In *Proceedings of the First International Workshop on Parallel Processing*, pages 399-404, Bangalore, India, December 1994.

Issues in Understanding the Scalability of Parallel Systems[*]

Umakishore Ramachandran *H. Venkateswaran* *Anand Sivasubramaniam* *Aman Singla*

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280.
{*rama, venkat, anand, aman*}*@cc.gatech.edu*

(Extended Abstract)

1 Introduction

Scalability is a term frequently used to qualify the match between an algorithm and architecture in a parallel system (an algorithm-architecture combination). 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, predict the performance of an application on a larger configuration of an existing architecture, 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 understand the scalability of other application-architecture pairs. But evaluating and predicting the scalability of parallel systems poses several problems due to the complex interaction between application characteristics and architectural features. In this paper, we propose an approach for evaluating the scalability of parallel systems and develop a framework for studying the inter-play between applications and architectures. Using this framework, we study the scalability of five parallel scientific applications on shared memory platforms with three different network topologies. We illustrate the power of this framework in addressing two related issues. First, we use it to evaluate abstractions of parallel systems that have been proposed for modeling parallel system behavior. Second, we show its important use in synthesizing architectural requirements from an application perspective.

Since real-life applications set the standards for computing, our approach uses such applications for studying the scalability of parallel systems. We call such an application-driven approach a *top-down approach to scalability study*. The main thrust of this approach is to identify important algorithmic and architectural artifacts that impact the performance of a parallel system, understand the interaction between them, quantify the impact of these artifacts on the execution time of an application, and use these quantifications in studying the scalability of the system. We associate an *overhead function* with each algorithmic and architectural artifact that impedes the performance of a parallel system. We isolate and quantify the algorithmic overheads such as serial fraction and work-imbalance from the overall execution time of an application. We also isolate other overheads such as network *latency* (the actual hardware transmission time in the network) and network *contention* (the amount of time spent waiting for a resource to become free in the network) arising from the interaction of the algorithm with the underlying hardware. Our approach uses a combination of experimentation, simulation and analytical techniques in quantifying these overheads.

Traditional performance metrics such as speedup [1], scaled speedup [8], sizeup [23], experimentally determined serial fraction [9], and isoefficiency function [10], are useful for tracking performance trends, but they do not provide adequate information needed to understand the reason why an application does not scale well on an architecture. The overhead functions that we identify, separate, and quantify, help us overcome this inadequacy. The growth of overhead functions as a function of system parameters can provide key insights on the scalability of a parallel system by suggesting application restructuring, as well as architectural enhancements. Crovella and LeBlanc [6] follow a similar approach towards quantifying cycles that are lost due to different overheads in a parallel system using experimentation. Our approach uses simulation to isolate parallel system overheads. The importance of simulation in capturing the dynamics of parallel system interactions has been addressed in [17, 14, 13, 4, 5].

This work is part of an on-going project which aims at understanding the significant issues in the design of scalable parallel systems using the above-mentioned top-down approach. In our earlier work, we studied issues such as task granularity, data distribution, scheduling, and synchronization, by implementing frequently used parallel algorithms on shared memory [18] and message-passing [16] platforms. In [21], we illustrate the top-down approach for the scalability study of message-passing systems. In [20], we conduct a similar study for shared memory systems. The utility of the framework in evaluating machine abstractions and synthesizing network requirements are presented in [19] and [22] respectively.

The top-down approach and the overhead functions are elaborated in Section 2. The different ways of implementing this approach and details of a simulation platform, SPASM (Simulator for Parallel Architectural Scalability Measurements), which quantifies these overhead functions, are also discussed in this section. Using a set of five parallel applications and three hardware platforms, we summarize the use of our framework in studying the scalability of parallel systems (section 3.1), evaluating the validity of abstractions (section 3.2) and synthesizing network requirements from an application perspective (section 3.3). Concluding remarks are presented in Section 4.

2 Top-Down Approach

Adhering to the RISC ideology in the evolution of sequential architectures, we would like to use *real world applications* in the performance evaluation of parallel machines. However, applications normally tend to contain large volumes of code that are not easily portable and a level of detail that is not very familiar to someone outside that application domain. Hence, computer scientists have traditionally used parallel algorithms that capture

---

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

of parallel system artifacts thus derived may also be used to abstract features in the application and simulated hardware to speed up the simulation.

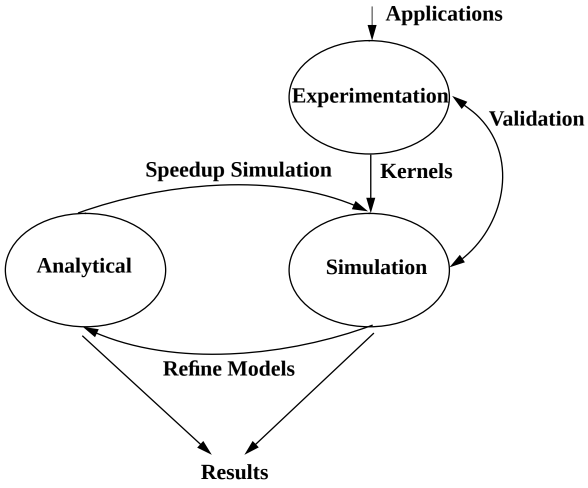

Figure 2: Framework

At the heart of our framework lies a simulation platform called SPASM, that provides an elegant set of mechanisms for quantifying the different overheads. Details of this simulation platform in the context of simulating shared memory platforms are presented in the next subsection. The reader is referred to [21] for the capabilities of SPASM in simulating message-passing platforms.

2.2 SPASM

SPASM is an execution-driven simulator written in CSIM [12]. As with other recent simulators [4, 5, 13], the bulk of the instructions in the parallel program is executed at the speed of the native processor (SPARC in this study) and only the instructions (such as LOADS and STORES) that may potentially involve a network access are simulated. The input to the simulator are parallel applications written in C. These programs are pre-processed (to label shared memory accesses), the compiled assembly code is augmented with cycle counting instructions, and the assembled binary is linked with the simulator code. The system parameters that can be specified to SPASM are: the *number of processors (p)*, the *clock speed* of the processor, the *network topology*, the *hardware bandwidth* of the links in the network, and the *switching delays*.

2.2.1 Metrics

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

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

SPASM also quantifies the different components of the interaction overhead. 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. These overheads may be broadly classified as latency and contention, and we associate an overhead function with each. The *Latency Overhead Function* is thus defined as the total amount of time spent by a processor waiting for messages due to the transmission time on the links and the switching overhead in the network assuming that the messages did not have to contend for any link. Likewise, the *Contention Overhead Function* is the total amount of time incurred by a processor due to the time spent waiting for links to become free by the messages. SPASM quantifies both the latency overhead function as well as the contention overhead function seen by a processor. This is done by time-stamping messages when they are sent. At the time a message is received, the time that the message would have taken in a contention free environment is charged to the latency overhead function while the rest of the time is accounted for in the contention overhead function. Though not relevant to this study, it is worthwhile to mention that SPASM provides the latency and contention incurred by a message as well as the latency and contention that a processor may choose to see. Even though a message may incur a certain latency and contention, a processor may choose to hide all or part of it by overlapping computation with communication. Such a scenario may arise with a non-blocking message operation on a message-passing machine or with a prefetch operation on a shared memory machine. But for the rest of this paper (since we deal with blocking load/store shared memory operations), we assume that a processor sees all of the network latency and contention.

Shared memory systems normally provide some synchronization support that is as simple as an atomic read-modify-write operation, or may provide special hardware for more complicated operations like barriers and queue-based locks. While the latter may save execution time for complicated synchronization operations, the former is more flexible for implementing a variety of such operations. For reasons of generality, we assume that only the test&set operation is supported by shared memory systems. We also assume that the memory module (at which the operation is performed), is intelligent enough to perform the necessary operation in unit time. With such an assumption, the only network overhead due to the synchronization operation (test&set) is a roundtrip message, and the overheads for such a message are accounted for in the latency and contention overhead functions described earlier. The waiting time incurred by a processor during synchronization operations is accounted for in the CPU time which would manifest itself as an algorithmic overhead.

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

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

- BARRIER: Mode corresponding to a barrier synchronization operation.

- MUTEX: Even though the simulated hardware provides only a test&set operation, mutual exclusion *lock* (implemented using test-test&set [2]) is available as a library function in SPASM. A program enters this mode during lock operations. With this mechanism, we can separate the overheads due to the synchronization operations from the rest of the program execution.

- PGM_SYNC: Parallel programs may use Signal-Wait semantics for pairwise synchronization. A lock is unnecessary for the Signal variable since only 1 processor writes into it and the other reads from it. This mode is used to differentiate such accesses from normal load/store accesses.

- NORMAL: A program is in the NORMAL mode if it is not in any of the other modes. An application programmer may further define sub-modes if necessary.

The *total time* for a given application is the sum of the *execution times* for each of the above defined modes. The *execution time* for each program mode is the sum of the *computation time*, the *latency overhead* and the *contention overhead* observed in the mode. Computation time in the NORMAL mode is the actual time spent in local computation in an application. The sum of latency and contention overheads in the NORMAL mode is the actual time incurred for ordinary data accesses. For the BARRIER and PGM_SYNC modes, the computation time is the wait time incurred by a processor in synchronizing with other processors that results from the algorithmic work imbalance. The computation time in the MUTEX mode is the time spent in waiting for 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 actual time incurred in accessing synchronization variables. The metrics identified by SPASM thus quantify the interesting components of the algorithmic and interaction overheads.

3 Uses of the Framework

In illustrating the use of our framework, we use a diverse range of applications and hardware platforms. Three of the applications (EP, IS and CG) are from the NAS parallel benchmark suite [3]; CHOLESKY is from the SPLASH benchmark suite [15]; and FFT is the well-known Fast Fourier Transform algorithm. EP and FFT are well-structured applications with regular communication patterns determinable at compile-time, with the difference that EP has a higher computation to communication ratio. IS also has a regular communication pattern, but in addition it uses locks for mutual exclusion during the execution. CG and CHOLESKY are different from the other applications in that their communication patterns are not regular (both use sparse matrices) and cannot be determined at compile time. While a certain number of rows of the matrix in CG is assigned to a processor at compile time (static scheduling), CHOLESKY uses a dynamically maintained queue of runnable tasks. For the underlying hardware, we use shared memory platforms with three different network topologies: a fully connected network, a binary hypercube and a 2-D mesh.

The framework described in Section 2 has widespread applicability. It can be used in performance debugging to identify application and architectural bottlenecks, suggesting application restructuring and architectural enhancements. It can be used to predict the performance of the application over a range of system parameters. The scalability of the above applications on the chosen hardware platforms is summarized in Section 3.1. The framework can be used to develop new analytical models, and to validate and refine existing analytical/theoretical models for parallel systems. In Section 3.2, we illustrate the use of the framework in validating models chosen for abstracting the network characteristics and locality properties of parallel systems. The framework can also help in synthesizing architectural requirements from an application viewpoint, which is very important for building well-balanced machines. In particular, the use of the framework in synthesizing network requirements for the chosen applications in presented in section 3.3.

3.1 Scalability Study of Parallel Systems

In [20] we illustrated the use of the framework in studying the scalability of the five parallel applications on the three simulated platforms. We separated and quantified the different overheads in the parallel system, and developed models to capture the growth of overheads with system parameters. The resulting overhead functions helped us identify and quantify the algorithmic and architectural bottlenecks in the parallel systems. The results from the study are summarized below.

EP displays a sufficiently high computation to communication ratio, with the computation time dominating over the latency and contention overheads in the network. Further, the algorithmic overheads in this application are negligible, resulting in a scalable execution with increasing processors across all hardware platforms.

Parallelization of IS increases the amount of work to be done for a given problem size. This inherent algorithmic overhead causes a deviation of the ideal curve from the linear curve, making the application unscalable for small problem sizes. On a fully connected network, the contention overhead is negligible and the latency converges to a constant with a sufficiently large number of processors. Thus, the scalability of this kernel on the fully connected network is expected to closely follow the ideal curve. For the hypercube the contention overhead grows logarithmically with the number of processors, while for the mesh this growth is linear, thus worsening the scalability of this application on these two platforms.

In FFT, the algorithmic overheads are marginal and the latency overhead decreases with increasing number of processors. Thus the contention overhead is the only artifact that can cause deviation from linear behavior. The communication in FFT is limited to a single phase where every processor communicates with every other processor. But these communication steps are skewed, and the network contention begins to show only on the mesh network where it grows linearly. FFT is thus scalable for the fully-connected and cube platforms. For the mesh platform, it would take 200 processors before the contention overhead starts dominating for a 64K problem size. Increasing the problem size improves its scalability on all three platforms.

For CG, the latency overhead decreases with increasing number of processors while 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.

CHOLESKY is not very scalable for the chosen problem size due to the inherent algorithmic overheads. Of the interaction overheads, latency decreases with increasing number of processors, making the contention component dictate the scalability of this application. The contention on the fully-connected and cube networks is negligible thus projecting speedup curves that closely follow the ideal speedup curve for these platforms. On the other hand, the contention grows logarithmically on the mesh making this platform less scalable.

Isolation and separation of the different overheads thus helped us identify and quantify application and architectural bottlenecks. Identifying such bottlenecks can suggest application restructuring and architectural enhancements. For instance, an initial implementation of IS exhibited a substantial contention overhead. An examination of the overhead functions over the course of the execution helped us restructure the implementation to reduce this overhead.

3.2 Validating Abstractions of Parallel Systems

Abstracting features of parallel systems is a technique often employed in performance analysis and algorithm development. For instance, abstracting parallel machines by theoretical models like the PRAM [24] has facilitated algorithm development and analysis. Such models try to hide hardware details from the programmer, providing a simplified view of the machine. Similarly, analytical models used in performance evaluation abstract complex system interactions with simple mathematical functions, parameterized by a limited number of degrees of freedom that are tractable. Abstractions are also useful in execution-driven simulators where details of the hardware and the application can be captured by abstract models in order to ease the demands on resource (time and space) usage in simulating large parallel systems. Some simulators [20, 4, 5, 13] already abstract details of instruction-set simulation, since such a detailed simulation is not likely to contribute significantly to the performance analysis of parallel systems.

An important question that needs to be addressed in using abstractions is their validity. Our framework serves as a convenient vehicle for evaluating the accuracy of these abstractions using real applications. In [19], we illustrate the use of the framework to evaluate the validity and use of abstractions in simulating the interconnection network and locality properties of parallel systems. An outline of the evaluation strategy and results are presented below.

For abstracting the interconnection network, we use the recently proposed *LogP* [7] 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. 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. Furthermore, if our abstraction closely models the behavior of a machine with a simple cache coherent protocol, then it would even more closely model the behavior of a machine with a fancier cache coherence protocol.

We use our simulation framework for evaluating these abstractions. We compare the results from simulating the five applications on a machine incorporating these abstractions with the results from an exact simulation of the actual hardware. Our results show that the latency overhead modeled by LogP is fairly accurate. On the other hand, the contention overhead modeled by LogP can become pessimistic for some applications since the model does not capture communication locality. The pessimism gets amplified as we move to networks with lower connectivity. With regard to the data locality question, results show that our ideal cache, which does not model any coherence protocol overheads, is a good abstraction for capturing locality over the chosen range of applications.

Apart from evaluating these abstractions in the context of real applications, the isolation and quantification of parallel system overheads helps us validate the individual parameters used in each abstraction. For instance, even when total execution time curves were similar, the latency and contention overheads helped us determine whether the LogP parameters were accurate in capturing the intended machine abstractions. The simulation of the system which incorporates these two abstractions is around 250-300% faster than the simulation of the actual machine. This factor can represent a substantial saving given that execution-driven simulation of real applications can take an inordinate amount of time. Using a similar approach, one may also use this framework to refine existing models (like reducing the pessimism in LogP in modeling contention), or even develop new models for accurately capturing parallel system behavior.

3.3 Synthesizing Network Requirements

For building a general-purpose parallel machine, it is essential to identify and quantify the architectural requirements necessary to assure good performance over a wide range of applications. Such a synthesis of requirements from an application view-point can help us make cost vs. performance trade-offs in important architectural design decisions. Our framework provides a convenient platform to study the impact of hardware parameters on application performance and use the results to project architectural requirements. We conducted such a study in [22] towards synthesizing the network requirements of the applications mentioned earlier, and the experimental strategy along with interesting results from our study are summarized here.

To quantify link bandwidth requirements for a particular network topology, we simulate the execution of the applications on such a topology and vary the bandwidth of the links in the network. As the bandwidth is increased, the network overheads (latency and contention) decrease, yielding a performance that is close to the ideal execution. From these results, we arrive at link bandwidths that are needed to limit network overheads (latency and contention) to an acceptable level of the overall execution time. We also study the impact of the number of processors, the CPU clock speed and the application problem size on bandwidth requirements. Computation to communication ratio tends to decrease when the number of processors or the CPU clock speed is increased, making the network requirements more stringent. An increase in problem size improves the computation to communication ratio, lowering the bandwidth needed to maintain an acceptable efficiency. Using regression analysis and analytical techniques, we extrapolate requirements for systems built with larger number of processors.

The results from the study suggest that existing link bandwidth of 200-300 MBytes/sec available on machines like Intel Paragon and Cray T3D can easily sustain the requirements of two applications (EP and FFT) even on high-speed processors of the future. For the other three, one may be able to maintain network overheads at an acceptable level if the problem size is increased commensurate with the processing speed.

The separation of the overheads plays an important role in synthesizing the communication requirements of applications. For instance, an application may have an algorithmic deficiency due to either a large serial part or due to work-imbalance, in which case 100% efficiency is impossible regardless of other architectural parameters. The separation of overheads enables us to quantify bandwidth requirements as a function of acceptable network overheads (latency and contention). The framework may also be used for synthesizing requirements of other architectural features such as synchronization primitives and locality capabilities from an application perspective.

4 Concluding Remarks

In this paper, we presented a novel approach for studying the scalability of parallel systems using real-world applications. Our approach uses a combination of experimentation, analytical modeling and simulation towards identifying, isolating and quantifying the different overheads in a parallel system that limit its scalability. We described an execution-driven simulation platform that can separate the interesting components of the algorithmic and interaction overheads from the overall execution time. Using a set of five parallel applications and three hardware platforms, we illustrated the use of our approach and simulation framework in 1) studying the scalability of these applications on the chosen hardware platforms; 2) evaluating the validity of parallel system abstractions; and 3) synthesizing network requirements from an application perspective.

References

[1] G. M. Amdahl. Validity of the Single Processor Approach to achieving Large Scale Computing Capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 483–485, April 1967.

[2] 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.

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

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

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

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

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

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

[13] 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.

[14] 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.

[15] 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.

[16] A. Sivasubramaniam, U. Ramachandran, and H. Venkateswaran. Message-Passing: Computational Model, Programming Paradigm, and Experimental Studies. Technical Report GIT-CC-91/11, College of Computing, Georgia Institute of Technology, February 1991.

[17] 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.

[18] 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.

[19] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. Abstracting network characteristics and locality properties of parallel systems. Technical Report GIT-CC-93/63, College of Computing, Georgia Institute of Technology, October 1993.

[20] 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.

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

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

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

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