A Framework for Evaluating Architectural Issues of Parallel Systems

A Thesis
Presented to
The Academic Faculty

by

Anand Sivasubramaniam

In Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Georgia Institute of Technology
August 1995

A Framework for Evaluating Architectural Issues of Parallel Systems

Approved:

Umakishore Ramachandran (chairman)

H. Venkateswaran (co-advisor)

Richard M. Fujimoto

Sudhakar Yalamanchili

Mark D. Hill

Date Approved

Dedication

To my parents

Acknowledgments

I cannot thank enough:

- my advisors, Kishore Ramachandran and H. Venkateswaran, for their constant guidance, support and encouragement. Many a time when I was up against a wall, their direction showed the light at the end of the tunnel.
- the members of my thesis committee for their valuable suggestions and comments.
- the architecture and systems group at Georgia Tech for the prolonged discussions/arguments and to the specific members of the group who have co-authored several papers with me.
- the CNS staff at Georgia Tech for providing an excellent computing environment during my stay.
- my wife, Vidya, who has faithfully supported me and kept me company both at home and at school well past the midnight hour.
- my parents, who have taught me the value of education and have been instrumental in keeping alive the urge in me to strive for success in all my endeavors.

Several other persons have been either directly or indirectly responsible for making this dissertation possible. My sincere apologies for not being able to acknowledge them individually.

Contents

List of Tables

List of Figures

Summary

Parallel computing is becoming increasingly important for scientific advancement and economic development. Despite the promise of being able to meet the increased computational needs of the future, the delivered performance of parallel machines often falls short of the projected peak performance. Evaluating and analyzing the performance of a parallel application on an architecture to explain the disparity between projected and delivered performance has widespread applicability in parallel systems[1] research. But conducting such a study is hard due to the additional degrees of freedom exhibited by these systems compared to their sequential counterparts. In this thesis, we identify performance metrics that give detailed information on parallel system performance and outline a framework that has been developed to quantify these metrics. The framework uses an application-driven approach with a combination of experimental, simulation and analytical techniques.

The evaluation framework can be used to research a variety of issues in parallel systems. Using a suite of applications and a shared memory architectural platform, we illustrate the use of the framework with four case-studies. The first study evaluates the scalability of the chosen suite of applications and the results are used to enhance performance by application restructuring. The second study explores the use of the framework in validating theoretical/analytical models of parallel systems and illustrates how these models can be used to ease resource requirements of execution-driven simulators. The third study characterizes the communication bandwidth requirements of parallel applications and quantifies the necessary link bandwidth in the interconnection network for a desired level of performance. The last study focuses on minimizing/tolerating the network latency in the context of shared memory

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

multiprocessors by examining the communication characteristics of real applications and deriving a set of explicit communication mechanisms from these characteristics. The last two studies are two different architectural solutions to the same problem, namely, lowering the communication overhead in an application.

Chapter 1

Introduction

Things don't turn up in this world until somebody turns them up.
- James Garfield

It is hard to fail, but it is worse never to have tried to succeed.
- Theodore Roosevelt

The RISC [58] ideology in the evolution of uniprocessors clearly illustrates the need for using real-world applications to drive architectural design. There are several applications spanning diverse domains whose requirements in terms of space and time, may not be satisfied due to inherent limitations in existing sequential machines. Parallel machines promise to take on the challenge of meeting the computational demands imposed by such applications. Applications exhibit different characteristics [45, 40] thus imposing diverse demands on the underlying hardware, while parallel machines also come in several flavors [83, 12, 25, 84, 39, 8, 18, 29, 48, 4, 82, 55]. To find out how good a job a machine does of meeting an application's demands, we need a way of evaluating the match between an application and an architecture. We use the term, parallel system, to denote an application-architecture combination, and the term scalability to qualify the match between the application and the architecture in a parallel system. Evaluating the scalability of a parallel system has widespread applicability. The results from such an evaluation may be used to: select the best architecture platform for an application domain, select the best algorithm for solving the problem on a given hardware platform, predict the performance of an application on a larger configuration of an existing architecture, predict the performance of large application instances, identify application and architectural bottlenecks in a parallel system to suggest application restructuring and architectural enhancements, and glean insight on the interaction between an application and an architecture to predict the scalability of other application-architecture pairs. But evaluating and analyzing the scalability of parallel systems poses several problems due to the complex interaction between application characteristics and architectural features. In this thesis, we present a framework for addressing these problems and use this framework to answer interesting questions about parallel systems.

1.1 The Framework

Architectural studies generally address low-level issues such as network latency, contention [3, 59] and synchronization [9, 51]. While such issues are extremely important, it is necessary to put the impact of these factors into perspective by considering them in the context of overall application performance. Recognizing this importance, recent studies use real applications to address specific architectural issues [23, 65]. However, there have been very few attempts at quantifying the effects of application and architectural interactions in a parallel system. Our approach uses real-world applications to study these interactions and we call such an application-driven approach a *top-down approach to scalability study*.

Many metrics have been proposed to quantify the execution of an application on a given machine. While such metrics can be useful for tracking overall performance trends, they do not provide adequate information towards identifying application and architectural bottlenecks. To remedy this problem, we propose a set of metrics called *overhead functions* that identify, isolate and quantify the bottlenecks in a parallel system. Performance measurement by experimenting with the application on the given hardware, analytical modeling and simulation are three well-known performance evaluation techniques. Each has its own limitations. Performance measurement by experimentation is limited by the amount of statistics that one can glean from the evaluation. The accuracy of analytical models is limited by the validity of the simplifying assumptions used in deriving the models. Finally, simulations of real systems demand considerable resources in terms of space and time. Our framework uses a combination of the three techniques while alleviating some of their drawbacks. At the heart of this framework lies a simulation platform called SPASM (Simulator for Parallel Architectural Scalability Measurements) which can simulate the execution of applications on a range of hardware platforms and provide detailed statistics for quantifying the overhead functions.

1.2 Uses of the Framework

Our framework can be used to answer several interesting questions about parallel systems. Some of the questions which we address in this thesis are:

- *How do we predict the performance of an application on a given machine?*
  Predicting the performance of a parallel system as a function of application and architectural parameters can help determine the best architecture platform for an application, and to select the best algorithm for solving the problem on a given hardware platform. It can also give us insights on how performance is expected to change in the future for larger machine configurations and larger application instances.

- *How do we identify application and architectural bottlenecks in a parallel system and use this information to improve performance?*
  When a system does not scale as we would like it to, we need to identify and quantify the individual bottlenecks in the execution. This information can give useful feedback to the application programmer for restructuring the application, and to the system architect for designing architectural enhancements.

- *To what level of detail should we study the parallel system towards understanding its scalability?*

The numerous degrees of freedom exhibited by parallel systems makes performance evaluation hard. Hence, it is interesting to find out if details of a system can be elided to ease the evaluation process while obtaining reasonably accurate results.

- *How do we go about building well-balanced machines?*
  Given a finite amount of resources, a system architect would like to know how much to invest in each component of the parallel machine for maximum rewards. Further, he would also be interested in the rate at which the speed of different system components need to relatively grow in the future for building well-balanced machines. Using this information, he can make cost versus performance trade-offs in important architectural design decisions.

- *What mechanisms should we provide in a machine to match an application's demands?*
  A parallel machine provides a set of mechanisms for services such as process control, communication and synchronization. An application's implementation on such a platform would be constrained by the mechanisms provided by the architecture which can result in performance penalties. On the other hand, close scrutiny of an application's characteristics can suggest a more suitable set of mechanisms which can be beneficial. Examining characteristics over a range of applications can give better insight on what mechanisms should be provided in a machine.

Adhering to our top-down approach, applications are used as the driving force in answering the above questions.

1.3 Organization of the Thesis

Chapter 2 presents the evaluation framework with details on the overhead functions and the techniques used to calculate these metrics. Chapter 3 describes a simulation platform called SPASM that has been developed for conducting the evaluations. Using a set of applications, the rest of the thesis illustrates the power of the framework in answering the questions raised earlier. Both the novel evaluation methodologies and the useful insights drawn from each investigation are presented. Chapter 4 shows the use of the framework in studying the scalability of parallel systems. Chapter 5 investigates the issue of eliding parallel system details towards easing the resource requirements of execution-driven simulation - an important performance evaluation technique. Chapter 6 focuses on the interconnection network and quantifies the network bandwidth requirements of the chosen applications. Chapter 7 examines the communication characteristics of the applications and derives a set of mechanisms to lower communication overhead. Finally, chapter 8 summarizes the contributions of this thesis and outlines directions for future work.

Chapter 2

The Framework

I think it would be a good idea.
- Mahatma Gandhi (on western civilization)

A pessimist sees the difficulty in every opportunity;
an optimist sees the opportunity in every difficulty.
- Winston Churchill

In conducting any evaluation, we need to identify a set of performance metrics that we would like to measure and the techniques and tools that will be used. The term evaluation framework is used to denote a combination of the metrics, techniques and tools, and in this chapter we present the framework which will be used in our evaluations.

2.1 Performance Metrics

Metrics which capture processor characteristics in terms of the clock speed (MHz), the instruction execution speed (MIPS), the floating point performance (MFLOPS), and the execution time for standard benchmarks have been widely used in modeling uniprocessor performance. However, in a parallel system the hardware specification (which quantifies the available compute power) may never be a true indicator of the performance delivered by the system. This is due to overheads arising from complex interactions between application characteristics and architectural features. Metrics for parallel system performance evaluation should quantify this gap between available and delivered compute power. *Scalability* is a notion that is frequently used to express the disparity between the two, in terms of the match between an application and an architecture. Metrics proposed for scalability attempt to quantify this match. The reader is referred to the survey by Kumar and Gupta [43] for details on different performance metrics for scalability.

Parallel computers promise the following enhancements over their sequential counterparts, each of which leads to a corresponding scaling strategy: 1) the number of processing elements is increased enabling a potential performance improvement for the same problem (*constant problem size scaling*); 2) other system resources like primary and secondary storage are also increased enhancing the capability to solve larger problems (*memory-constrained scaling*); 3) due to the larger number of processing elements, a much larger problem may be solved in the same time it takes to solve a smaller problem on a sequential machine (*time-constrained scaling*). *Speedup(p)*, *scaled speedup* [36] and *sizeup* [81] are three performance metrics used in each of these scaling strategies respectively.

When there is a gap between available and delivered compute power, speedup, scaled-speedup and sizeup fail to show whether the problem rests with the application and/or the architecture. Three other metrics [44, 41, 56] attempt to address this deficiency. For a problem with a fixed size, the processor utilization (efficiency) normally decreases with an increase in the number of processors. Similarly, if we scale up the problem size keeping the number of processors fixed, the efficiency usually increases. *Isoefficiency function* [44] relates these two artifacts in typical parallel systems and is defined as the rate at which the problem size needs to grow with respect to the number of processors to keep the efficiency constant. Karp and Flatt [41] use an *experimentally determined serial fraction* f for a problem with a fixed size in evaluating parallel systems. This metric is computed by executing the application on the actual hardware and calculating the effective loss in speedup. On an ideal architecture with no overheads introduced by the hardware, f would be equal to the theoretical serial fraction of the application. Hardware deficiencies lead to a higher f and a constant

f with increasing number of processors implies a scalable parallel system very often suggesting that there is no overhead introduced by the underlying hardware. Nussbaum and Agarwal [56] quantify scalability as a ratio of the application's asymptotic speedup when run on the actual architecture to its corresponding asymptotic speedup when run on an EREW PRAM [30] for a given problem size.

Isoefficiency function, experimentally determined serial fraction, and Nussbaum and Agarwal's metric, attempt to identify the cause (the application or the architecture) of the problem when the parallel system does not scale as expected. Once the problem is identified, it is essential to find the individual application and architectural artifacts that lead to these bottlenecks and quantify their relative contribution towards limiting the overall scalability of the system. But it is difficult to glean this knowledge from the three metrics. Identifying, isolating and quantifying the different overheads in a parallel system that limit its scalability is crucial for improving the performance of the system by application restructuring and architectural enhancements.

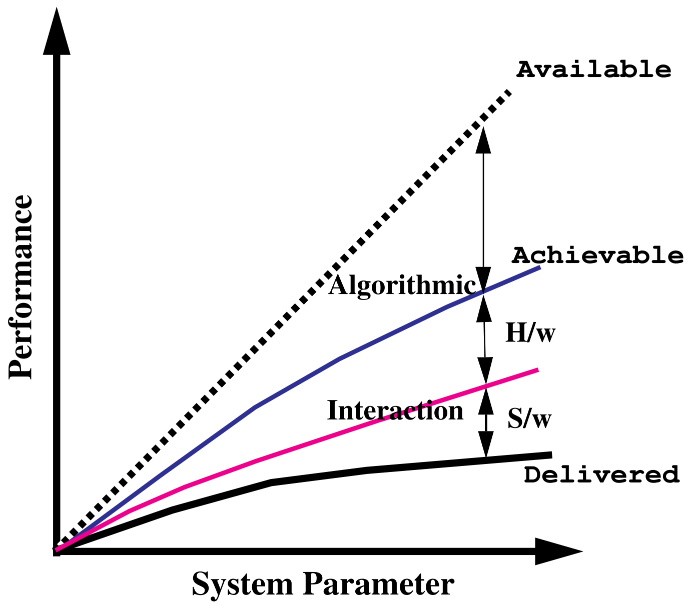


Figure 1: Overheads in a Parallel System

Let us assume that we are tracking performance as a function of some system parameter as shown in Figure 1. For instance, the number of processors can be the system parameter and speedup can be the performance metric. Ignoring effects of superlinearity, one would expect a speedup that increases linearly with an increase in the available processing power, as shown by the curve named "available" in Figure 1. But overheads in the system would limit its scalability resulting in a speedup that grows much slower as shown by the curve labeled "delivered" in Figure 1. The disparity between the "available" and "delivered" curves is due to growth of overheads in the system. Parallel system overheads may be broadly classified into a purely algorithmic component (*algorithmic overhead*), a component arising from the interaction of the application with the system software (*software interaction overhead*), and a component arising from the interaction of the application with the hardware (*hardware interaction overhead*). Algorithmic overheads arise from the inherent serial part in the application, the work-imbalance between the executing threads of control, any redundant computation that may be performed, and additional work introduced by the parallelization. The achievable performance of an application even on an ideal machine such as the PRAM [92] would be lower than the "available" curve as depicted by the "achievable" curve in Figure 1 due to algorithmic overheads.. The implementation of the application on an actual machine would result in further slowdown due to software and hardware interaction overheads. Software interaction overheads such as overheads for scheduling, message-passing, and software synchronization arise due to the interaction of the application with the system software. Hardware slowdown due to network latency (the transmission time for a message in the network), network contention (the amount of time spent in the network waiting for availability of network resources), synchronization and cache coherence actions, contribute to the hardware interaction overhead. To fully understand the scalability of a parallel system, it is important to isolate and quantify the impact of each of these components on the overall execution as shown in Figure 1. An *overhead function* [76, 75] is proposed to capture the growth of a particular system overhead with respect to a specific system parameter. Crovella and LeBlanc [19] propose a similar set of metrics called Lost Cycles. Both these metrics quantify the contribution of each overhead towards the overall execution time. The studies differ in the techniques used to quantify these metrics. Crovella and LeBlanc [19] use experimentation, while simulation is used in our approach.

2.2 Evaluation Techniques

Experimentation, analytical modeling and simulation are three well-known techniques for evaluating parallel systems. Experimentation involves implementing the application on the actual hardware and measuring its performance. Analytical models abstract hardware and application details in a parallel system and capture complex system features by simple mathematical formulae. These formulae are usually parameterized by a limited number of degrees of freedom so that the analysis is kept tractable. Simulation is a valuable technique which exploits computer resources to model and imitate the behavior of a real system in a controlled manner. We compare the pros and cons of each technique for parallel system performance evaluation in terms of the amount of *statistics* that can be collected from the evaluation, the *accuracy* of the results, and the *cost* in terms of computer and human resources expended in the evaluation. The reader is referred to our detailed comparison [73] of these techniques for examples of evaluation exercises undertaken using each technique.

2.2.1 Statistics

Metrics such as speedup, scaled speedup, sizeup, isoefficiency function and experimentally determined serial fraction that reveal only scalar information are much easier to calculate. In fact, the overall execution times of the parallel system parameterized by number of processors and problem sizes would suffice to calculate these metrics. On the other hand, the measurement of overhead functions and lost cycles would need more sophisticated techniques that can extract a considerable amount of information from the execution.

Experimentation can give the overall execution time of an application on a specific hardware platform but this information is not adequate to isolate and quantify parallel system overheads. Two frequently used solutions to alleviating this problem are instrumentation and hardware monitoring. Instrumentation augments the application code or the system software to accumulate statistics about the program execution. This augmentation may be performed either by the application programmer by hand, or may be relegated to a pre-processor or even the compiler. A detailed analysis of the execution time would require a considerable amount of instrumentation of the application code. In some cases, such detailed instrumentation may itself become intrusive, yielding inaccurate results. Also, it is difficult to capture hardware artifacts such as network traffic and cache actions by a simple instrumentation of the application [74]. Hardware monitoring tries to remedy the latter deficiency by enlisting hardware support to accumulate these statistics. But this support may not be available uniformly across all hardware platforms. Further, hardware monitoring alone cannot give sufficient information about algorithmic and software interaction overheads. A combination of application instrumentation and hardware monitoring [19] may be used to remedy some of these problems. But the strategy would still suffer from the intrusive nature of instrumentation in interfering with both the algorithmic and hardware monitoring mechanisms. Finally, apart from simply being able to study the impact of processors and application size, we would also like the flexibility to investigate the impact of other system parameters, such as the CPU clock speed and network bandwidth, on scalability. Since most of the parameters of the hardware are fixed, performance measurement by experimentation does not provide this flexibility.

Analytical models can directly present statistics for system overheads that are modeled. The values for the hardware and the workload parameters can be plugged into the model corresponding to the system overhead being studied. With models available for each system overhead, we can predict the overall execution time. The drawback is that each system overhead needs to be modeled or ignored in calculating the execution time. Since the metric being studied is expressed as a function of all the relevant system parameters, this technique can be very useful when accurate mathematical models can be developed to capture the interaction between the application and the architecture.

Simulation provides a convenient monitoring environment for observing details of parallel system execution, allowing the user to accumulate a range of statistics about the application, the hardware, and the interaction between the two. In addition, a simulator can supply these metrics for different time windows in execution which can give useful feedback. Since the application and hardware details are modeled in software, the system parameters can be varied and the system simulated again to give the desired metrics. These results may be used to give regression performance models as a function of the system parameters.

2.2.2 Accuracy

The results from any evaluation should be accurate and credible. Intrinsic and external factors determine the accuracy of an evaluation. To a large extent, the accuracy depends on intrinsic factors such as the closeness of the chosen input models to the actual system. The accuracy may also depend on external factors such as monitoring and instrumentation. Even the act of measurement may sometimes perturb the accuracy of the evaluation.

Experimentation can use real applications to conduct the evaluation on actual machines giving accurate and credible results with respect to factors intrinsic to the system. But, as we observed earlier, it may be necessary to instrument the application in order to obtain the desired statistics. Such external intrusion can perturb the accuracy of results.

The accuracy of analytical models depends on the simplifying assumptions made about the hardware and the application details to keep the models tractable. A static analysis of application code as done by these models may not capture dynamic system interactions and data-dependent executions. Real applications often display a dynamic computation and communication behavior that may not be pre-determined [70]. The results from the worst-case and average-case analysis used with these models can thus vary significantly from the real execution. While these models present convenient parameterized functions to capture the asymptotic behavior of system scalability, it is difficult to calculate the constants associated with these functions. Such constants may not prove important for an asymptotic analysis but can become crucial when evaluating real parallel systems of finite size.

Simulation provides a controlled execution capability by which the details of the execution can be gleaned in a non-intrusive manner. Thus, the accuracy of results depends purely on the accuracy of input models. Execution-driven simulation provides the capability to faithfully simulate all the details of a real-world application. We can also simulate all the details of the hardware, though in many circumstances a level of abstraction may be chosen to give sufficiently accurate results for the intended purposes. The accuracy of these abstractions may also be validated by comparing the results with those obtained from a detailed simulation of the machine or an experimental evaluation on the actual machine.

2.2.3 Cost

The cost in terms of computer and human resources is expended in two phases of the evaluation process. The initial cost is expended in developing the input models and the subsequent cost in conducting the evaluation itself. A slight change of system parameters, which do not alter the input model, would demand only an evaluation cost. On the other hand, more drastic changes to the application and/or the hardware would also incur the cost of re-designing the input models.

With experimentation, the cost at the application end is the effort expended in developing algorithms for the given application, arriving at a suitable implementation which is optimized for the given hardware, and debugging the resulting code. This cost is directly dependent on the complexity of the application and would be incurred in any case since the application would ultimately need to be implemented for the given hardware. Further, this cost can be amortized in conducting the evaluations over a range of hardware platforms and a range of application parameters with little modifications to the application code. At the hardware end, a substantial initial cost is incurred in building the machine itself and getting it to work. The cost for conducting the actual evaluation is the execution time of the given application, which may be considered reasonable since the execution is on the native parallel hardware.

Computationally, analytical modeling is appealing since it involves calculating relatively simple mathematical functions. On the other hand, a substantial effort is expended in the development of models. Development of the model involves identifying a set of application and hardware parameters that are important from the performance viewpoint and which can be maintained tractable, studying the relationship between these parameters, and expressing these relationships by simple mathematical functions. Repeated evaluations using the same model for different system configurations may amortize this cost, but since the applicability of these models is limited to specific hardware and application characteristics, repeated usage of the same models may not always be possible.

There is a cost associated with developing models for simulation but this is usually dominated by the cost of the simulation itself. Execution-driven simulations of real parallel systems demand considerable computational resources in terms of space and time. Several studies have tried to address this problem. Abstracting application and hardware details in formulating the simulation model [77, 14, 50, 27] is one approach used to ease resource requirements. Another way of alleviating the cost is by parallelizing the simulation itself like the Wisconsin Wind Tunnel [62] approach which uses the CM-5 hardware for simulating shared memory multiprocessors, and the Timepatch [68] technique for parallel simulation of multiprocessor caches.

2.3 Our Approach

Table 1 summarizes the qualitative comparison between the evaluation techniques in terms of the three figures of merit. Each technique has an important role to play in the performance evaluation of parallel systems. Our approach uses a combination of the three techniques while avoiding some of their individual drawbacks. We would like to benefit from: 1) the realism and accuracy of experimentation in evaluating large parallel systems; 2) the convenience and power of analytical models in predicting the performance and scalability of the system as a function of system parameters; and 3) the flexibility and detailed statistics provided by execution-driven simulators. Our approach is outlined in Figure 2. Experimentation is used to implement real-world applications on actual machines, to understand their behavior and to extract interesting kernels (abstractions of applications that capture the computationally interesting phases of the execution) that occur in them. These kernels are fed to an execution-driven simulator called SPASM which faithfully models the dynamics of the parallel system interactions. The statistics that are drawn from the simulation may be used to develop new analytical models or to validate and refine existing models. The simulation numbers may also be used to calculate the constants associated with the functions in the models, and the combined results from the simulation and the analytical models can be used to project the scalability of large parallel systems as a function of system parameters. Sometimes if an analytical model is sufficiently validated/refined, it may be possible to use this model in the simulator itself to abstract some details of the simulation to ease resource requirements. The validity of such a simulation model can in turn be verified by comparing the simulation results with

Table 1: Comparison of Evaluation Techniques

	Experimentation	Analytical/Theoretical Modeling	Simulation
Statistics	Overall execution time is the only readily available statistic. Additional statistics may be obtained by hardware monitoring and instrumentation. Hardware monitoring relies on support from the underlying hardware and instrumentation can become intrusive. Many underlying system parameters are also fixed.	Directly present statistics for the system overheads that are modeled as a function of the system parameters. Drawback is that each system overhead needs to be modeled or ignored.	Can give a range of statistics about the system as well as detailed performance profiles for different windows in application execution.
Accuracy	Since real-world applications are implemented on actual hardware, there is no accuracy or realism lost in the choice of input models. On the other hand, instrumentation of the application code to obtain detailed statistics can perturb the accuracy of results.	Accuracy is often lost due to the simplifying assumptions made in choosing the abstractions for the application and the hardware. Further, static analysis may not accurately model the dynamic computation and communication behavior of many real applications.	Owing to the controlled execution capability, simulation can faithfully model the details of a real-world application on the specified hardware platform giving accurate results.
Cost/Effort	Cost of actual evaluation is reasonable since the execution is on the native parallel hardware. The cost in developing the applications would be incurred in any case since the application would ultimately need to be implemented for the given hardware.	Actual evaluation is cheap since it involves calculating relatively simple mathematical functions. But, a considerable cost may be expended in developing the models.	Simulations of real parallel systems demand substantial resource usage in terms of time and space. There is also a cost incurred in developing simulation models, but this cost is usually over-shadowed by the cost of actual simulation.

those from an experimental evaluation on the actual hardware. This idea is explored further in Chapter 5.

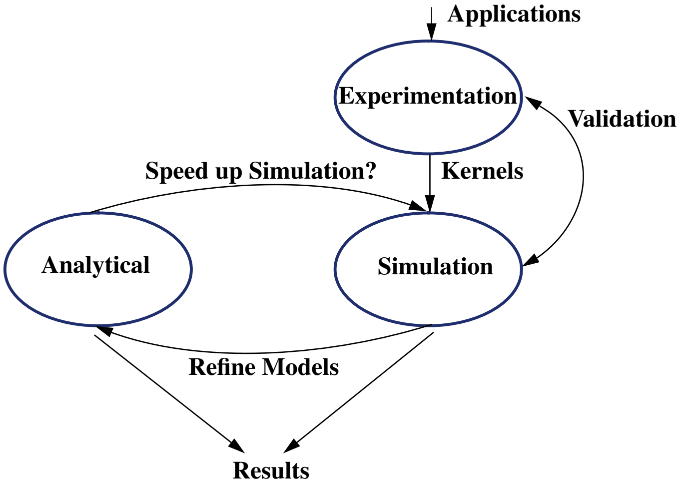


Figure 2: The Framework

In this chapter, we have proposed a set of metrics called overhead functions which provide detailed statistics on parallel system execution. We have compared the relative benefits of three well-known performance evaluation techniques and outlined an approach which uses a combination of the three techniques. At the heart of our evaluation framework lies a simulation platform called SPASM which is used to identify, isolate and quantify the overhead functions. The next chapter describes this simulation platform. An overview of the experimental platform (the architectural platform and characteristics of a suite of applications) which is used in the subsequent evaluation studies of this thesis is also presented in the next chapter.

Chapter 3

SPASM

Men have become the tools of their tools.

- Henry Thoreau

The best advice to those about to embark on a very large simulation is often the same as Punch's famous advice to those about to marry: Don't!

- Brately, Fox, and Schrage

In this chapter, we present details on SPASM (Simulator for Parallel Architectural Scalability Measurements), an important component of our evaluation framework, which helps us isolate and quantify the parallel system overheads.

3.1 Overview

SPASM is an execution-driven simulator written in CSIM [52] used for simulating the execution of a parallel program on a parallel machine. As with other recent simulators [13, 17, 26, 87, 62], the bulk of the instructions in the parallel program is executed at the speed of the native processor (SPARC in most of our studies) and only instructions such as LOADs/STOREs on a shared memory platform, and SENDs/RECEIVEs on a message passing platform, that may potentially involve a network access are simulated. The rationale behind this approach is that since uniprocessor architecture is getting standardized with the advent of RISC technology, we can fix most of the processor characteristics (such as instruction sets, clocks per instruction, floating point capabilities, pipelining) by using a commodity processor as the baseline for each processor in our parallel system. A detailed simulation of the processor architecture is not likely to contribute significantly to our understanding of the scalability of a parallel system. The input to the simulator is parallel applications written in C. On a message passing system, the calls (SENDs/RECEIVEs) which trap to the simulator are inserted into the application program explicitly by the programmer. On a shared memory system, a pre-processor inserts code into the application program to trap to the simulator on a shared memory reference. On both systems, the compiled assembly code is augmented with cycle counting instructions which is used to keep track of the time spent in the application program since the last trap to the simulator. Finally, the assembled binary is linked with the rest of the simulator code.

A simulation platform like SPASM allows us to vary a wide range of hardware parameters such as the number of processors, the CPU clock speed, the network topology, the bandwidth of the links in the network, the network switching delays, and the cache parameters (the block size, cache size, associativity, etc). We have used SPASM to study both message passing [76] and shared memory platforms [75, 77, 78, 61]. Within the scope of shared memory platforms, we have used SPASM to study different network topologies [75, 77], memory consistency models and cache coherence protocols [61]. We confine ourselves to the shared memory platform in this thesis and in the rest of this chapter we present the details of SPASM relevant to shared memory platforms.

3.2 Statistics

Each simulated processor in SPASM passes through a set of system-defined and user-defined modes while executing a shared memory parallel program. Detailed statistics are collected by SPASM for each processor individually for each of these modes and the results are presented for a representative processor specified by the user. These statistics are used to compute the overhead functions. The system-defined modes provided by SPASM are:

- BARRIER: Mode corresponding to a barrier synchronization operation. SPASM provides a tournament barrier [51] as a library routine to support this operation.

- MUTEX: Mode corresponding to a mutual exclusion lock. The library routine provided by SPASM implements this using a test-test&set [9] operation.

- 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.

- NORMAL: Mode corresponding to the actual work done in the application. A program is in the NORMAL mode if it is not in any of the other modes. A user may choose to specify user-defined modes within this mode to get detailed statistics for specific regions of the application. Such a feature can narrow down bottlenecks in the execution and help in application restructuring.

For each executing processor, SPASM accumulates the following statistics for each mode of execution:

- *computation time*: This is the time taken to execute on an ideal machine such as the PRAM [92].

- *latency*: Accesses to variables in a shared memory system may involve the network, and the physical limitations of the network tend to contribute to overheads in the execution. The latency overhead is 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.

- *contention*: The contention overhead is the total amount of time incurred by a processor due to the time spent waiting for availability of network resources by its messages.

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. 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 for identifying network bottlenecks and comparing relative merits of different networks and their capabilities.

Since the focus of our subsequent studies is primarily on hardware interaction overheads, particularly the communication overhead, we do not address software interaction overheads such as scheduling[1] in this thesis. However, the modularity in the implementation of SPASM makes it easy to investigate these other overheads if needed as is illustrated in a related study [76] where we use it to model software message passing overheads.

The *total time* (simulated time) for a given application is the maximum of the *execution 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 execution time for a processor is the sum of its execution times over all the modes. And the execution time for a processor in any particular mode is the sum of the computation time, latency and contention overheads. *Speedup* using p processors is measured as the ratio of the total time on 1 processor to the total time on p processors ("delivered" curve in Figure 1).

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

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

and the resulting speedup gives the "achievable" curve in Figure 1. As we mentioned earlier, the difference between the ideal time and the execution time for a linear speedup gives the algorithmic overhead, and the difference between the total time and the ideal time gives the interaction overhead (see Figure 1).

Table 2: SPASM Statistics

	Comp. Time	Latency	Contention
NORMAL	Useful Work	Data Access	Data Access
BARRIER	Work Imbalance	Sync. Access	Sync. Access
PGM_SYNC	Work Imbalance	Sync. Access	Sync. Access
MUTEX	Serial Part	Sync. Access	Sync. Access

The computation time, latency and contention overheads capture interesting facets in each mode of execution (Table 3.2). Computation time in the NORMAL mode is the time spent in local computation which is a measure of the actual work in the application. The latency and contention overheads in the NORMAL mode reflect the network overheads for ordinary data accesses which is determined by the data access pattern of the application. For the BARRIER and PGM_SYNC modes, the computation time is the time incurred by a processor waiting for one or more processors to arrive at the synchronization point as a consequence of the algorithmic work imbalance. The computation time in the MUTEX mode is the time spent waiting to acquire a lock and represents the serial part in an application arising due to critical sections. For the BARRIER and MUTEX modes, the computation time also includes the cost of implementing the synchronization primitive and other residual effects due to latency and contention for prior accesses. In all three synchronization modes, the latency and contention overheads together represent the network overheads incurred in accessing synchronization variables. The metrics identified by SPASM thus quantify the interesting components of the algorithmic and interaction overheads.

3.3 Application and Hardware Characteristics

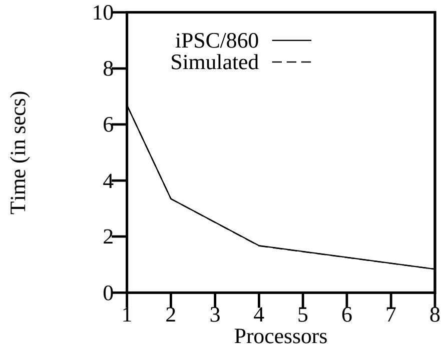
In each of the subsequent studies in this thesis, we use a set of applications with diverse characteristics and simulate their execution on SPASM. Three of the applications (EP, IS and CG) are from the NAS parallel benchmark suite [11]; CHOLESKY is from the SPLASH benchmark suite [71]; and FFT is the 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 certain phases of 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 on these applications.

Even though the studies may differ in the choice of simulated hardware platforms, there are certain common traits across the platforms. Whenever a chosen platform does not conform to these specifics, the details will be explicitly mentioned. As we noted earlier, this thesis focuses on shared memory platforms particularly Cache Coherent Non-Uniform Memory Access (CC-NUMA) machines. 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 at each processing node is a 2-way set-associative cache (64KBytes with 32 byte blocks) that is maintained sequentially consistent using an invalidation-based (Berkeley protocol) full-mapped directory-based cache coherence scheme. We use a SPARC chip as the baseline for fixing the processor characteristics. For the interconnection network, we choose three different network topologies (a *fully connected network*, a *binary hypercube* and a *2-D mesh*) from the connectivity spectrum. All three networks use serial (1-bit wide) unidirectional links but the bandwidth of these links can be varied. 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 and uses the e-cube algorithm [80] for routing messages. 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). A wormhole routing strategy is employed for the messages and the message-size can vary upto 32 bytes (with an 8 byte header). The switching delay is assumed to be negligible compared to the transmission time and is ignored.

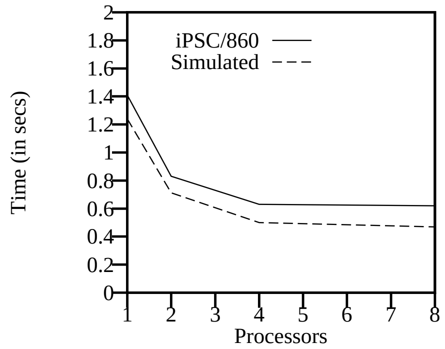
3.4 Validation

To validate our simulation model, we compare the results that we get from SPASM to the results obtained from implementing the applications on the actual machine. There are two main drawbacks with such an approach. First, we can only verify the overall execution time and not the individual parallel system overheads owing to the limited statistics obtainable from experimentation on the machine (see Section 2.2.1). Second, we can only validate results for a simulated machine when it exactly matches a real machine. Due to the lack of a better solution and to get a reasonable amount of confidence in our evaluation results, we verify the results we obtain from an experimental evaluation on the Intel iPSC/860 [38] and the KSR-1 [64] with results from comparable message passing and shared memory models on SPASM. Figures

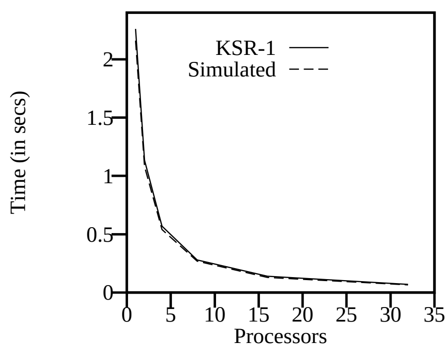
3 (a), (b), (c) and (d) show that our simulation model closely captures the scalability trends of EP (an application with negligible overheads) and IS (an application with fairly high algorithmic and interaction overheads for a small problem size of 64K chosen in these experiments). These figures reflect the general behavior observed over the chosen applications.



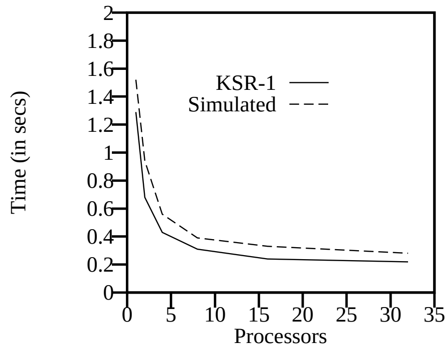
(a) EP (64K problem) on iPSC/860



(b) IS (64K problem) on iPSC/860



(c) EP (64K problem) on KSR-1



(d) IS (64K problem) on KSR-1

Figure 3: Validation Experiments

In the rest of this thesis, we illustrate the power of our framework in answering some of the questions raised in Chapter 1. In each of the subsequent chapters, we state the specific problem that we are trying to solve, give an overview of our evaluation methodology, and present interesting results from the evaluation.

Chapter 4

Scalability Studies of Parallel Systems

I examined aspects of scalability, but did not find a useful, rigorous definition of it. Without such a definition, I assert that calling a system "scalable" is about as useful as calling it "modern". I encourage the technical community to either rigorously define scalability or stop using it to describe systems.

- Mark Hill

4.1 The Problem

In this chapter, we illustrate the power of our framework in studying the scalability of parallel systems. In particular, we address the questions related to understanding the performance of specific application-architecture pairs, identifying application/architectural bottlenecks, and using this information to improve performance by application re-structuring and architectural enhancements. An understanding of the interaction between application-architecture pairs is also used to project the scalability of larger systems.

4.2 The Approach

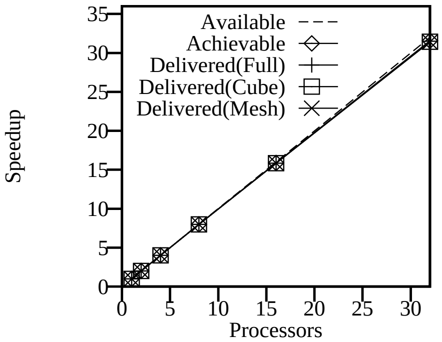
It is often useful to find out how a system would scale as a function of different system parameters such as the number of processors, application problem size, CPU clock speed, and network bandwidth. SPASM allows us to vary these different parameters and study their impact on overall performance. Since our focus is on illustrating an approach to scalability study, we restrict ourselves to the case where only the number of processors (p) in the machine can vary while all other system parameters are fixed.

We illustrate our methodology by studying the scalability of the five applications EP, IS, FFT, CG and CHOLESKY on the CC-NUMA hardware platform outlined in section 3.3 with the fully connected, mesh and hypercube network topologies. We fix the clocks of the processors in our simulated machine at 33 MHz and the bandwidth of the links in the network at 20 MBytes/sec. For the applications, we use a problem size of 64K for EP, IS and FFT, a sparse matrix of size 1400X1400 containing 100,300 non-zeroes for CG, and a 1806-by-1806 matrix with 30,824 non-zeros for CHOLESKY. We also briefly discuss the impact of larger problem sizes.

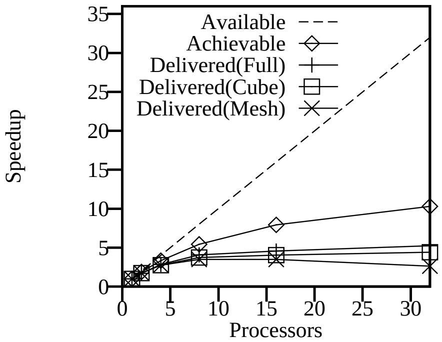
Figures 4 (a), (b), (c), (d) and (e) show the “achievable" and “delivered” speedup curves (section 2.1) for EP, IS, FFT, CG and CHOLESKY respectively on the three hardware platforms. For IS and CHOLESKY, the achievable curve shows a significant deviation from linear speedup indicating a considerable algorithmic overhead in these applications. When we look at the deviation of the delivered curve from the achievable curve, there is negligible deviation for EP on the three hardware platforms; a marginal deviation for FFT and CG; and a significant deviation for IS and CHOLESKY. In the following sections, we examine the performance of each application in detail and quantify the different overheads limiting their scalability.

4.3 Performance Results

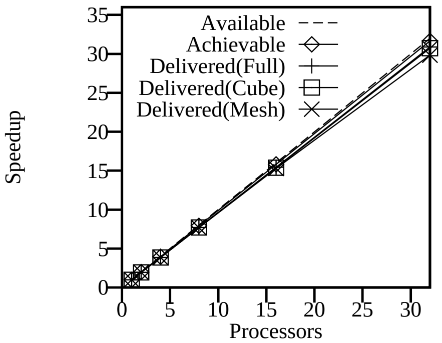
For each application, we show the execution time, the latency, and the contention overhead graphs for the mesh platform since the interaction overhead is the maximum on this topology. The first shows the total execution time, while the latter two show the communication overheads ignoring the computation time. In each of these graphs, we show the curves for the individual modes of execution. We also present the latency overhead and contention overhead curves for the three network topologies. Owing to wormhole routing and relatively small number of processors in the system, the latency incurred by a single message is relatively independent of the network topology. The number of messages in the NORMAL mode (i.e. due to ordinary data access) is



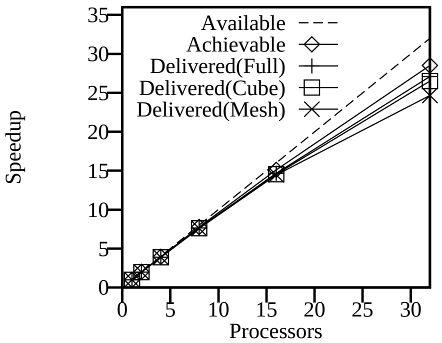
(a) EP



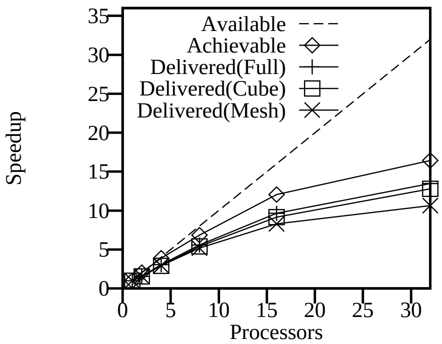
(b) IS



(c) FFT



(d) CG



(e) CHOLESKY

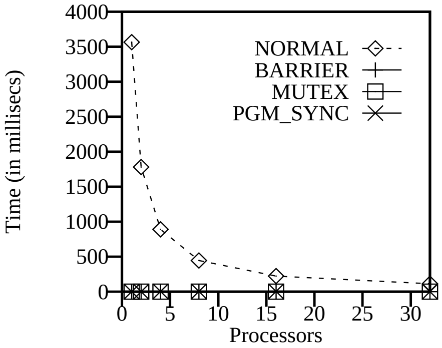
Figure 4: Speedup

determined by the memory reference pattern of the application as well as cache line replacements due to capacity misses. Discounting data dependent executions, the number of messages in the NORMAL mode is also expected to be independent of network topology. Due to the vagaries of the synchronization accesses, it is conceivable that the number of messages could differ across network platforms for the other modes. However, in our experiments we have not seen any significant deviation. As a result, the latency overhead curves for all the applications look alike across network platforms and we present only one latency curve. On the other hand, it is to be expected that the contention overhead will increase as the connectivity in the network decreases.

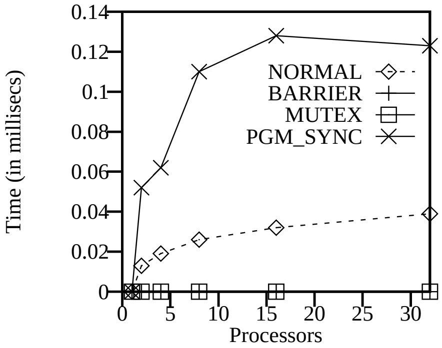
EP

The communication in EP is restricted to calculating a few global sums and the time spent in these operations is insignificant compared to the local computation time (see the Appendix). Figures 5 (b) and (c) show the growth of the latency and contention overheads respectively for the different modes of execution of EP. But these overheads are negligible compared to the total execution time which is dominated by the NORMAL mode (Figure 5 (a)).

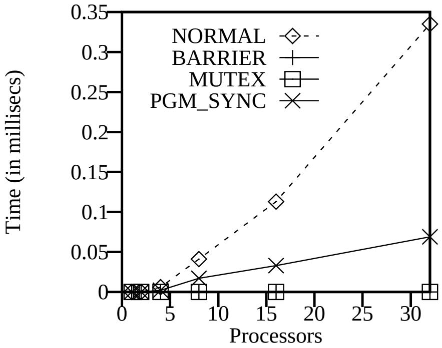
Figure 5 (d) shows the latency overhead and contention overheads for the three network topologies. Since the number of communication events for global sums is logarithmic in the number of processors, the latency overhead curve exhibits a logarithmic behavior. Although there is a potential for contention overhead for the fully connected network due to message queuing at a node, the observed contention is marginal. With less connectivity, the contention overhead even dominates the latency overhead with increasing number of processors. For instance, the cross-over point between contention and latency occurs at around 25 processors for the cube and at around 16 processors for the mesh. Table 3 summarizes the overhead functions obtained by regression analysis of the datapoints from our simulation.



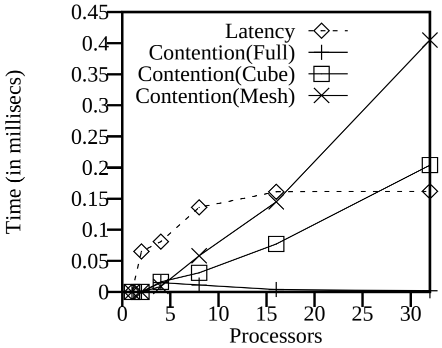
(a) Mode-wise Execn. Time (Mesh)



(b) Mode-wise Latency (Mesh)



(c) Mode-wise Contention (Mesh)



(d) Latency and Contention

Figure 5: Scalability of EP

Table 3: EP : Overhead Functions

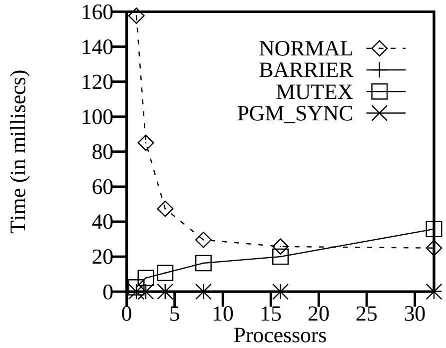
EP	Full	Cube	Mesh
Comp. Time (s)	$3.6/p$	$3.6/p$	$3.6/p$
Latency (us)	$140.4 \log p$	$140.4 \log p$	$140.4 \log p$
Contention (us)	$Negligible$	$6.6p$	$13.7p$

As can be seen from Table 3, the constant associated with the computation time is much larger than those associated with latency and contention. The computation time goes down linearly with the number of processors showing there is negligible algorithmic overhead. EP is thus perfectly scalable on all three hardware platforms. Even for a relatively small problem size (64K in this case), it would take nearly 1000 processors for the overheads to start dominating. While an increase in the problem size would increase the coefficient associated with the computation time, it does not have any effect on the coefficients of the other overhead functions. Hence, the scalability of EP for a larger problem size can only get better.

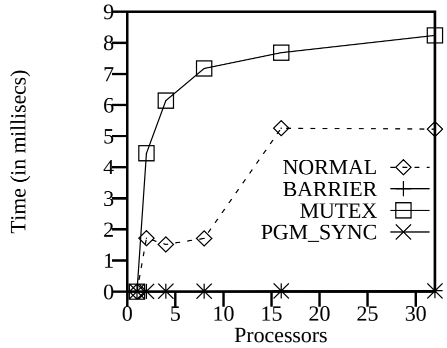
IS

Parallelization of IS increases the total amount of work to be done across all processors because there is a phase in the execution where the work done by a processor does not decrease when the number of processors is decreased (see the Appendix). For small problem sizes such as the 64K problem chosen in this exercise, this inherent algorithmic overhead causes a significant deviation of the achievable curve from the linear speedup curve (see Figure 4 (b)). The capability of SPASM in providing statistics for different segments of program execution by identifying user-defined modes helps in identifying where the algorithmic overhead is incurred. This information can provide useful hints to the programmer suggesting alternate implementations.

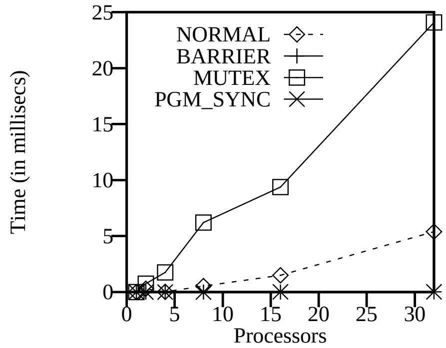
There is also a significant interaction overhead component for IS which may be analyzed by considering the different modes of execution. NORMAL and MUTEX



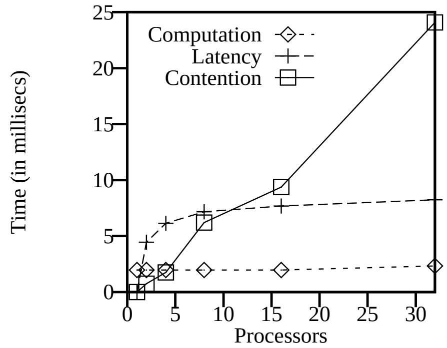
(a) Mode-wise Execn. Time (Mesh)



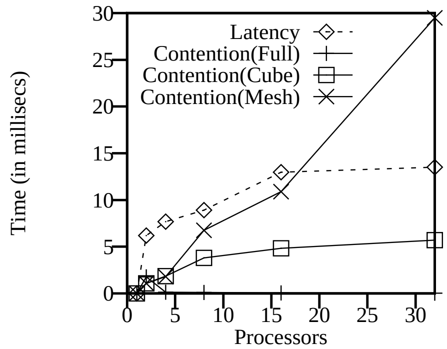
(b) Mode-wise Latency (Mesh)



(c) Mode-wise Contention (Mesh)



(d) MUTEX mode (Mesh)



(e) Latency and Contention

Figure 6: Scalability of IS

are the only significant modes of execution (see Figure 6 (a)) in IS. The latency and contention overheads incurred in the MUTEX mode for accessing lock variables is higher than the corresponding components in the NORMAL mode for accessing ordinary data (see Figures 6 (b) and (c)). As a result of this, the total execution time in the MUTEX mode surpasses that in the NORMAL mode beyond 20 processors (Figure 6 (a)), which also explains the dip in the speedup curve for mesh (Figure 4 (b)). Since the problem rests within the MUTEX mode, we can focus in on this mode and analyze the execution (Figure 6 (d)). The computation time in this mode which reflects the time spent spinning to acquire a lock (implemented as a test-test&set) remains constant with increasing processors. To the application programmer, this indicates that the serial portion of the algorithm is not a bottleneck. To the architect, it indicates that the contention for a lock does not increase with the processors. On the other hand, the contention in the network is the main problem implying that the architect should improve the network bandwidth to get maximum rewards.

Figure 6 (e) shows the latency overhead and contention overheads for the three network topologies. In IS, since every processor needs to access data from all other processors, and since the data is equally partitioned among the executing processors, the number of accesses to remote locations grows as $(p-1)/p$ (see the Appendix). For large p, the latency thus converges to a constant explaining the flattening of the latency overhead curve. On the mesh network, the contention overhead is a severe problem surpassing the latency overhead at around 18 processors. Table 4 summarizes the overheads for IS obtained by regression analysis of the datapoints from our simulation.

As we mentioned, the algorithmic overhead in parallelizing IS is responsible for a computation time which does not decrease linearly with the number of processors (Table 4). Table 4 shows that the contention overhead is negligible and the latency overhead converges to a constant with a sufficiently large number of processors on a fully connected network. Thus for a fully connected network, the scalability of IS is

Table 4: IS : Overhead Functions

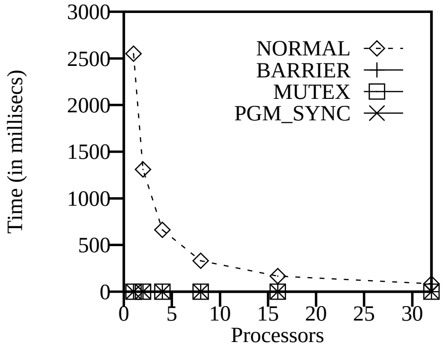
IS	Full	Cube	Mesh
Comp. Time (ms)	$129.3/p^{0.7}$	$129.3/p^{0.7}$	$129.3/p^{0.7}$
Latency (ms)	$13.2(1-1/p)$	$13.2(1-1/p)$	$13.2(1-1/p)$
Contention (ms)	*Negligible*	$4.0\log p$	$0.9p$

expected to closely follow the achievable curve. For the cube and mesh platforms, the contention overhead grows logarithmically and linearly with the number of processors, respectively. Therefore, the scalability of IS on these two platforms is likely to be worse than for the fully connected network. From the above observations, we can conclude that IS is not very scalable for the chosen problem size on the three hardware platforms. However, if the problem is scaled up, the coefficient associated with the computation time will increase thus making IS more scalable.

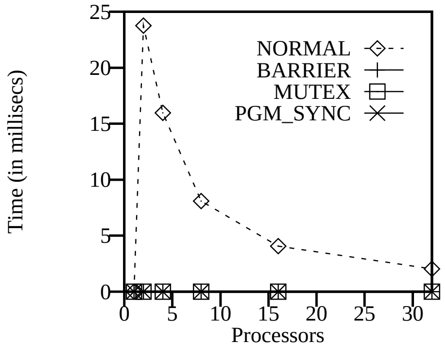
FFT

The algorithmic and interaction overheads for FFT are marginal. Thus the delivered curves for all three platforms as well as the achievable curve are close to the linear one as shown in Figure 4 (c). The execution time is dominated by the NORMAL mode (Figure 7 (a)). The latency and contention overheads (Figures 7 (b) and (c)) incurred in this mode are insignificant compared to the total execution time for upto 32 processors.

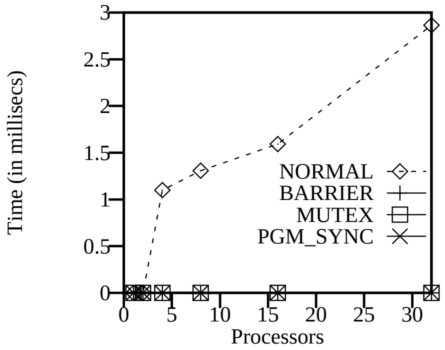
The communication in FFT has been optimized as suggested in [20] into a single phase where every processor accesses the data of all the other processors in a skewed manner. The number of such non-local accesses incurred by a processor grows as $O((p-1)/p^2)$ with the number of processors, and the latency overhead curve reflects this behavior. As a result of skewing the communication among the processors, the contention is negligible on the full and the cube platforms (Figure 7 (d)). On the mesh the contention surpasses the latency overhead at around 28 processors. Table 5



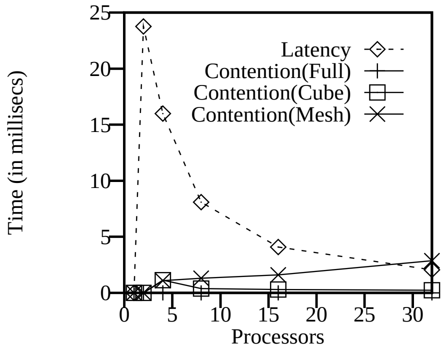
(a) Mode-wise Execn. Time (Mesh)



(b) Mode-wise Latency (Mesh)



(c) Mode-wise Contention (Mesh)



(d) Latency and Contention

Figure 7: Scalability of FFT

summarizes the overheads for FFT obtained by regression analysis of the datapoints from our simulation.

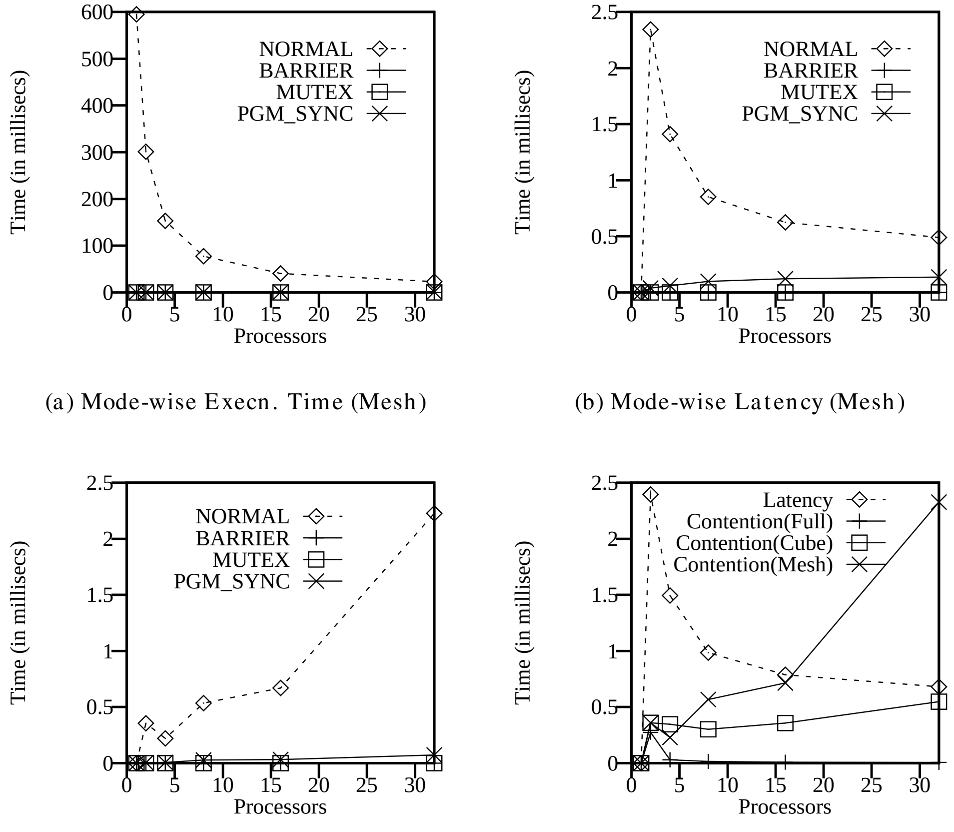
Table 5: FFT : Overhead Functions

FFT	Full	Cube	Mesh
Comp. Time (s)	$2.5/p$	$2.5/p$	$2.5/p$
Latency (ms)	$49.9/p^{0.9}$	$49.9/p^{0.9}$	$49.9/p^{0.9}$
Contention (us)	*Negligible*	*Small*	$63.5p$

A linear decrease in the computation time with the number of processors indicates that the algorithmic overhead is marginal. Of the interaction overheads, the latency component does not play an important role since the number of network accesses incurred by a processor decreases as the number of processors is increased for the same problem size. The contention overhead is thus the only artifact that can limit the scalability of FFT. But with skewed communication accesses, the contention overhead has also been minimized and begins to show only on the mesh network where it grows linearly (see Table 5). Thus we can conclude that FFT is scalable for the fully-connected and cube platforms. For the mesh platform, it would take 200 processors before the contention overhead starts dominating for the 64K problem size. With increase in problem size (n), the local computation that performs a radix-2 Butterfly is expected to grow as $O((n \log n)/p)$ while the communication for a processor is expected to grow as $O(n(p-1)/p^2)$. Hence, increase in data size which increases the computation to communication ratio will improve its scalability on all hardware platforms.

CG

The algorithmic and interaction overheads for CG are larger than the corresponding overheads for EP, but not as dominant as in IS (Figure 4 (d). The NORMAL mode is the only dominant mode of execution as depicted in Figure 8 (a). The communication in the NORMAL mode for data accesses incurred in calculating the matrix-vector product (the main part of the application) outweighs the overhead in accesses for synchronization variables during the BARRIER and PGM_SYNC modes (Figures 8 (b) and (c)). But the communication is still insignificant compared to the overall execution time for the range of processors considered.



(a) Mode-wise Execn. Time (Mesh)

(b) Mode-wise Latency (Mesh)

(c) Mode-wise Contention (Mesh)

(d) Latency and Contention

Figure 8: Scalability of CG

The fewer the rows assigned to a processor, the fewer will be the number of elements of the vector that may need to be accessed for the matrix-vector product (see the Appendix). Therefore, as the number of processors increases, the number of rows of the sparse matrix allocated to a processor decreases, thereby decreasing the likelihood of non-local memory references. Hence, the latency overhead decreases with an increase in the number of processors. The contention overhead increases from the full to the cube topology and surpasses the latency overhead for the mesh (Figure 8 (d)) at around 17 processors. Table 6 summarizes the overheads for CG.

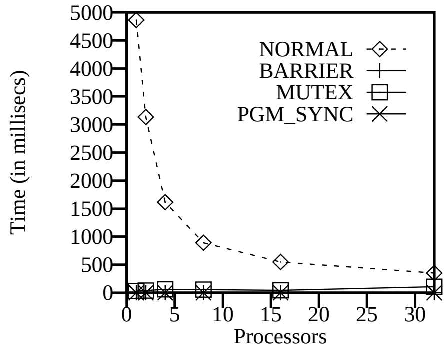
Table 6: CG : Overhead Functions

CG	Full	Cube	Mesh
Comp. Time (ms)	$571.2/p$	$571.2/p$	$571.2/p$
Latency (ms)	$2.9/p^{0.4}$	$2.9/p^{0.4}$	$2.9/p^{0.4}$
Contention (us)	$Negligible$	$8.1p$	$68.2p$

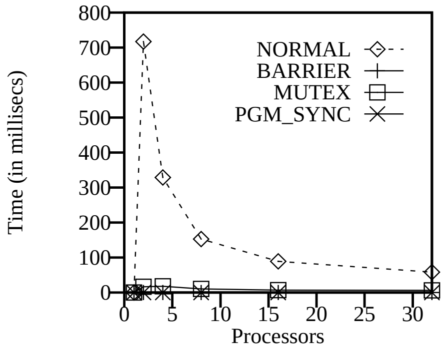
The algorithmic overhead for CG is marginal. Of the interaction overheads, the latency overhead decreases with increasing number of processors and the contention overhead is more pronounced. The contention overhead is negligible for the fully-connected network, grows linearly for the cube and the mesh with a larger coefficient for the mesh compared to the cube. CG is thus scalable for the fully-connected network and becomes less scalable for networks with lower connectivity like the cube and the mesh. Since the NORMAL mode dominates the total execution time, the scalability of the matrix-vector product would determine the scalability of CG. Scaling the problem size increases the number of non-local memory accesses linearly while the increase in local computation would be at least linear if not larger. Thus an increase in problem size is likely to enhance the scalability of CG on all three network platforms.

CHOLESKY

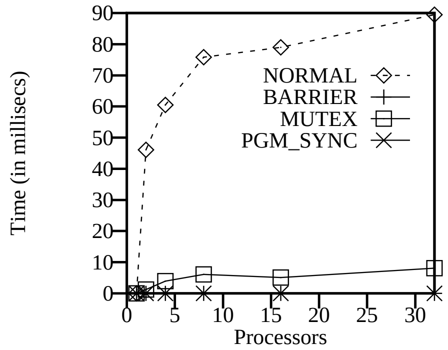
As with IS, the additional work done in the parallelization of CHOLESKY is an important algorithmic overhead in limiting its scalability (Figure 4 (e)). An examination of the execution times (Figure 9 (a)) shows that the bulk of the time is spent in the NORMAL mode which performs the actual sparse matrix factorization. The communication overheads in the NORMAL mode for the data accesses of the sparse matrix outweigh the accesses for synchronization variables (Figures 9 (b) and (c)). Thus the time spent in the MUTEX mode (which represents dynamic scheduling and accesses to critical sections) is insignificant compared to the NORMAL mode. Although, the contention overhead in the NORMAL mode increases quite rapidly with the number of processors, the overall impact of communication on the execution time is insignificant (see Figure 9 (a)) upto 32 processors.



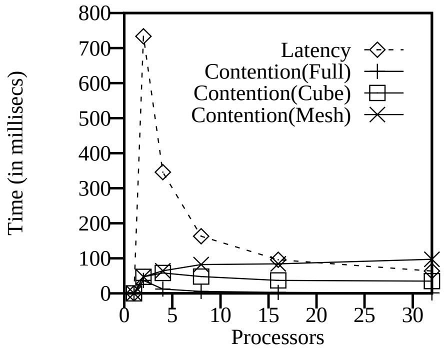
(a) Mode-wise Execn. Time (Mesh)



(b) Mode-wise Latency (Mesh)



(c) Mode-wise Contention (Mesh)



(d) Latency and Contention

Figure 9: Scalability of CHOLESKY

The number of non-local memory accesses made by a processor decreases with increasing number of processors explaining a decreasing latency overhead. The contention overhead is negligible for the fully-connected network and grows with increasing processors for the cube becoming more dominant than the latency overhead for the mesh (Figure 9 (d)) at around 20 processors. Table 7 summarizes the overheads for CHOLESKY.

Table 7: CHOLESKY : Overhead Functions

CHOLESKY	Full	Cube	Mesh
Comp. Time (s)	$3.9/p^{0.8}$	$3.9/p^{0.8}$	$3.9/p^{0.8}$
Latency (s)	$1.2/p^{0.9}$	$1.2/p^{0.9}$	$1.2/p^{0.9}$
Contention (ms)	$Negligible$	$Constant$	$39.9 \log p$

Table 7 shows that the computation time does not decrease linearly with an increase in the number of processors due to the additional work done in parallelization of the application. The latency decreases with increasing number of processors and the interaction overhead would thus be dictated by the contention overhead. The contention on the fully-connected and cube networks is negligible thus projecting speedup curves that closely follow the achievable curve for these platforms. On the other hand, the contention grows logarithmically on the mesh making this platform less scalable. With increasing problem sizes, the coefficient associated with the computation time in the above table is likely to grow faster than the coefficients associated with the communication overheads (verified by experimentation). Hence, an increase in problem size would enhance the scalability of CHOLESKY on all hardware platforms.

4.4 Summary

In this chapter, we have illustrated how our framework can be used to study the scalability of specific application-architecture pairs. Using SPASM, we have analyzed the performance of the five applications on shared memory platforms with three different network topologies. Separation and quantification of parallel system overheads have provided key insights on system scalability. Algorithmic overheads such as the parallelization overhead are dominant in limiting the scalability of IS and CHOLESKY. SPASM's ability to pinpoint the region of execution where these overheads are incurred provides useful hints to the programmer for application re-structuring. For instance, an initial implementation of IS yielded a contention overhead that far exceeds the one shown in Figure 6 (e). SPASM helped narrow down the problem and discover an alternate implementation where the communication accesses between processors were skewed to reduce the overhead to the one shown in Figure 6 (e). Isolation of different components of the interaction overheads give feedback to the system architect for making architectural enhancements. On the architectural side, this exercise shows that in shared memory machines with private caches, as long as the applications are well-structured to exploit locality, the key determinant to scalability is network contention. This is particularly true for most commercial shared memory multiprocessors which have sufficiently large caches.

The investigations in this study provide a detailed understanding of each application and its intrinsic properties which impact its performance on the underlying hardware. This understanding serves as a guidepost in conducting the other evaluations in this thesis.

Chapter 5

Evaluating Machine Abstractions

A little inaccuracy sometimes saves a ton of explanation.

- H. H. Munro

5.1 The Problem

Performance analysis of parallel systems is complex due to the numerous degrees of freedom that they exhibit. 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 [30, 6, 7, 33, 16, 86, 20, 72] 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 [57, 85, 22, 88, 49] abstract complex system interactions with simple mathematical formulae. An abstraction of a parallel system artifact is thus intended to mimic the behavior of its corresponding real world counterpart so that we need not be burdened with the details of the artifact during the evaluation.

Earlier in this thesis, we presented an execution-driven simulator called SPASM which is extremely useful in capturing the dynamic interactions between the application and architecture in a parallel system. However, such execution-driven simulators which model every artifact of a parallel system demand considerable resources both in terms of space and time (see section 2.2.3). A sufficiently abstract simulation model which does not compromise on accuracy can help in easing this problem. Hence, apart from algorithm development and analytical modeling, abstractions may also be useful for speeding up execution-driven simulators. As we mentioned in section 2.3, analytical modeling and simulation co-exist in our evaluation framework towards avoiding some of their individual drawbacks (Figure 10). The datapoints from simulation are used to validate/refine existing models or to even develop new models. The validated models are incorporated back into the simulator to ease resource requirements. We investigate this idea in this chapter. In particular, we investigate to see if the detailed simulation of the *interconnection network* and *locality* properties of a parallel system can be avoided by choosing appropriate abstractions to model these components.

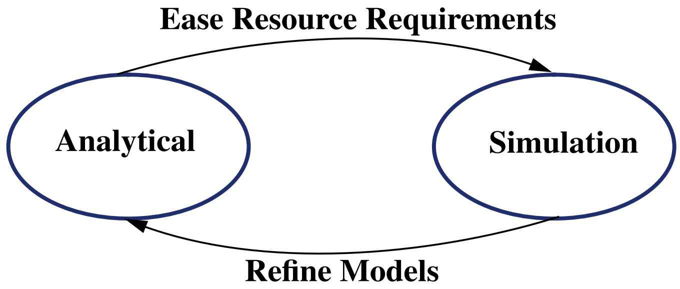


Figure 10: Co-existence of Analytical Modeling and Simulation

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. Focusing only on shared memory multiprocessors, an important question that arises is to what extent can caches be abstracted 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.

5.2 The Approach

We explore the abstraction issues in the context of simulating Cache Coherent Non-Uniform Memory Access (CC-NUMA) shared memory machines employing the same applications as in Chapter 4. For abstracting the interconnection network, we use the recently proposed *LogP* [20] 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. Note that communication locality is subsumed in the abstraction for the interconnection network. Thus in the rest of this chapter (unless explicitly stated otherwise) we use the term 'locality' to simply mean data locality. 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. To evaluate these abstractions, we simulate three machine models namely the *target* machine, the *LogP* machine and the *cLogP* machine on SPASM. The target machine is the CC-NUMA machine with the three network topologies that has been used for the scalability studies in Chapter 4. In this machine model, all the pertinent hardware features of the interconnection network and cache coherence actions are simulated in detail. A description of the other two machine models is given below.

5.2.1 The LogP Machine

The LogP model proposed by Culler et al. [20] 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 parameters for the network:

- 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.

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 is ignored in this study. The 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 in our simulation. 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. This parameter is chosen to be 1.6 microseconds assuming 32 byte messages and a link bandwidth of 20 MBytes/sec. 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. The g parameter would depend on the connectivity of the network and is calculated using the bisection bandwidth available per processor as suggested by Culler et al. [20]. Accordingly, the g parameter for the full, cube and mesh topologies are 3.2/p, 1.6 and 0.8*p_x microseconds respectively (where p is the number of processors and p_x is the number of columns in the mesh).

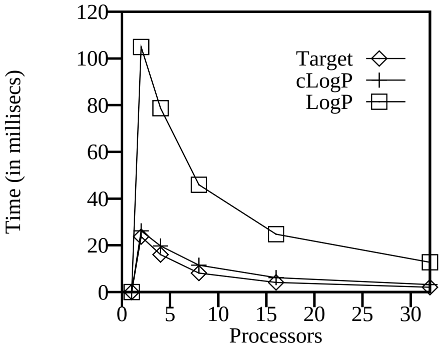
5.2.2 The cLogP machine

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. 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 "target" machine without having to model the details of any specific cache coherence protocol, since it is not our intent 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 "target" 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. Only if the network accesses incurred in the cLogP model are significantly lower than the accesses on the "target" machine, would we need to make our cLogP abstraction more realistic. But as we will soon observe, our results 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 target machine implements a fancier invalidation-based cache coherence protocol (which would reduce the network accesses even further), that would only enhance the agreement between the results for the cLogP and target machines.

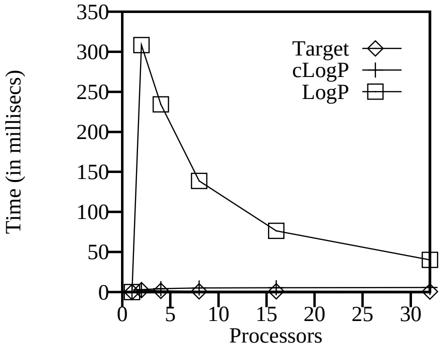
5.3 Performance Results

Validating the Network Abstraction

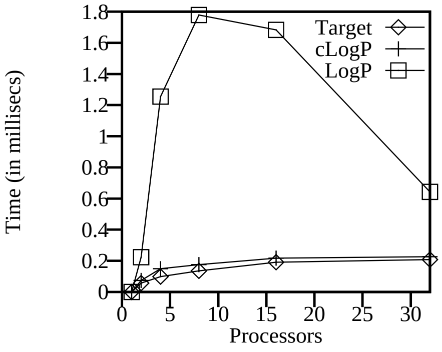
For validating the chosen network abstraction, we compare the network overheads in the cLogP and target machines for the chosen applications. Figures 11 (a), (b), (c), (d) and (e) show that the latency overhead curves for the cLogP machine display a trend (shape of the curve) very similar to the target machine. As we noted in Chapter 4, there is negligible difference in latency overhead across network platforms



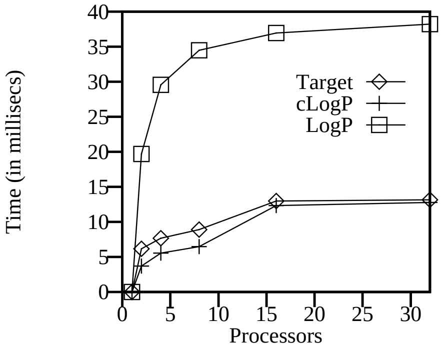
(a) FFT on Full



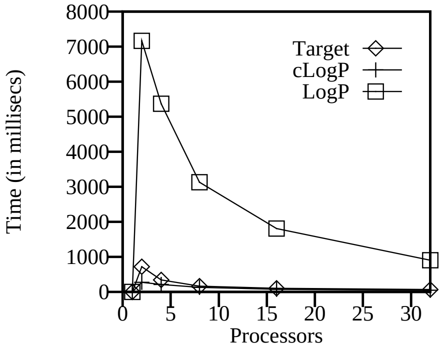
(b) CG on Full



(c) EP on Full



(d) IS on Full

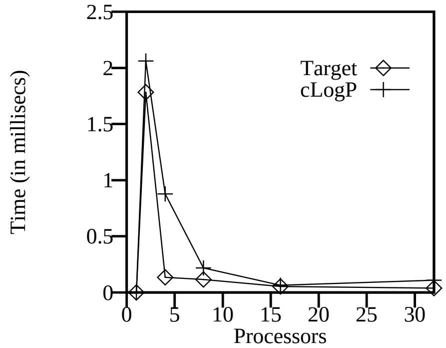


(e) CHOLESKY on Full

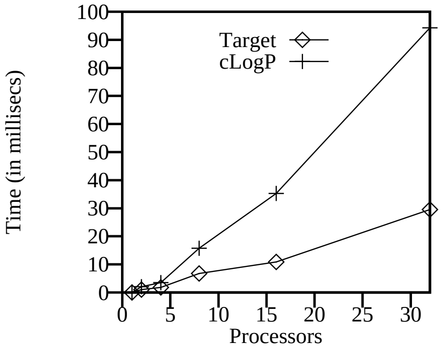
Figure 11: Latency Overhead

for the chosen applications since the size of the messages and the transmission time dominate over the number of hops traversed. Since the 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 target machines. 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 target machine. On the other hand, cLogP does not model coherence traffic thereby incurring fewer network messages than the target machine, which can have the effect of making L more optimistic. 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 target machine for FFT (Figure 11 (a)) and CG (Figure 11 (b)) since there is very little coherence related activity in these two applications; on the other hand, the optimism favors cLogP in IS (Figure 11 (d)) and CHOLESKY (Figure 11 (e)) 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 quite closely models the latency attribute.

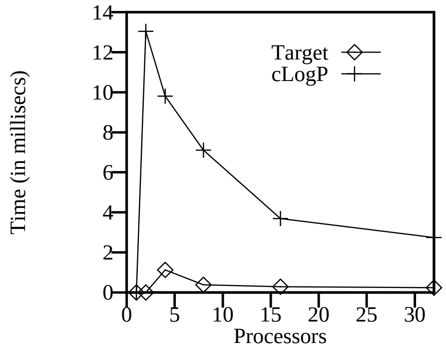
Figures 12 (a), (b), (c), (d), (e) and (f) show the contention overhead curves for the cLogP and the target machines. The g-parameter in cLogP is estimated using the bisection bandwidth of the network as suggested in [20]. Such an estimate assumes that every message in the system traverses the bisection which can become very pessimistic (as reflected in Figure 12) when the application displays sufficient communication locality [2, 3]. This pessimism increases as the connectivity of the network decreases (as can be seen in Figures 12 (a) and (b)) 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 (Figures 12 (e) and (f)). In fact, this amplified effect changes the very trend of the cLogP contention curves compared to the target machine. These results indicate that the contention



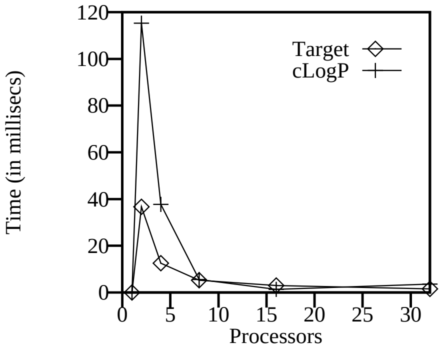
(a) IS on Full



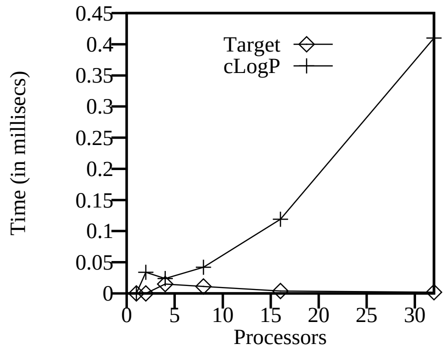
(b) IS on Mesh



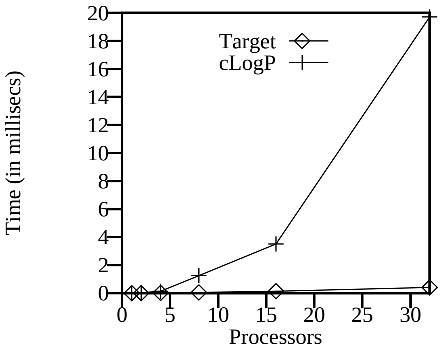
(c) FFT on Cube



(d) CHOLESKY on Full



(e) EP on Full



(f) EP on Mesh

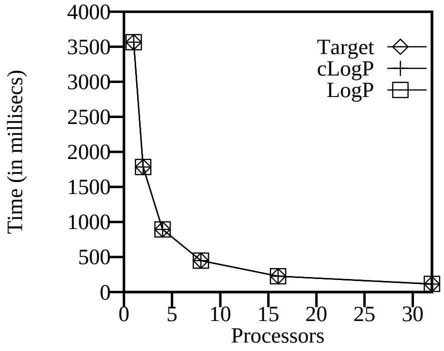
Figure 12: Contention Overhead

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.

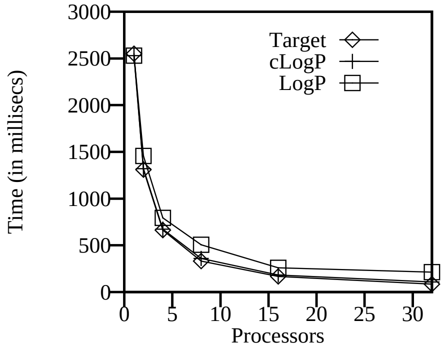
Validating the Locality Abstraction

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 target machines (Figure 13 (a)), regardless of network topology. On the fully connected and cube networks there is little difference in the results for FFT as well, and the difference on the mesh interconnect is shown in Figure 13 (b). FFT has more communication compared to EP, and the effect of non-local references is amplified for networks with lower connectivity. For IS (Figure 13 (c)), 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 (and thus difficult to program to maximize locality), the difference between LogP and cLogP curves is more significant (see Figures 13 (d) and (e)) since the LogP implementation cannot be optimized statically to exploit locality.

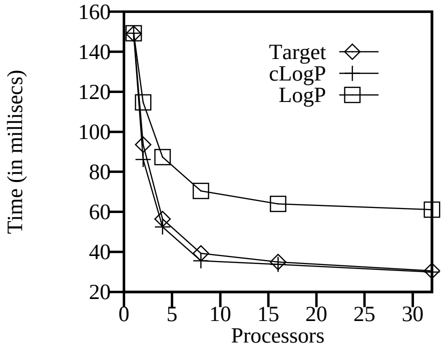
Isolating the latency and contention overheads from the total execution time helps us identify and quantify locality effects. Figures 11 (a), (b) and (c) illustrate some of these effects for FFT, CG, and EP respectively. During the communication phase in FFT, a processor reads consecutive data items from an array displaying spatial locality. In either the cLogP or the target 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



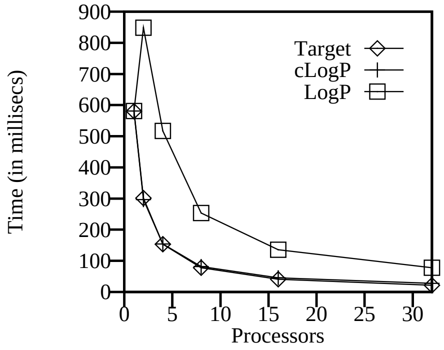
(a) EP on Full



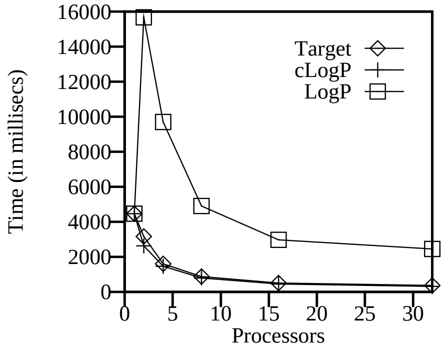
(b) FFT on Mesh



(c) IS on Full



(d) CG on Full



(e) CHOLESKY on Full

Figure 13: Execution Time

on the LogP machine incurs a latency (Figure 11 (a)) which is approximately four times that of the other two. Similarly, ignoring spatial and temporal locality in CG (Figure 11 (b)) 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 signaled by another (see the Appendix). 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 11 (c). Similarly, a test-test&set primitive [9], would behave like an ordinary test&set operation in the LogP machine. As can be seen in Figure 13 (a), 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 target machine. The above results clearly show that the execution curves of cLogP and the target 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 target 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 over the chosen range of applications.

5.4 Summary

In all the considered cases, the latency overhead from the LogP abstraction and the target network closely agree. On the other hand, there is a disparity between the model and the target network for the contention overhead in many cases. Since g is computed using only the bisection bandwidth of the network (as is suggested in [20]), 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 increases with decreasing connectivity in the 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. We could thus fix some of the simplifying assumptions made in developing analytical models and use the datapoints from simulation to refine such models.

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, even for static applications, it is not easy to model all aspects of spatial and temporal locality. In dynamic applications, this problem is exacerbated owing to factors such as dynamic scheduling and synchronization. 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 target machine.

The network messages incurred in our cLogP abstraction represents the minimum overhead for any invalidation-based cache coherence protocol in conjunction with a sequentially consistent memory model. We compared the performance of such an abstraction with a machine that incorporates a simple 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 target machine is expected to agree even more with the chosen abstraction. This result suggests that cache coherence overhead is insignificant 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.

For a system architect, the low coherence-related network activity has important implications. We cannot hope to gain significantly by incorporating fancier invalidation-based cache coherence protocols in our sequentially-consistent shared memory platform. While working under the constraints of an invalidation-based cache coherence scheme, we need to direct our efforts towards lowering (by fast network hardware as is discussed in Chapter 6) or tolerating (by weaker memory models and prefetch/poststore mechanisms as is discussed in Chapter 7) the inherent communication overhead resulting from the data access pattern of the application. On the other hand, we cannot make any such pronouncement for update-based schemes using the results from this exercise. It may be noted that Wood et al. [91] 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 these claims.

Our main motivation for studying the accuracy of abstractions is for speeding up execution-driven simulators using such abstractions. The simulation of the cLogP machine which incorporates the two abstractions is around around 250-300% faster than the simulation of the target 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 on the target machine take between 8-10 hours.

Isolation and quantification of parallel system overheads provided by SPASM is instrumental in conducting this study. For instance, even when total execution time curves are similar, the latency and contention overhead curves help determine whether the model parameters are accurate in capturing the individual machine abstractions. This helps us 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 application characteristics. We have thus illustrated the use of our framework in evaluating the validity of machine abstractions and the importance of these abstractions in easing the resource requirements of execution-driven simulators.

Chapter 6

Synthesizing Bandwidth Requirements

To speed today, to be put back tomorrow.

- Edmund Spenser

6.1 The Problem

A system architect is interested in finding out how much resources (in terms of money, effort and time) should be invested in different components of a parallel machine to get the maximum rewards. In the first place, he needs to know if existing resources are adequate since there is no point in providing more than what is needed in a machine. When costs become prohibitive, he would like to know how much performance would be lost if he cannot provide what is needed. Thus quantifying architectural requirements can help make cost vs. performance trade-offs in important design decisions towards building a well-balanced machine. The network is an important artifact in a parallel machine limiting the performance of many applications, and is the focus of this chapter. Adhering to our application-driven approach, we quantify the bandwidth requirements of the five applications identified in section 3.3. We also study the impact of different system parameters such as the number of processors, CPU clock speed, and application problem size on these bandwidth requirements.

Latency and *contention* are the two defining characteristics of a network from an application viewpoint. Both latency and contention depend on a number of factors including the connectivity, the bandwidth of the links in the network, the switching delays, and the length of the message. Of the architectural factors, link bandwidth and connectivity are the most crucial network parameters impacting latency and contention. Hence, in order to quantify requirements that limit network overheads (latency and contention) to an acceptable level, it is necessary to study the impact of link bandwidth and network connectivity on these overheads.

Several studies [24, 3, 2] use analytical models to study the impact of connectivity and link bandwidth for different networks. The results show that network requirements are sensitive to the workload making it necessary to study them in the context of real applications. Other researchers [65, 23] have used applications to quantify architectural requirements. Cypher et al. [23] use a range of scientific applications in quantifying the processing, memory, communication and I/O requirements. They present the communication requirements in terms of the number of messages exchanged between processors and the volume (size) of these messages. Communication in an application may be characterized by four attributes [70] : communication volume, the communication pattern, the communication frequency and the ability to overlap communication with computation. A static analysis of the communication as conducted by Cypher et al. [23] fails to capture the last two attributes, making it very difficult to quantify the contention in the system. On the other hand, a simulation platform such as SPASM can capture all the dynamic interactions in a parallel system.

6.2 The Approach

We simulate the execution of the five parallel applications (EP, IS, FFT, CG and CHOLESKY) on the CC-NUMA hardware platform outlined in section 3.3. For a specific network topology, we vary the bandwidth of the links in the network. As the link bandwidth is increased, the efficiency of the system is also expected to increase as shown in Figure 14. But we soon reach a point of diminishing returns beyond which increasing the bandwidth does not have a significant impact on application performance (the curve flattens) since the network overheads (both latency and contention)

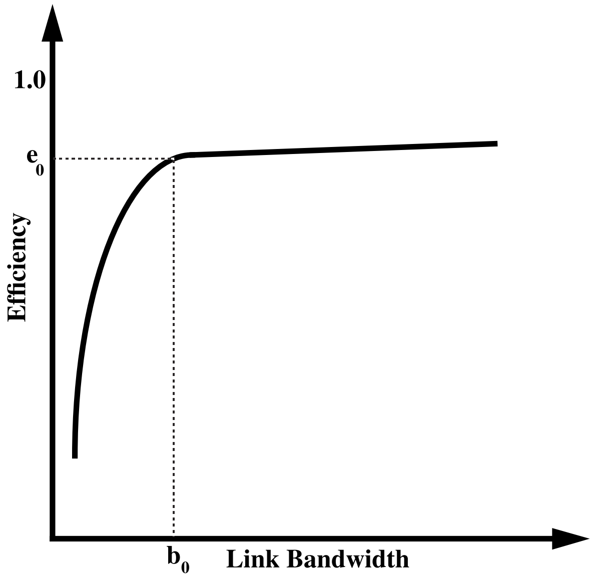


Figure 14: Link Bandwidth vs. Efficiency

are sufficiently low at this point. In all our results, we observe such a distinct knee. One would expect the efficiency beyond this knee to be close to 100%. But owing to algorithmic overheads such as serial part or work-imbalance, the knee may occur at a much lower efficiency (e_0 in Figure 14 may be much lower than 1.0). These algorithmic overheads may also cause the curves for each configuration of system parameters to flatten out at entirely different levels in the efficiency spectrum. The bandwidth corresponding to the knee (b_0) still represents an ideal point at which we would like to operate since the network overheads beyond this knee are minimal and the network is no longer the bottleneck for any loss of efficiency.

As we mentioned earlier, a system architect is interested in finding out how the bandwidth requirements will change as different system parameters are varied. The system parameters that we investigate in this study are the number of processors, CPU clock speed and the application problem size. We track the horizontal movement of the knee in Figure 14 as the different system parameters are varied. The experiments have been conducted over a range of processors (p=4, 8, 16, 32, 64), CPU clock speeds (s=33, 100, 300 MHz) and link bandwidths (b=1, 20, 100, 200, 600 and 1000

MBytes/sec). The problem size n of the applications has been varied as 16K, 32K, 64K, 128K and 256K for EP, IS and FFT, 1400×1400 and 5600×5600 for CG, and 1806×1806 for CHOLESKY. In studying the effect of each parameter, we keep the other two constant. Instead of presenting all the detailed graphs, we summarize here the general observations drawn from the graphs.

For all the applications, we observed that the knee shifts to the right as the number of processors is increased indicating the need for higher bandwidth. As the number of processors is increased, the network accesses incurred by a processor in the system may increase or decrease depending on the application, but each such access would incur a larger overhead from contending for network resources (due to the larger number of messages in the network as a whole for the chosen applications). Further, the computation performed by a processor is expected to decrease, lowering the computation to communication ratio, thus making the network requirements more stringent. As the CPU clock speed is increased, the computation to communication ratio decreases. In order to sustain the same efficiency, communication has to be sped up to keep pace with the CPU speed thus shifting the knee in Figure 14 to the right uniformly across all applications. An increase in problem size is likely to increase the amount of computation performed by a processor. At the same time, a larger problem may also increase the network accesses incurred by a processor. In EP, FFT, CG, and CHOLESKY the former effect is more dominant thereby increasing the computation to communication ratio, making the knee move to the left as the problem size is increased. The two counteracting effects nearly compensate each other in IS showing negligible shift in the knee.

We track the change in the knee in Figure 14 as a function of system parameters by quantifying the link bandwidth needed to limit the network overheads to a certain fraction of the overall execution time. This fraction would determine the closeness of the operating point to the knee. For instance, if the network overhead is less than 10% of the overall execution time, then it amounts to saying that we are

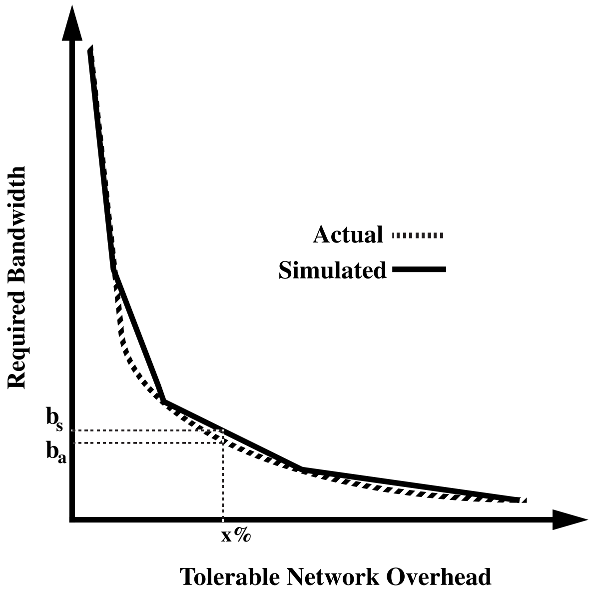


Figure 15: Network Overhead vs. Bandwidth Requirements

achieving an efficiency that is within 90% of the ideal efficiency (on a machine with zero network overhead). Ideally, one would like to operate as close to the knee as possible. But owing to cost or technological constraints, one may be forced to operate at a lower bandwidth and it would be interesting to investigate if it may still be possible to obtain reasonable efficiencies under these constraints. Figure 15 shows the trade-off between the tolerable network overhead and the resulting bandwidth that needs to be sustained to maintain the overhead within a specified level. With the ability to tolerate a larger network overhead, the bandwidth requirements are expected to decrease as shown by the curve labeled "Actual" in Figure 15. To calculate the bandwidth requirement needed to limit the network overhead (both the latency and contention component) to a certain value, we simulate the applications and the network in their entirety over a range of link bandwidths. We plot the bandwidths and the resulting network overheads as shown by the curve labeled "Simulated" in Figure 15. We perform a linear interpolation between these data points to calculate the bandwidth (b_s) required to limit the network overhead to $x\%$ of the total execution time. This bandwidth would represent a good upper bound on the corresponding "Actual" bandwidth (b_a) required. Instead of presenting all the interpolated graphs, we simply tabulate the requirements for x = 10%, 30% and 50% in the following discussions. These requirements are expected to change with the number of processors, the CPU clock speed and the application problem size. The rate of change in the knee is used to study the impact of these system parameters. In cases where the analysis is simple, we use our knowledge of the application and architectural characteristics in extrapolating the performance for larger systems. In cases where such a static analysis is not possible (due to the dynamic nature of the execution), we perform a non-linear regression analysis of the simulation results using a multivariate secant method with a 95% confidence interval in the SAS [66] statistics package.

6.3 Performance Results

Using the above methodology, we now discuss for each application its intrinsic characteristics that impact the communication and computation requirements; present the link bandwidth requirements as a function of increasing number of processors, CPU clock speed, and problem size; and project the requirements for a 1024-node system with a problem size appropriate for such a system. The results presented in the following discussions are for a binary hypercube topology. In section 6.4 we will discuss how these requirements would change for other topologies.

EP

EP has a high computation to communication ratio with the communication being restricted to a few logarithmic global sum operations. The bulk of the time is spent in local computation and as a result, even a bandwidth of 1 MByte/sec is adequate to limit network overheads to less than 10% of the execution time (see Table 8). As the number of processors is increased, the communication incurred by a processor in the global sum operation grows logarithmically and a bandwidth of 10 MBytes/sec can probably sustain even a system with 1024 processors. As the clock speed is increased, the time spent by a processor in the local computation is expected to decrease linearly and the bandwidth requirement for the global sum operation needs to increase at the same rate in order to maintain the same efficiency. Table 9 reflects this behavior. As the problem size (n) is increased for this application, the local computation incurred by a processor is expected to grow as $O(n)$ while the communication (both the number of global sum operations as well as the number of messages for a single operation) remains the same. As a result, the bandwidth requirements are expected to decrease linearly with problem size. Given that real world problem sizes for this application are of the order of $n = 2^{28}$ [11], a very low link bandwidth (less than 1 MByte/sec) would suffice to yield an efficiency close to 100%.

Table 8: EP: Impact of Processors on Link Bandwidth (in MBytes/sec)

	50% ovhd.	30% ovhd.	10% ovhd.
p=4	< 1.0	< 1.0	< 1.0
p=8	< 1.0	< 1.0	< 1.0
p=16	< 1.0	< 1.0	< 1.0
p=32	< 1.0	< 1.0	< 1.0
p=64	< 1.0	< 1.0	1.17
B/w Fns.	-	-	-

n=128K, s=33 MHz

Table 9: EP: Impact of CPU speed on Link Bandwidth (in MBytes/sec)

	50% ovhd.	30% ovhd.	10% ovhd.
s=33MHz	0.54	0.90	1.17
s=100MHz	1.69	2.81	3.67
s=300MHz	4.68	7.80	10.41

p=64, n=128K

IS

IS is more communication intensive than EP and its bandwidth requirements are expected to be considerably higher. There are two dominant phases in the execution that account for the bulk of the communication (see the Appendix). In the first, a processor accesses a part of the local buckets of every other processor in making updates to the global buckets allotted to it. In the second phase, a processor locks each of the global buckets (that is partitioned by consecutive chunks across processors) in ranking the list of elements allotted to it. With an increase in the number of processors, a processor needs to access data from more processors in the former phase. In the latter, the amount of global buckets that is allocated to a processor decreases linearly with increase in processors due to the partitioning scheme. Hence, in both these phases, the communication is expected to grow as $O(p)$ with increase in processors. Further, the computation performed by a processor decreases with an increase in processors, but the rate is less than linear owing to algorithmic deficiencies in the problem (Section 4.3). These factors combine to yield a considerable bandwidth requirement for larger systems (see Table 10), if we are willing to tolerate less than 10% network overheads. The bandwidth function has been obtained by performing a non-linear regression analysis of the simulation data points for the given system parameter, and the resulting function has been used to calculate the requirements for the 1024 node system. As the CPU clock speed is increased, the computation to communication ratio decreases, making the requirements more stringent as shown in Table 11. As the problem size (n) is increased, the communication in the above mentioned phases increases linearly. The local computation also increases, but the former effect is more prominent as is shown in Table 12, where the bandwidth requirements grow moderately with problem size.

Using these results, the bandwidth requirements for IS are projected in Table 13 for a 1024 node system and a problem size of 2^{23} that is representative of a real world problem [11]. This table shows that bandwidth requirements of IS are

Table 10: IS: Impact of Processors on Link Bandwidth (in MBytes/sec)

	50% ovhd.	30% ovhd.	10% ovhd.
p=4	7.75	12.91	68.69
p=8	13.38	30.75	92.87
p=16	22.00	66.44	168.71
p=32	38.65	78.61	211.45
p=64	47.03	84.61	293.44
B/w Fns.	$23.60p^{0.28} - 28.79$	$74.41p^{0.22} - 91.21$	$88.68p^{0.34} - 82.12$
p=1024	143.25	251.91	907.80

n=128K, s=33 MHz

Table 11: IS: Impact of CPU speed on Link Bandwidth (in MBytes/sec)

	50% ovhd.	30% ovhd.	10% ovhd.
s=33MHz	47.03	84.61	293.44
s=100MHz	102.49	224.69	770.16
s=300MHz	356.14	649.95	1344.72

p=64, n=64K

considerably high. We may at best be able to operate currently at around 50% network overhead range with 33 MHz processors given that link bandwidth of state-of-the-art networks is around 200-300 MBytes/sec. With faster processors like the DEC Alpha, the network becomes an even bigger bottleneck for this application.

In projecting the above bandwidth requirements for this application with 1024 processors, both the number of buckets as well as the number of list elements have been increased for the larger problem. But bucket sort is frequently used in cases where the number of buckets is relatively independent of the number of elements in the list to be sorted. A scaling strategy where the size of the list is increased and the number of buckets is maintained constant would cause no change in communication in the above mentioned phases of IS, while the computation is expected to grow as $O(n)$. Hence, if we employ such a scaling strategy and increase the problem size linearly with the CPU clock speed, we may be able to limit the network overheads to

Table 12: IS: Impact of Problem Size on Link Bandwidth (in MBytes/sec)

	50% ovhd.	30% ovhd.	10% ovhd.
n=16K	46.60	83.80	270.08
n=32K	47.16	84.52	286.98
n=64K	47.03	84.61	293.44
n=128K	47.48	85.09	303.41
n=256K	48.67	85.53	307.75
B/w Fns.	$0.007n^{1.00} + 46.61$	$0.006n^{0.99} + 84.10$	$19.57n^{0.26} + 230.19$
n=8192K	110.75	133.19	441.66

p=64, s=33 MHz

Table 13: IS: Link Bandwidth Projections (in MBytes/sec)

	50% ovhd.	30% ovhd.	10% ovhd.
s=33MHz	337.34	396.55	1366.34
s=100MHz	735.15	1053.08	3586.08
s=300MHz	> 5000	> 5000	> 5000

$p = 1024$, $n = 2^{23}$

within 30-50% for this application with existing technology.

FFT

The computation performed by a processor in FFT grows as $O((n \log n)/p)$ while the communication grows as $O(n(p-1)/p^2)$. Thus, these components decrease at comparable rates with an increase in the number of processors. As the number of processors is increased, the contention encountered by each message in the network is expected to grow. However, due to skewing of the communication accesses (see the Appendix) the bandwidth requirements of the network grow slowly with the number of processors as is shown in Table 14. These requirements can be satisfied even for faster processors (see Table 15). As we mentioned earlier, the computation to communication ratio is proportional to $O(\log n)$, and the network requirements are expected to become even less stringent as the problem size is increased. Table 16 confirms this observation.

Table 14: FFT: Impact of Processors on Link Bandwidth (in MBytes/sec)

	50% ovhd.	30% ovhd.	10% ovhd.
p=4	< 1.0	6.40	16.35
p=8	< 1.0	6.52	16.40
p=16	< 1.0	7.52	16.75
p=32	< 1.0	7.83	16.87
p=64	< 1.0	8.65	17.19
B/w Fns.	-	$0.75p^{0.36} + 5.11$	$0.01p^{0.99} + 16.37$
p=1024	-	14.85	29.93

n=64K, s=33 MHz

Table 15: FFT: Impact of CPU speed on Link Bandwidth (in MBytes/sec)

	50% ovhd.	30% ovhd.	10% ovhd.
s=33MHz	< 1.0	8.65	17.19
s=100MHz	8.65	13.81	29.86
s=300MHz	17.19	29.20	88.81

p=64, n=64K

In projecting the requirements for a 1024-node system, link bandwidths of around 100-150 MBytes/sec would suffice to limit the network overheads to less than 10% of the execution time (see Table 17). The results shown in the above tables confirm the theoretical results presented in [35] where the authors show that FFT is scalable on the cube topology and the achievable efficiency is only limited by the ratio of the CPU clock speed and the link bandwidth.

CG

The main communication in CG occurs in the multiplication of a sparse matrix with a dense vector (see the Appendix). Each processor performs this operation for a contiguous set of rows allocated to it. The elements of the vector that are needed by a processor to perform this operation depend on the distribution of non-zero elements in the matrix and may involve external accesses. Once an element is fetched, a processor

Table 16: FFT: Impact of Problem Size on Link Bandwidth (in MBytes/sec)

	50% ovhd.	30% ovhd.	10% ovhd.
n=16K	< 1.0	9.42	17.63
n=32K	< 1.0	9.03	17.45
n=64K	< 1.0	8.65	17.19
n=128K	< 1.0	8.38	17.03
n=256K	< 1.0	7.97	16.84
B/w Fns.	-	$11.02 - 0.4 \log n$	$18.43 - 0.2 \log n$
n=2^{30}	-	3.02	14.43

p=64, s=33 MHz

Table 17: FFT: Link Bandwidth Projections (in MBytes/sec)

	50% ovhd.	30% ovhd.	10% ovhd.
s=33MHz	-	5.15	21.64
s=100MHz	-	8.22	48.58
s=300MHz	-	17.38	144.50

$p = 1024$, $n = 2^{30}$

may reuse it for a non-zero element in another row at the same column position. As the number of processors is increased, the number of rows allocated to a processor decreases thus decreasing the computation that it performs. Increasing the number of processors has a dual impact on communication. Since the number of rows that need to be computed decreases, the probability of external accesses decreases. There is also a decreased probability of reusing a fetched data item for computing another row. These complicated interactions are to a large extent dependent on the input data and are difficult to analyze statically. We use the data sets supplied with the NAS benchmarks [11]. The results from our simulation are given in Table 18. We observe that the effect of lower local computation, and lesser data reuse has a more significant impact in increasing the communication requirements for larger systems. Increasing the clock speed has an almost linear impact on increasing bandwidth requirements as given in Table 19. As the problem size is increased, the local computation increases,

Table 18: CG: Impact of Processors on Link Bandwidth (in MBytes/sec)

	50% ovhd.	30% ovhd.	10% ovhd.
p=4	1.74	2.90	8.71
p=8	3.25	5.41	16.23
p=16	5.81	9.68	52.03
p=32	9.73	16.22	82.39
p=64	15.63	46.10	124.19
B/w Fns.	$1.25p^{0.62} - 1.28$	$0.04p^{1.63} + 3.61$	$18.80p^{0.51} - 33.07$
p=1024	94.79	393.32	618.28

n=1400*1400, s=33 MHz

Table 19: CG: Impact of CPU speed on Link Bandwidth (in MBytes/sec)

	50% ovhd.	30% ovhd.	10% ovhd.
s=33MHz	15.63	46.10	124.19
s=100MHz	43.50	96.75	386.12
s=300MHz	120.89	262.84	1022.14

p=64, n=1400*1400

and the probability of data re-use also increases. The rate at which these factors impact the requirements depends on the sparsity factor of the matrix. Table 20 shows the requirements for two different problem sizes. For the 1400×1400 problem, the sparsity factor is 0.04, while the sparsity factor for the 5600×5600 problem is 0.02. The corresponding factor for the 14000×14000 problem suggested in [11] is 0.1 and we scale down the bandwidth requirements accordingly in Table 21 for a 1024 node system. The results suggest that we may be able to limit the overheads to within 50% of the execution time with existing technology. As the processors get faster than

Table 20: CG: Impact of Problem Size on Link Bandwidth (in MBytes/sec)

	50% ovhd.	30% ovhd.	10% ovhd.
n=1400*1400	15.63	46.10	124.19
n=5600*5600	9.48	25.47	78.55

p=64, s=33 MHz

Table 21: CG: Link Bandwidth Projections (in MBytes/sec)

	50% ovhd.	30% ovhd.	10% ovhd.
s=33MHz	34.87	120.04	247.33
s=100MHz	97.05	251.93	1200.49
s=300MHz	269.7	684.41	2035.64

p=1024, n=14000*14000

100 MHz, it would need a considerable amount of bandwidth to limit the overheads to within 30%. But with faster processors, and larger system configurations, one may expect to solve larger problems as well. If we increase the problem size (number of rows of the matrix) linearly with the clock speed of the processor, one may expect the bandwidth requirements to remain constant, and we may be able to limit network overheads to within 30% of execution time even with existing technology.

CHOLESKY

Each processor while working on a column in this application will need to access the non-zero elements in the same row position of other columns. Once a non-local element is fetched, the processor can reuse it for the next column that it has to process. The communication pattern in processing a column is similar to that of CG. The difference is that the allocation of columns to processors in CHOLESKY is done dynamically. As with CG, an increase in the number of processors is expected to decrease the computation to communication ratio as well as the probability of data reuse. Further, the network overheads for implementing dynamic scheduling are also expected to increase for larger systems. Table 22 reflects this trend, showing that bandwidth requirements for CHOLESKY grow modestly with increase in processors. Still, the requirements may be easily satisfied with existing technology for 1024 processors. Even with a 300 MHz clock, one may be able to limit network overheads to around 30% as shown in Table 23. Owing to resource constraints, we have not been able to simulate other problem sizes for CHOLESKY in this study. But an increase in problem size is expected to increase the computation to communication ratio (experimentally verified on the KSR-1) which suggests that bandwidth requirements are expected to decrease with problem size. Hence, as the processors get faster, one may still be able to maintain network overheads at an acceptable level with existing technology if the problem size is increased correspondingly.

Table 22: CHOLESKY: Impact of Processors on Link Bandwidth (in MBytes/sec)

	50% ovhd.	30% ovhd.	10% ovhd.
p=4	5.62	11.98	76.56
p=8	6.86	13.11	78.32
p=16	7.77	14.48	80.83
p=32	8.91	16.02	84.12
p=64	10.49	17.48	87.35
B/w Fns.	$1.47p^{0.37} + 3.43$	$2.14p^{0.33} + 8.82$	$6.77p^{0.27} + 66.60$
p=1024	23.44	31.26	110.70

n=1806*1806, s=33 MHz

Table 23: CHOLESKY: Impact of CPU speed on Link Bandwidth (in MBytes/sec)

	50% ovhd.	30% ovhd.	10% ovhd.
s=33MHz	10.49	17.48	87.35
s=100MHz	16.56	60.13	278.92
s=300MHz	29.51	171.29	712.60

p=64, n=1806*1806

6.4 Summary

In this chapter, we quantified the link bandwidth requirements of five applications for the binary hypercube topology as a function of the number of processors, CPU clock speed and problem size. Based on these results we also projected the requirements of large systems built with 1024 processors and CPU clock speeds upto 300 MHz.

We observed that EP has negligible bandwidth requirements and FFT has moderate requirements that can be easily sustained. The network overheads for CG and CHOLESKY may be maintained at an acceptable level for current day processors, and as the processor speed increases, one may still be able to tolerate these overheads with existing link bandwidth technologies of 200-300 MBytes/sec provided the problem size is increased commensurately. The network overheads of IS are tolerable for slow processors, but the requirements become unmanageable as the clock speed increases. As we observed, the deficiency in this problem is in the way the problem is scaled (the number of buckets is scaled linearly with the size of the input list to be sorted). On the other hand, if the number of buckets is maintained constant, it may be possible to sustain bandwidth requirements by increasing the problem size linearly with the processing speed.

The above results have been presented for the binary hypercube network topology. The cube represents a highly scalable network where the bisection bandwidth grows linearly with the number of processors. Even though cubes of 1024 nodes have been built [36], cost and technology factors often play an important role in its physical realization. Agarwal [3] and Dally [24] show that wire delays (due to increased wire lengths associated with planar layouts) of higher dimensional networks make low dimensional networks more viable. The 2-dimensional [48] and 3-dimensional [18, 8] toroids are common topologies used in current day networks, and it would be interesting to project link bandwidth requirements for these topologies.

A metric that is often used to compare different networks is the bisection bandwidth available per processor. On a k-ary n-cube, the bisection bandwidth available per processor is inversely proportional to the radix k of the network. One may use a simple rule of thumb of maintaining per processor bisection bandwidth constant in projecting requirements for lower connectivity networks. This argument is very similar to the rationale behind the g parameter of LogP [20] which models network contention. For example, considering a 1024-node system, the link bandwidth requirement for a 32-ary 2-cube would be 16 times the 2-ary 10-cube bandwidth; similarly the requirement for a 3-D network would be around 5 times the 10-D network. As we mentioned in Chapter 5, such a projection becomes pessimistic since it assumes that the communication in an application is devoid of any network locality and that each message crosses the bisection. With a little knowledge about the communication behavior of applications, one may be able to reduce the degree of pessimism. In both FFT and IS, every processor communicates with all other processors, and thus only 50% of the messages cross the bisection. Similarly, instrumentation in our simulation showed that only around 50% of the messages in CG and CHOLESKY traverse the bisection. To reduce the degree of pessimism in these projections, one may thus introduce a correction factor of 0.5 that can be multiplied with the above-mentioned factors of 16 and 5 in projecting the bandwidths for 2-D and 3-D networks respectively. EP would still need negligible bandwidth and we can still limit network overheads of FFT to around 30% on these networks with existing technology. But the problem sizes for IS, CG and CHOLESKY would have to grow by a factor of 8 compared to their hypercube counterparts if we are to sustain the corresponding efficiency on a 2-D network with current technology. Despite the correction factor, these projections are still expected to be pessimistic since the temporal aspect of communication is ignored. The projection assumes that every message in the system traverses the bisection at the same time. If the message pattern is temporally skewed, then a lower link bandwidth may suffice for a given network overhead. It is almost impossible to determine these skews statically, especially for applications like CG and CHOLESKY where the communication pattern is dynamic. It is necessary to conduct a detailed simulation for these network topologies to confirm these projections, but that is beyond the scope of this thesis.

Using the technique outlined in this paper, it would be possible for an architect to synthesize the bandwidth requirements of an application as a function of system parameters. For instance, given a set of applications, the system size (number of processors) and the CPU speed, an architect may use this technique to calculate the bandwidth that he needs to support in hardware. In cases where cost/technological problems prohibit supporting this bandwidth, the architect may use the results to find out the efficiency that would result from a lower hardware bandwidth or the factor by which the problem size needs to be scaled to maintain good efficiency. The results may also be used to quantify the rate at which the network (which is often custom-built) capabilities have to be enhanced in order to accommodate the rapidly improving off-the-shelf components used in realizing the processing nodes.

The separation of overheads provided by our framework is crucial in quantifying the network bandwidth requirements. As we mentioned earlier in Chapter 2, several factors (apart from the network overheads) can limit the scalability of a parallel system. The separation of overheads enables us to discount the contribution of all other factors other than the network overheads. We were thus able to quantify the bandwidth needed to limit the network overheads within a tolerable factor. Further, conducting this evaluation on an execution-driven simulator such as SPASM helped us to faithfully capture all the attributes (volume, pattern, frequency, and tolerance) of communication in applications, and vary the different system parameters to investigate their impact on bandwidth requirements.

Chapter 7

Synthesizing Communication Mechanisms

Few things are harder to put up with than a good example.
- Mark Twain

7.1 The Problem

Scalability studies of the representative parallel applications on a range of network topologies (Chapter 4) clearly showed that the single most important overhead limiting performance from the architectural viewpoint is the communication overhead. There are two solution approaches for this problem. One solution is to make the network as fast as possible so that even though the application does not make any fewer network accesses, the overheads will not manifest as a significant component of the total execution time. This is the strategy adopted in the previous chapter where we quantified the link bandwidth required to limit network overheads within a tolerable factor. But the resources to sustain this bandwidth may simply not be available in some cases. The second approach is to reduce the network accesses incurred in the execution or to tolerate the communication overhead if these accesses are unavoidable. Cache coherence protocols [10, 79, 15], weak memory consistency models [46, 28, 1, 32, 47], prefetch [53], poststore, and multithreading [5, 89] are some of the proposed latency reducing and tolerating techniques. It has been shown that no one technique is universally applicable for all applications [34]. On the other hand, a close examination of the communication behavior of a range of applications can help derive a set of architectural mechanisms that may prove beneficial and we conduct such a study in this chapter. By examining the communication properties of applications, we propose a set of explicit communication primitives that are generalizations of the poststore and prefetch mechanisms. Using information available from the application (via the compiler) and/or the hardware, these primitives effect poststore and prefetch intelligently.

In shared memory machines, it is customary to provide private caches with each processor that facilitate the exploitation of locality by replicating data at each node. Cache coherence protocols maintain consistency across these caches to conform with the underlying memory consistency model. Such protocols broadly fall into two categories: *write-invalidate* and *write-update.* Invalidation-based schemes are more suited to migratory data and can become inefficient when the producer-consumer relationship for shared data remains relatively unchanged during the course of execution. On the other hand, update-based protocols can result in significant overheads due to repeated updates to the same data before they are used by another processor, as well as redundant updates when there are changes to the sharing pattern of a data item. The update and invalidation based schemes thus have their relative advantages and disadvantages, and based on application characteristics one may be preferable over the other. Invalidations are useful when an application changes its sharing pattern, and updates are useful to effect direct communication once a sharing pattern is established.

There are hardware schemes [54] which dynamically switch between invalidation and update based schemes. But relying entirely on the hardware may not provide the flexibility to adapt to the inherent characteristics of different applications. Recognizing that a fixed protocol for cache consistency may not suffice, there have been recent proposals [37, 63] to have a menu of coherence protocols implemented in software so that an application can choose the right one depending on its communication characteristics. However, this may increase the programming complexity by requiring the application developer to choose the right coherence protocol. Therefore, there is a need for striking a balance between flexibility and programming ease. We need hardware/software mechanisms that give the flexibility for dynamically adapting the way coherence actions are effected, while at the same time not imposing a heavy burden on the programmer in terms of programming complexity. Toward this end, we first need to look at applications and study their communication/sharing characteristics. These characteristics can in turn suggest primitives that should be supported in the underlying system which can be implemented either in hardware or software. As far as possible, our objective is to relegate the task of using these primitives to the compiler so that the programming complexity for the application programmer is not increased.

7.2 Communication in Applications

Sender-initiated, where the producer sends the data item to one or more consumers as soon as the data item is produced, and *receiver-initiated*, where a consumer requests and receives the data item from the producer, are the two main communication paradigms employed in applications. For performance reasons, it is beneficial to use sender-initiated communication wherever possible because: 1) the transfer of data item can be effected with a single message instead of a request and a transfer message needed with the receiver-initiated case; 2) synchronization between the producer and consumer is usually needed in the receiver-initiated case so that the receiver can know when to request for the data. On shared memory machines, update-based cache coherence schemes and the poststore operation try to avail of the benefits of sender-initiated communication while invalidation-based schemes and the prefetch operation default to receiver-initiated communication. To reap the full benefits of sender-initiated communication, the sender needs to know *who* will be the next set of consumers for a data item, and *what* data items to send. Leaving the who/what questions to be answered entirely by the hardware (as is done with simple update-based cache coherence schemes) or to be answered speculatively with mechanisms like poststore, may not provide the full performance benefits. On the other hand, an examination of the communication in applications can help answer these questions.

7.2.1 Sender-initiated Communication

Communication in applications may be broadly classified into *static* and *dynamic*.

Static

If the data access pattern of an application can be pre-determined at compile-time then the resulting communication is defined as static. FFT (Fast Fourier Transform), SOR (Successive Over Relaxation), and Matrix Multiplication, are examples of such applications where the entire execution displays static communication pattern. There are several other applications where the communication is static at least for certain phases. For instance, global sums which use a static logarithmic tree structure to effect communication are frequently used in applications. For such static communication, the producer knows the exact set of consumers. Hence, in order to optimize on static communication, it would be beneficial to have an explicit communication mechanism wherein a producer can selectively propagate a data item to a set of consumers that it specifies. The compiler may be able to use such a primitive to effect the communication by analyzing the data access pattern of the application without placing undue burden on the programmer.

Dynamic

Several other applications fall in the dynamic category where the communication varies with the dynamics of program execution. This dynamism poses some problem in determining the consumer set for effecting explicit communication. Fortunately, synchronization information in the program can be used sometimes for triggering such actions. For instance, consider the data items produced in a critical section under the control of a mutual exclusion lock. Figure 16 shows this scenario where

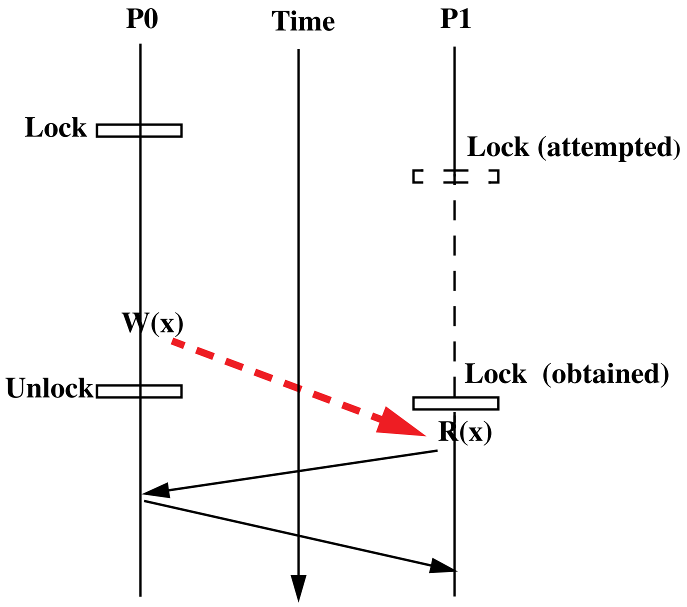


Figure 16: Accesses to variables governed by locks

P0 writes a variable in a critical section which is subsequently read by P1 after obtaining the lock. If we had some mechanism for P0 to selectively send data to P1, that would reduce the observed data access latency for P1. Clearly some programmer help is needed in associating data items with the lock structure. Also some help is needed in implementing the lock library calls such that any waiting processor for a lock is identifiable through the lock structure. Assuming such help is available, if there was a mechanism that allows explicit communication then the compiler could use the association of data with the lock for P0 to send the data when produced to any waiting P1 (which can be looked up using the lock structure). Figure 16 shows this sender-initiated transfer using broken lines. In application scenarios exhibiting significant lock contention (where combining synchronization with data transfer helps significantly) such a mechanism will help even more since there is very high likelihood for a consumer to be waiting for the lock. Producer-consumer applications such as those which manage queues dynamically can hope to gain from this scheme.

Many applications in the dynamic category go through phases where the sharing pattern does not change within a phase but changes across phases. The producer may not be able to provide a consumer set to effect explicit communication as in the static cases. The Nbody simulation using the Barnes-Hut algorithm is a typical application which exhibits such behavior. The application simulates the motion of bodies over time steps, and in each step the motion of a body is determined by the state of other bodies that can affect its motion. In successive steps, the bodies are not expected to move drastically and the communication pattern would thus remain the same for these steps. But, over several time steps the bodies can move to establish a different sharing pattern. As we mentioned earlier, an invalidation-based protocol would be unsuitable for the successive steps where the sharing pattern does not change. A simple minded update protocol would send updates to a superset of the old consumer set and would not adapt itself to changes in the sharing pattern. For such applications, it would be useful to have a mechanism that adapts itself to a new sharing pattern dynamically and sends updates to the consumers in the new set. For another class of applications such as EM3D [21] and CG in the dynamic category, the communication pattern may not be apparent at compile time but the communication becomes deterministic once the program begins execution (i.e. the input data is read in). Since communication is dependent on the input data set, the compiler cannot explicitly give the consumer set when the data item is produced. However it can be seen that this sharing pattern is just a special case of the dynamic sharing pattern observed in applications such as Barnes-Hut. Therefore, if we had a mechanism for handling the general case then that would work for this special case as well.

In all the above cases, we have attempted to exploit sender-initiated data transfer by studying application characteristics to find the consumer set. In the static case, the information is gleaned directly from the application. In the first of the two dynamic cases, the information could be obtained using association of data with locks and the synchronization structure itself (such as a lock). In the second dynamic case, the information has to be gleaned by tracking the sharing pattern of the application at runtime.

7.2.2 Receiver-initiated Communication

There are several applications with dynamic communication characteristics where the sharing information is simply not available at the time the data is produced. In applications such as IS and CHOLESKY where there is no lock contention (Chapter 4), the producer cannot effect the data transfer since it does not know the next consumer (no consumer is waiting for the lock). We then need to resort to receiver-initiated communication where the consumer explicitly requests the data item. The prefetch mechanism is usually the best bet in such situations. However, since it is in units of the cache line it has limitations when used for fetching large structures. For instance, a processor in CHOLESKY gains access to a critical section via a lock variable, accesses all the elements in the column associated with this lock variable, and then relinquishes control of the critical section. To cut down on the number of messages, and to enable streaming of the data back to the requesting processor it would be beneficial to have a prefetch-like primitive that is decoupled from the cache line size which can prefetch all the elements of a column at the same time.

7.3 Architectural Mechanisms

Having identified application needs for minimizing communication overhead, we now present architectural mechanisms that help us accomplish these goals. These mechanisms implement the optimizations for the typical application communication scenarios described in Section 7.2. Our application-driven approach has thus made it possible to derive these mechanisms. The mechanisms are to be implemented in a shared memory machine with coherent caches that uses an invalidation-based protocol with a directory to maintain the caches consistent. The important point to note is that these mechanisms are integrated with the basic underlying coherence protocol. In this sense they are very different from the explicit communication primitives such as those proposed in MIT Alewife [42] or Stanford Flash [90], in that there is no address space management nor explicit coherence maintenance burden at the application level on the programmer for using the primitives.

The purpose of the following discussion is to merely show how the primitives have been derived from the communication scenarios outlined in the earlier section illustrating the power of our application-driven approach. Consequently, we do not discuss the details of the underlying coherence protocol nor the details of the coexistence of the primitives with the coherence protocol. The reader is referred to a related thesis [67] for further details on the implementation of these primitives. Of the proposed primitives, PSET_WRITE provides explicit communication for the static case, SYNC_WRITE provides explicit communication for the dynamic case within critical sections, and the two variants of the RSTREAM primitive provide arbitrary-sized prefetch capabilities. For the dynamic case with varying sharing patterns across different phases of execution, we provide a primitive called SEL_WRITE to be used in conjunction with normals reads and writes.

PSET_WRITE(address, value, pmask): This primitive is used for static communication where the processor issuing this request gives an explicit mask (*pmask*) of the processors to which the new *value* for the *address* is to be propagated. At the end of this operation, all processors in pmask have a valid copy of the block reflecting the latest value, while the copies in processors which were valid earlier but were not present in pmask are invalidated.

SYNC_WRITE(address, value, pmask): This primitive is just a special case of the PSET_WRITE primitive to be used by the software when association of locks with the data they govern is available via programmer annotations. Presumably this primitive is issued by a processor that is currently holding a mutual exclusion lock. The compiler can generate code by de-referencing the lock structure governing the critical section to send an update to the first processor (if any) waiting on this lock. Thus 'pmask' is a singleton processor that is next in line to get the lock. It may be worthwhile to have a variant of this primitive where the ownership of the location is transfered to the processor in pmask along with the data if that processor will subsequently write into it.

SEL_WRITE(address, value): As we mentioned earlier, applications sometime go through phases where the sharing pattern does not change within a phase but changes across phases. The SEL_WRITE (Selective Write) primitive is used to send updates within a phase, and the set of processors to which updates should be sent is dynamically determined from the current sharing pattern (available in the memory directory). Across phases, we need some way of detecting changes in the sharing pattern and use this information to effect the change in the directory accordingly. The behavior of Barnes-Hut which goes through these phases of sharing patterns gives us one way of detecting the change. In this application, when a new reader joins an existing sharing pattern that is already formed, then it is an indication of a changing sharing pattern. At this point, we can wipe out the existing pattern (invalidate the caches that had copies), indicate to the directory that there is no pattern currently, and allow the new pattern to form again. Subsequent readers would only cause the sharer set to form and not wipe out the pattern since the directory indicates that the new pattern has not formed. The new pattern is set to the formed state in the directory on a subsequent SEL_WRITE by a producer. Using this idea, we can construct a coherence protocol which adapts dynamically to changes in the sharing pattern [61].

RSTREAM(address, # bytes), RSTREAM_EX(address, # bytes): These primitives are generalizations of the prefetch mechanism for use in receiver-initiated communication scenarios. Traditional prefetch operations work at a block level and attempt to bring the block to the issuing processor in a VALID or EXCLUSIVE mode. The RSTREAM and RSTREAM_EX primitives allow specifying the number of bytes that need to be prefetched in addition to shared or exclusive mode of access. Semantically these primitives work as though the processor issued multiple block level prefetch requests, but the optimization comes in being able to bundle multiple prefetch requests into one.

There is very little hardware cost that will be incurred in implementing these primitives [61]. Preliminary results from simulation studies on SPASM also indicate performance benefits with these primitives [61]. An extensive description of the issues involved in implementing these primitives and performance evaluation is beyond the scope of this thesis and the reader is referred to a related thesis [67] for further details.

7.4 Summary

Scalability studies of parallel applications (Chapter 4) illustrated the need for architectural solutions to the communication overhead problem. In this chapter, we presented a solution to this problem by examining the communication characteristics of some applications and deriving appropriate mechanisms that should be supported in the architecture towards reducing/tolerating the communication overhead. We tried to exploit the benefits of sender-initiated communication using application knowledge and information available at runtime to find out the next consumer(s) for a data item. The architectural mechanisms co-exist with the underlying cache coherence protocol without having to burden the programmer with coherence maintenance.

Our evaluation framework plays an important role in conducting the exercise undertaken in the chapter. The application-driven approach helped identify typical communication scenarios and derive a set of mechanisms that can optimize these scenarios. Chapter 4 serves as the motivation for this study clearly indicating the performance benefits that we may hope to gain by architectural solutions to the communication overhead problem. A related study [69] develops a realistic model for a shared memory machine, and using SPASM shows that for a spectrum of applications almost all the inherent communication in them may be overlapped with computation.

The primitives proposed in this chapter strive to achieve this goal by pushing the delivered speedup curve as close as possible to the achievable curve (Figure 1). The profiling capabilities of SPASM are invaluable in understanding the communication behavior of applications and identifying the scenarios outlined in section 7.2.

Chapter 8

Concluding Remarks and Future Work

If the facts don't fit the theory, change the facts.

- *Albert Einstein*

I never think of the future. It comes soon enough.

- *Albert Einstein*

Evaluating and analyzing the performance of an application on an architectural platform has widespread applicability in parallel systems research. But conducting such a study is hard because of the complex interaction between application characteristics and architectural features. In this thesis, we have presented a framework for conducting evaluations and have used the framework to answer interesting questions about parallel systems.

8.1 Contributions

The primary contribution of this thesis is a framework for evaluating parallel systems that encompasses a set of metrics, techniques and tools. To address the inadequacy of existing metrics in providing sufficient information about parallel system performance, we have identified a set of metrics called overhead functions. An overhead function tracks the growth of an overhead which limits system scalability as a function of some system parameter. Experimenting on the actual hardware, analytical modeling, and simulation are three well-known performance evaluation techniques that can be used for quantifying overhead functions but each has its own drawbacks.

Our approach uses a combination of the three benefiting from the realism of experimentation on actual hardware, the detailed statistics obtainable from simulation, and the convenience of analytical models. At the heart of the framework is a simulation tool called SPASM which can model the execution of applications on a range of parallel hardware platforms. The novel feature of this simulator is its capability to isolate and quantify the overhead functions.

The other contributions of this thesis are a consequence of the exercises in using the framework to answer the questions raised in Chapter 1. The novel methodologies employed in each exercise have provided interesting results (over a chosen suite of applications) that are summarized below:

- Separation and quantification of parallel system overheads have provided key insights on system scalability for the chosen application-architecture pairs. Identifying application and architectural bottlenecks have been invaluable in restructuring the application or for suggesting architectural enhancements towards improving performance (see Chapter 4).

- Execution-driven simulators demand considerable resources in terms of space and time, and it is worthwhile to investigate if a sufficiently abstract simulation model of a parallel system can ease these demands without significantly compromising on accuracy. In this thesis, we have focused on abstractions for the interconnection network and the locality properties of parallel systems and have evaluated their validity (see Chapter 5). We have chosen the LogP [20] model for abstracting the interconnection network and have proposed an ideal cache for abstracting the locality properties. Separation of parallel system overheads has shown that the latency parameter in the LogP model is fairly accurate while the parameter which models network contention is pessimistic in many applications. The ideal cache, which models a coherent cache at each node in the machine without modeling the overheads associated with coherence maintenance, suffices to capture the locality properties over the chosen applications.

A simulation model using these abstractions is around 250-300% faster which can represent a substantial saving since execution-driven simulators are very time-consuming.

- Studying the scalability of the chosen applications has shown that the communication overhead resulting from a limited network bandwidth is a significant bottleneck limiting performance on the architectural side. If the network bandwidth is sufficiently increased, then the communication overhead is not expected to manifest as a significant portion of the total execution time. Varying a range of system parameters such as the network bandwidth, number of processors, CPU clock speed, and application problem sizes on SPASM, we have quantified the bandwidth needed to limit network overheads within a tolerable level (see Chapter 6). For some applications, existing link bandwidth technologies of 200-300 MBytes/sec would suffice since the communication overhead is not significant in these applications. Other applications demand considerable network bandwidth, but one may still be able to tolerate the communication overhead in these applications provided the problem sizes are increased commensurately. Further, our analysis has quantified the rate at which application problem sizes need to grow as a function of other system parameters in order to keep communication overhead within a tolerable factor.

- Owing to resource limitations, it may not be possible to provide the network bandwidth demanded by applications. To lower communication overhead in such circumstances, there is a need for minimizing or tolerating network accesses. By examining the communication characteristics of each application, we have identified typical communication scenarios and have derived a set of architectural mechanisms that are beneficial in these scenarios (see Chapter 7). These primitives co-exist with the underlying cache coherence protocol and try to accommodate sender-initiated data transfer wherever possible using information available at compile-time and/or run-time.

8.2 Future Work

There are several interesting avenues for future research which broadly fall in two categories: improving the framework itself and extending each of the case-studies undertaken in this thesis; and undertaking the task of answering other interesting questions about parallel systems.

Extending on the issues addressed in this thesis, some interesting directions for future work are given below:

- In the first place, we would like to broaden the spectrum of applications to encompass more diverse domains such as combinatorial optimization, graphics, image understanding, and databases. In addition, there are new application domains such as data fusion and molecular biology that could benefit from parallel computing. Apart from solving the computationally demanding problems in these domains, we would also be able to get a wider spectrum of realistic input models to drive computer systems design.

- Chapter 4 has illustrated the use of SPASM in performance debugging. Monitoring, animation and visualization support can further enhance the benefits from SPASM.

- As we observed earlier, we need to ease the resource demands of execution-driven simulators. One way is by abstracting details in the simulation model. We have investigated this idea in Chapter 5 by choosing specific abstractions for the interconnection network and locality properties of a parallel system. There is plenty of scope for improving these abstractions or even developing newer abstractions for different system features. Another way of easing simulation demands is by parallelizing the simulator itself. Preliminary work [68, 62, 14] in this area has addressed issues in parallelizing the simulation of specific features of a parallel system such as the cache [68, 62] and interconnection network [62, 14]. We need to further explore optimistic and conservative techniques [31] for simulating parallel systems in their entirety.

- In Chapter 6, we have quantified the link bandwidth needed by applications on a binary hypercube topology. We have also identified one way of projecting these requirements for other topologies. But this scheme is pessimistic since it does not take application characteristics into account. We need a better technique to project bandwidth requirements for other network topologies, especially in the context of an execution-driven simulator.

- Broadening the spectrum of applications would enable us to augment the set of communication primitives identified in Chapter 7. There are issues related to compile-time support to incorporate these primitives without programmer intervention. We also need to compare how these mechanisms perform with respect to other latency hiding techniques such as multi-threading and prefetching.

Several other interesting questions about parallel systems remain unanswered at this point, and the evaluation framework may be used to address these questions. We have not concentrated on software interaction overheads such as scheduling, memory management, and software message-passing in this thesis. Networks of workstations are becoming increasingly important as parallel computing engines [55]. These platforms use off-the-shelf hardware components making it difficult to provide fancy mechanisms for applications in the hardware. In such environments, software interaction overheads would play an important role in realizing high-performance computing. Applications are bound to get larger in the future making them increasingly reliant on tertiary storage. Further, new application domains such as multimedia and visual computing demand high speed networks, graphics engines and other novel peripheral devices. Input-output support would thus be a critical issue in the near future. The framework developed in this thesis provides sufficient infrastructure to study these issues and to quantify/qualify the support that is needed from an application viewpoint.

Apart from driving systems design, the framework has been extensively used for performance-conscious application program development. The framework has also been invaluable as an educational tool in parallel computing to illustrate issues in parallel computer architecture, systems software and performance evaluation.

Appendix A

Application Characteristics

We use a set of five applications with diverse characteristics for the evaluation studies in this thesis. EP, IS and CG are from the NAS [11] benchmark suite, CHOLESKY is from the SPLASH [71] benchmark suite, and FFT is a well-known 1-D Fast Fourier Transform algorithm. In the following discussion, we describe each application in terms of the data access pattern, the communication pattern, the computation granularity (the amount of work done) and data granularity (the volume of data communicated) for each phase of the program.

EP

Table 24: Characteristics of 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

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 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, 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 24 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 operation in "particle method" codes. An implementation of the algorithm is described in [60] and Table 25 summarizes its characteristics. The input list of size n 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 $chunk$ =

Table 25: Characteristics of 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	$nbuckets$ integers	Lock each bucket
8	Local List Ranking	Small	N/A	N/A

$nbuckets/p$ 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 all other processors 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 [9]) 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

Table 26: Characteristics of 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

FFT is a one dimensional complex Fast Fourier Transform of n 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. Communication is only incurred in the middle phase during which the *cyclic* layout of data is changed to a *blocked* layout as described in [20]. 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

Table 27: Characteristics of 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

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 27 summarizes the characteristics for each iteration of CG.

CHOLESKY

Table 28: Characteristics of 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

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 [71] 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.

Bibliography

Copy from one, it's plagiarism; copy from two, it's research.

- Wilson Mizner

[1] S. V. Adve and M. D. Hill. Weak Ordering - A New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[2] 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.

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

[4] A. Agarwal et al. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, June 1995.

[5] A. Agarwal, B-H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, 1990.

[6] 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.

[7] 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.

[8] R. Alverson et al. The Tera Computer System. In *Proceedings of the ACM 1990 International Conference on Supercomputing*, pages 1–6, Amsterdam, Netherlands, 1990.

[9] 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.

[10] J. Archibald and J-L. Baer. Cache Coherence Protocols : Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

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

[12] T. Blank. The MasPar MP-1 Architecture. In *Proceedings of COMPCON SPRING 90 - The 35th IEEE Computer Society International Conference*, pages 20–24, San Francisco, California, Spring 1990.

[13] 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.

[14] D. D. Burger and D. A. Wood. Accuracy vs. performance in parallel simulation of interconnection networks. In *Proceedings of the Ninth International Parallel Processing Symposium*, April 1995.

[15] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. Directory-based cache coherence in large-scale multiprocessors. *IEEE Computer*, 23(6):49–57, June 1990.

[16] 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.

[17] 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.

[18] Cray Research, Inc., Minnesota. *The Cray T3D System Architecture Overview Manual*, 1993.

[19] M. E. Crovella and T. J. LeBlanc. Parallel Performance Prediction Using Lost Cycles Analysis. In *Proceedings of Supercomputing '94*, November 1994.

[20] 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.

[21] D. E. Culler et al. Parallel Programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, November 1993.

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

[23] 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.

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

[25] W. J. Dally et al. The J-Machine : A Fine-Grain Concurrent Computer. In G. X. Ritter, editor, *Information Processing 89*, pages 1147–1153. Elsevier Science Publishers B. V. (North-Holland), 1989.

[26] 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.

[27] M. D. Dikaiakos, A. Rogers, and K. Steiglitz. FAST: A Functional Algorithm Simulation Testbed. In *Proceedings of MASCOTS '94*, pages 142–146, February 1994.

[28] M. Dubois, C. Scheurich, and F. Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, June 1986.

[29] T. H. Dunigan. Kendall Square Multiprocessor : Early Experiences and Performance. Technical Report ORNL/TM-12065, Oak Ridge National Laboratory, Oak Ridge, TN, March 1992.

[30] 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.

[31] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[32] K. Gharachorloo, D. Lenoski, J. Laudon, P.Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, 1990.

[33] 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.

[34] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W-D. Weber. Comparative evaluation of latency reducing and tolerating techniques. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 254–263, May 1991.

[35] A. Gupta and V. Kumar. The scalability of FFT on parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(8):922–932, August 1993.

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

[37] M. Heinrich et al. The performance impact of flexibility in the Stanford FLASH Multiprocessor. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

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

[39] Intel Corporation, Oregon. *Paragon User's Guide*, 1993.

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

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

[42] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B-H Lim. Integrating message-passing and shared memory: Early experience. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 54–63, May 1993.

[43] V. Kumar and A. Gupta. Analyzing Scalability of Parallel Algorithms and Architectures. *Journal of Parallel and Distributed Computing*, 22(3):379–391, September 1994.

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

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

[46] L. Lamport. How to make a Multiprocessor Computer that Correctly executes Multiprocess Programs. *IEEE Transactions on Computer Systems*, C-28(9), 1979.

[47] J. Lee and U. Ramachandran. Synchronization with Multiprocessor Caches. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 27–37, 1990.

[48] 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.

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

[50] 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.

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

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

[53] T. Mowry and A. Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[54] H. Nilsson, P. Stenstrom, and M. Dubois. Implementation and evaluation of update-based cache protocols under relaxed memory consistency models. Technical report, Dept. of Comp. Eng., Lund Univ., July 1993.

[55] The First Network of Workstations Workshop, San Jose, CA, October 1994.

[56] D. Nussbaum and A. Agarwal. Scalability of Parallel Machines. *Communications of the ACM*, 34(3):57–61, March 1991.

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

[58] David A. Patterson. Reduced Instruction Set Computers. *Communcations of the ACM*, 28(1):8–21, January 1985.

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

[60] 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.

[61] U. Ramachandran, G. Shah, A. Sivasubramaniam, A. Singla, and I. Yanasak. Architectural mechanisms for explicit communication in shared memory multiprocessors. In *Proceedings of Supercomputing '95*, December 1995. To appear.

[62] 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.

[63] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–336, April 1994.

[64] Kendall Square Research. Technical summary, 1992.

[65] 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.

[66] SAS Institute Inc., Cary, NC 27512. *SAS/STAT User's Guide*, 1988.

[67] G. Shah. *Architectural Mechanisms for Software Directed Management of Coherent Caches*. PhD thesis, College of Computing, Georgia Institute of Technology, Atlanta, GA, August 1995.

[68] G. Shah, U. Ramachandran, and R. Fujimoto. Timepatch: A novel technique for the parallel simulation of multiprocessor caches. In *Proceedings of the ACM SIGMETRICS 1995 Conference on Measurement and Modeling of Computer Systems*, pages 315–316, May 1995.

[69] G. Shah, A. Singla, and U. Ramachandran. The quest for a zero overhead shared memory parallel machine. In *Proceedings of the 1995 International Conference on Parallel Processing*, August 1995.

[70] 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.

[71] 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.

[72] A. Sivasubramaniam, U. Ramachandran, and H. Venkateswaran. A Computational Model for Message-Passing. In *Proceedings of the Sixth International*

Parallel Processing Symposium, pages 358–361, Beverly Hills, California, March 1992.

[73] A. Sivasubramaniam, U. Ramachandran, and H. Venkateswaran. A comparative evaluation of techniques for studying parallel system performance. Technical Report GIT-CC-94/38, College of Computing, Georgia Institute of Technology, September 1994.

[74] A. Sivasubramaniam, G. Shah, J. Lee, U. Ramachandran, and H. Venkateswaran. Experimental Evaluation of Algorithmic Performance on Two Shared Memory Multiprocessors. In Norihisa Suzuki, editor, *Shared Memory Multiprocessing*, pages 81–107. MIT Press, 1992. Also in *Proceedings of the First International Symposium on Shared Memory Multiprocessing*, April 1991.

[75] 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.

[76] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. A Simulation-based Scalability Study of Parallel Systems. *Journal of Parallel and Distributed Computing*, 22(3):411–426, September 1994.

[77] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. Abstracting network characteristics and locality properties of parallel systems. In *Proceedings of the First International Symposium on High Performance Computer Architecture*, pages 54–63, January 1995.

[78] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. On characterizing bandwidth requirements of parallel applications. In *Proceedings of the ACM SIGMETRICS 1995 Conference on Measurement and Modeling of Computer Systems*, pages 198–207, May 1995.

[79] P. Stenstrom. A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, 23(6):12–24, June 1990.

[80] H. Sullivan and T. R. Bashkow. A large scale, homogenous, fully-distributed parallel machine. In *Proceedings of the 4th Annual Symposium on Computer Architecture*, pages 105–117, March 1977.

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

[82] V. S. Sunderam. PVM : A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[83] Thinking Machines Corporation, Cambridge, Massachusetts. *The Connection Machine CM-200 Series Technical Summary*, June 1991.

[84] Thinking Machines Corporation, Cambridge, Massachusetts. *The Connection Machine CM-5 Technical Summary*, October 1991.

[85] T-F. Tsuei and M. K. Vernon. Diagnosing parallel program speedup limitations using resource contention models. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages I–185–189, August 1990.

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

[87] J. E. Veenstra and R. J. Fowler. MINT: A Front End for Efficient Simulation of Shared Memory Multiprocessors. In *Proceedings of MASCOTS '94*, pages 201–207, February 1994.

[88] 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.

[89] W-D. Weber and A. Gupta. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 273–280, June 1989.

[90] S. C. Woo, J. P. Singh, and J. L. Hennessy. The performance advantages of integrating block data transfer in cache-coherent multiprocessors. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[91] 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.

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

Vita

Anand Sivasubramaniam was born on 30th October 1967 in Madras, India. Being the only child of his parents who were both medical practitioners, he was raised with a lot of attention and affection. Since his school days, Anand had a keen acumen in finding out how things worked which may have very well ended his career early in childhood at the age of 3 when he received an electric shock while experimenting with a live electric socket. But the divine lord intervened and decided that Anand should pursue a career in Engineering much to the chagrin of his parents. Feeling that computers will become an integral component of everyday life, Anand chose to specialize in Computer Science. After spending 14 years in the same school, he embarked on his undergraduate study in Computer Science at the Indian Institute of Technology (IIT), Madras.

Anand's four year stint at IIT taught him the finer things in life exposing him to a diverse spectrum of cultures, opinions and eccentricities. On the academic front, he learned the rudiments of Computer Science and his early inquisitiveness in finding out how things functioned persisted motivating him to focus on the areas of Computer Architecture and Operating Systems. He was awarded a B.Tech in Computer Science in 1989. Complying with the norm in those days, he applied to several schools in USA for graduate study and was admitted to four with full financial assistance. He chose Georgia Tech from this list since he felt that the architectural/systems research at Tech was more up his alley.

Anand landed on American soil for the first time in September 1989. Since then, it has taken him nearly six years of toil and fun to complete this dissertation. Along the way, he picked up an MS in Computer Science from Georgia Tech apart from getting married. After considering several options, Anand felt that it would be better to pursue an academic route at this point in his career prefering the freedom of research and the opportunity to work with students. As of writing this document, he is joining the Computer Science and Engineering department at Pennsylvania State University as Assistant Professor in Fall 1995.