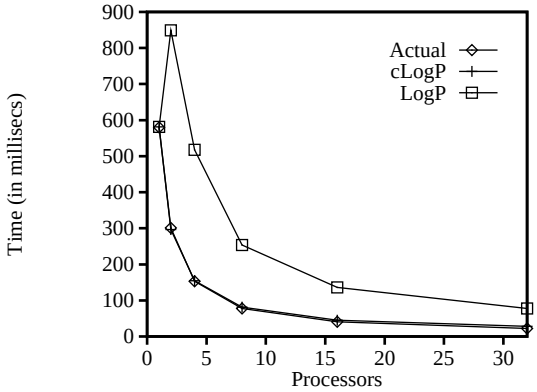

Figure 23: CG on Full

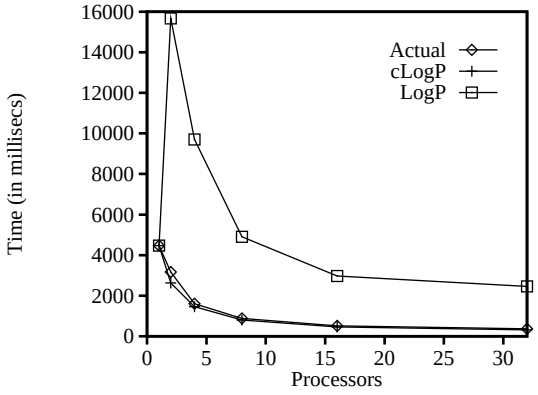

Figure 24: CHOLESKY on Full

Execution Time (contd)

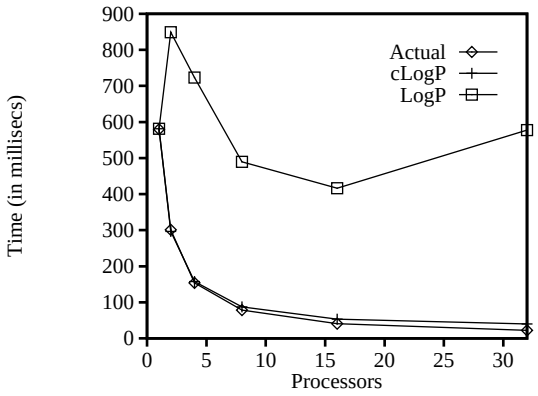

Figure 25: CG on Cube

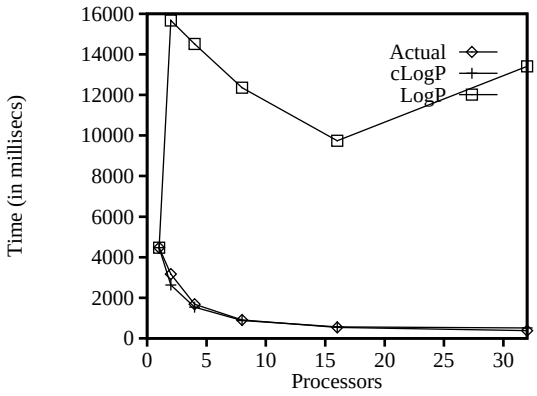

Figure 26: CHOLESKY on Cube

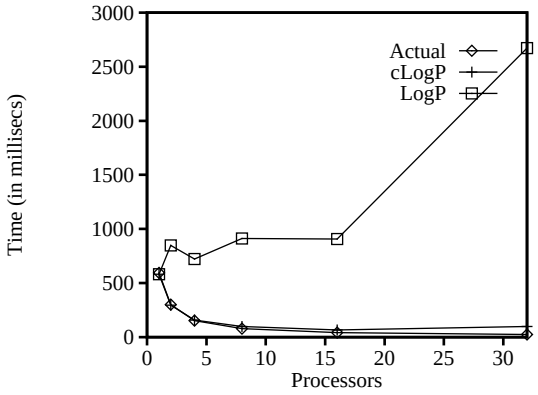

Figure 27: CG on Mesh

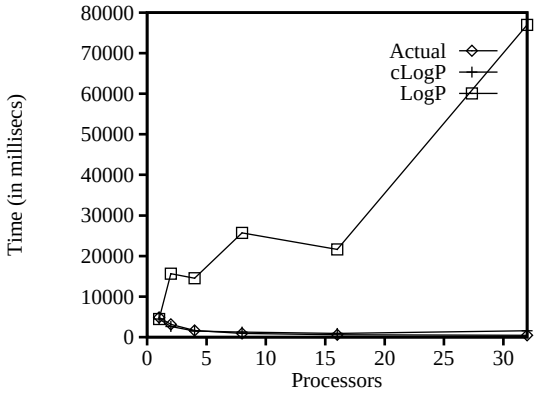

Figure 28: CHOLESKY on Mesh

Appendix

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 an "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 n (64K in this study) 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. 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 (Wait-Signal semantics), 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

IS is an "Integer Sort" application that uses bucket sort to rank a list of integers which is an important

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

operation in "particle method" codes. An implementation of the algorithm is described in [20] and Table 2 summarizes its characteristics. The input list of size n (64K in this study) is equally partitioned among the processors. Each processor maintains two sets of buckets. One set of buckets (of size $nbuckets$) is used to maintain the information for the portion of the list local to it. The other set (of size $chunk = nbuckets/p$ 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 [6]) 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 a one dimensional complex Fast Fourier Transform of n (64K in this study) points that plays an important role in Image and Signal processing. n is a power of 2 and greater than or equal to the square of the number of processors p. There are three important phases in the application. In the first and last phases, processors perform the radix-2 butterfly computation on n/p local points. The only communication is incurred in the middle phase in which the *cyclic* layout of data is changed to a *blocked* layout as described in [11]. 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 data ($\frac{n}{p^2}$ points) read from processor $i+1$ and ending with data read from processor $i-1$ in $p-1$ substeps. This communication schedule minimizes contention both in the network and at the processor ends. These three phases are separated by barriers.

CG

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

Table 4: Characteristics of CG

CG is a "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. The sparse matrix of size $n * n$ and the vectors are partitioned

by rows assigning an equal number of contiguous rows to each processor (static scheduling). We present the results for five iterations of the Conjugate Gradient Method in trying to approximate the solution of a system of linear equations. There is a barrier at the end of each iteration. Each iteration involves the calculation of a sparse matrix-vector product and two vector-vector dot products. These are the only operations that involve communication. The computation granularity between these operations is linear in the number of rows (the problem size) and involves a floating point addition and multiplication for each row. The vector-vector dot product is calculated by first obtaining the intermediate dot products for the elements in the vectors local to a processor. This is again a local operation with a computation granularity linear in the number of rows assigned to a processor with a floating point multiplication and addition performed for each row. A global sum of the intermediate dot products is calculated by a logarithmic reduce operation (as in EP) yielding the final dot product. For the computation of the matrix-vector product, each processor performs the necessary calculations for the rows assigned to it in the resulting matrix (which are also the same rows in the sparse matrix that are local to the processor). But the calculation may need elements of the vector that are not local to a processor. Since the elements of the vector that are needed for the computation are dependent on the randomly generated sparse matrix, the communication pattern for this phase is random. Table 4 summarizes the characteristics for each iteration of CG. A sparse matrix of size 1400X1400 containing 100,300 non-zeroes has been used in the study.

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 of size $n * n$. 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 [24] 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.