MPI-SIM: USING PARALLEL SIMULATION TO EVALUATE MPI PROGRAMS

Sundeep Prakash
Rajive L. Bagrodia

University of California
Computer Science Department
Los Angeles, CA 90095, U.S.A.

ABSTRACT

This paper describes the design and implementation of MPI-SIM, a library for the execution driven parallel simulation of MPI programs. MPI-LITE, a portable library that supports multithreaded MPI is also described. MPI-SIM, which is built on top of MPI-LITE, can be used to predict the performance of existing MPI programs as a function of architectural characteristics including number of processors and message communication latencies. The simulation models can be executed sequentially or in parallel. Parallel executions of MPI-SIM models are synchronized using a set of asynchronous conservative protocols. MPI-SIM reduces synchronization overheads by exploiting the communication characteristics of the program that it simulates. The paper presents validation and performance results from the use of MPI-SIM to simulate applications from the NAS Parallel Benchmark suite. Using the techniques described in this paper, we were able to reduce the number of synchronizations in the parallel simulation as compared with the synchronous quantum protocol and were able to achieve speedups ranging from 3.2-11.9 in going from sequential to parallel simulation using 16 processors on the IBM SP2.

1 INTRODUCTION

Simulators for parallel programs can be used to test, debug and predict the performance of parallel programs for a variety of parallel architectures. Most existing simulators (Brewer et al 1991, Davis et al 1991, Covington et al 1991) use direct execution to simulate the sequential blocks of code, and simulate only the communication and/or I/O events. As sequential execution of such models (Legedza and Weihl 1996, Reinhardt et al 1993, Dickens et al 1994, Dickens et al 1996) are typically slow (slowdown factors of 2 to 15 per processor are not atypical), several researchers have used parallel execution of such models with varying degrees of success. The primary difficulty in obtaining better performance is the significant synchronization overhead in the parallel simulator.

In this paper we explore the use of a novel conservative synchronization algorithm for parallel simulation of message passing parallel programs. We combine the existing null message (Misra 1986) and conditional event (Chandy and Sherman 1989) protocols together with a number of optimizations to significantly reduce the frequency and cost of synchronizations in the parallel simulator. The optimized simulation protocol has been incorporated in a simulation library for MPI (MPI Forum 1993), called MPI-SIM. An existing MPI program may be linked with the MPI-SIM library (after an appropriate pre-processing stage described subsequently) to predict its performance as a function of the desired architectural characteristics; a programmer is not required to make any modifications to the original MPI program. This paper also presents the results of an experimental study to evaluate the utility of MPI-SIM in the simulation of the NAS Parallel Benchmark Suite.

2 MPI SIMULATION MODEL

2.1 MPI Overview and Core Functions

MPI (MPI Forum 1993) is a message passing library which offers a host of point-to-point and collective interprocess communication functions to a set of single threaded processes executing in parallel. All communication is performed using a **communicator**—which describes the group of communicating processes. Only member processes may use a given communicator. This paper assumes that the program does not have any I/O commands; simulation of the I/O constructs is described in Bagrodia et al (1997). In the subset of MPI we simulate, all collective communication functions are implemented in terms of point-to-point communication functions, and all point-to-point communication functions are implemented using a set of *core* non-blocking MPI functions. The core functions include MPI_Issend, a *non-blocking synchronous send*, MPI_Ibsend, a *non-blocking buffered send*, MPI_Irecv *non-blocking receive* and MPI_Wait.

The primary difference between the two sends is that the synchronous send completes only when the receiver has accepted the message using a matching receive; the buffered send completes as soon as the data has been copied to a local buffer. The buffer space is released only when the data has been transmitted to the receiver via a synchronous send. Each point-to-point MPI message carries a *tag* and the *sender-id.* A receive may be selective, accepting a message only from a given sender and/or with a given tag. Alternately, it may use the wild card arguments MPI_ANY_SOURCE or MPI_ANY_TAG to indicate that a message from any source process or with any tag value is acceptable. The wait is simply a function which blocks the process until the specified non-blocking (send or receive) operation has completed.

In this paper, we use the terms Target Program to refer to the MPI program whose performance is to be predicted, Target Machine as the machine on which the target program executes, Simulator as the program that simulates execution of the target program on the target machine, and Host Machine as the machine on which the simulator executes. In general the host machine may be sequential or parallel. For direct execution, it is important that the processor configurations in the host and target machine be similar.

2.2 Preprocessing MPI programs for MPI-SIM

In general, the host machine will have fewer processors than the target machine (for sequential simulation, the host machine has only one processor); this requires that the simulator provides the capability for multithreaded execution. As MPI programs execute as a collection of single threaded processes, it is necessary to provide a capability for multithreaded execution of MPI programs in MPI-SIM. We have developed MPI-LITE, a portable library to support multithreaded MPI programs. Executing an existing MPI program as a multithreaded program requires additional modifications. The primary one deals with transforming the permanent variable, *i.e.* global variables and static variables within functions. If the unmodified MPI program is executed as a multithreaded program, all threads on a given host process will access a single copy of each permanent variable. To prevent this, it is necessary to *privatize* the permanent variable such that each thread has a local copy. Each permanent variable is redeclared with an additional dimension whose size is equal to the maximum number of threads in a host process. Each reference to the permanent variable is also modified such that each thread uses its id to access its own copy of the permanent variable. This process of adding a dimension to the permanent variables is referred to as **privatization**. A preprocessor is provided with MPI-SIM that automatically privatizes permanent variables, converts each MPI call to the corresponding MPI-SIM call, and implements miscellaneous transformations needed to link the program with the MPI-SIM library. In MPI-SIM the routines for inter-thread communication are syntactically identical to those for inter-process communication except for the use of a special prefix to distinguish between the two.

2.3 Simulation Model for Core Functions

We present a model for execution and simulation of the four core functions. The simulation model defines a logical process (LP) for each process in the target program. Each LP, has a message queue for each communicator of which the LP is a member, a simulation clock, and an ordered list (ordered by simulation timestamp) of the pending (send and receive) operations of the LP; this list is referred to as the *request list*. Simulation of a process in the target program by a corresponding LP in the simulator proceeds as follows: sequential code blocks are simulated via direct execution. Each call to an MPI communication statement (collective or point-to-point) is translated to a call to the corresponding MPI-SIM function. MPI-SIM internally implements each call to a collective function in terms of the core communication commands described in Section 2.1. For brevity, we do not describe the translation in the paper; the reader is referred to Prakash (1996). We briefly describe the simulation of the core commands.

The sends in the MPI core are simulated by sending a message (with source, destination, tag, communicator and data) to the receiver LP. The message is timestamped with the send timestamp, which is the current simulation time of the sending LP and the receive timestamp, which is the send timestamp plus the predicted message latency. For buffered sends, the overheads and functionality for buffer availability check are included in the simulation. The simulation of MPI_Irecv simply adds a request to the request list. The action taken for the wait depends on the type of the specified operation. For instance, for wait on a receive operation, the LP is blocked until a matched message is available. Of course, the LP must remove messages in the order of their simulation timestamps and *not* in the order in which messages are physically deposited in its queue. When an appropriate matching message is removed, the LP's simulation clock is updated to the maximum of the current simulation time and the receive timestamp of the matching message, an acknowledgment is sent to the sender, and the LP is resumed. For the synchronous send operation, the LP blocks until the corresponding acknowledgment has been received from the destination. At this time, the simulation time of the LP is updated to the maximum of the current simulation time and the receive timestamp of the acknowledgment.

3 PARALLEL EXECUTION OF MPI SIMULATION MODEL

Two types of protocols have commonly been used in the parallel simulation of parallel programs: the synchronous or quantum protocol (e.g. SimOS (Rosenblum et al 1995, Rosenblum et al 1997)), and the asynchronous protocols (e.g. LAPSE (Dickens et al 1994)). In the synchronous protocol, each LP periodically simulates its corresponding process for a previously determined interval Q, termed the **simulation quantum**, and then executes a global barrier. These barriers are used to ensure that messages from remote LPs will be accepted in their correct timestamp order. An LP waiting at a receive will accept a matching message from its buffer only if *the receive timestamp of the message is less than the simulation time at which the current quantum terminates*. If more that one such message is present, the LP will select the one with the earliest timestamp; if no such messages are present, the LP remains blocked, and its simulation time is updated to the end of the current quantum. The synchronous protocol is guaranteed to be accurate only if $Q<L$, where L is the communication latency of the target architecture. However, a small Q implies frequent global synchronizations leading to poor performance. (If the host machine provides an efficient hardware implementation of global synchronization (e.g., CM5), it might be feasible to obtain good performance even with a small value of Q.) Simulation efficiency can be improved by using a larger quantum; however with $Q>L$, it is no longer possible to guarantee that the simulator is accurate. Thus parallel simulators (e.g. SimOS) that use this protocol offer two simulation modes: fast and inaccurate, or slow and accurate.

MPI-SIM uses an asynchronous protocol, which reproduces the communication ordering of the target program in the simulator. LPs have two attributes associated with them at all times: **Execution Status** (blocked, running or terminated) and **Simulation Status** (deterministic or non-deterministic mode). An LP is **blocked** if it has executed a receive statement and no matching message is available; otherwise it is said to be **running**. An LP is in deterministic mode *if every receive request in its request list explicitly specifies the source (i.e. no receive contains MPI_ANY_SOURCE as the source).* Each LP executes without synchronizing with other LPs until it gets blocked on some wait operation; a synchronization protocol is used to decide if the LP can or cannot proceed with a message from its buffer. We briefly describe our protocol.

Each LP in the model computes a local quantity called its Earliest Input Time or **EIT** (Jha and Bagrodia 1993). The EIT represents a lower bound on the receive timestamp of future messages that the LP may receive. Consequently, upon executing a wait statement, an LP can safely select a matching message with a receive timestamp less than its EIT. Different asynchronous protocols differ only in their method for computing EIT. Our implementation supports various protocols including the *Null Message Protocol (NMP)* (Chandy and J. Misra 1979), the *Conditional Event Protocol (CEP)* (Chandy and Sherman 1989), and a new protocol, which is a combination the two (Jha and Bagrodia 1993). Due to space limitations, we have omitted details of the protocol; the interested reader is referred to Prakash (1996).

The primary overhead in implementing parallel conservative protocols is due to the communications to compute EIT and the blocking suffered by an LP that has not been able to advance its EIT. We have suggested and implemented a number of optimizations to significantly reduce the frequency and strength of synchronization in the parallel simulator thus reducing unnecessary blocking in its execution. The primary optimizations include:

1. **Automatic detection of deterministic fragments in the parallel program.** In general, an LP is blocked either if its buffer does not contain a matching message or if the timestamp on the message is greater than the LP's EIT. However, an LP in the *deterministic* mode can proceed as soon as it finds a matching message, regardless of its EIT. This is an optimization within the framework of the null message protocol.
2. **Reducing blocking time of an LP by exploiting the communication characteristics of the application.** By precisely defining potential message sources, an LP can reduce the communications that are used to advance its EIT.
3. **Reducing the frequency of synchronization with dynamic extraction of lookahead.** Lookahead is the ability of an LP to predict lower bounds on future times at which it will *generate* a message for other LPs. Extracting tight estimates for each communicating partner leads to fewer synchronizations than the commonly used static methods for computing lookahead.

4 RESULTS

4.1 Benchmarks

We have validated MPI-SIM, and measured its performance for the NAS (Numerical Aerodynamic Simulation) Parallel Benchmarks (NPB 2) (Bailey et al 1995), a set of programs designed at the NASA NAS program to evaluate supercomputers. The IBM-SP2 at UCLA was selected as both the target and host machines. Each node of the IBM-SP2 is a POWER2 node with 128Kb of cache and 256Mb of main memory.

Table 1: NAS Benchmarks

			Target Procs..	Target 1	Target 2	Target 3
Names	Lines	Class		Target Program Size/Simulator Size (Host Procs. for Simulator)		
LU	4623	A	4,8,16	14M/57M (1,2,4)	8M/32M(2,4,8)	5M/18M(4,8,16)
MG	2712	S	4,8,16	600K/8M (1,2,4)	400K/5M (1,2,4,8)	300K/3M (1,2,4,8,16)
BT	6290	S	4,9,16	2M/24M (1,2,4)	1M/15M (1,2,4,9)	1M/12M (1,2,4,9,16)
SP	5555	S	4,9,16	700K/7M (1,2,4)	500K/6M (1,2,4,9)	500K/5M (1,2,4,9,16)

Nodes are connected using a high performance switch, which offers a point-to-point bandwidth of 40Mb/s, and has a hardware latency of 500ns. The NPB benchmarks are written in Fortran 77 with embedded MPI calls for communication. Since MPI-SIM currently supports privatization only for C programs, it was necessary to convert the benchmarks to C. We were able to convert four out of the five benchmarks using f2c (Feldman et al 1990), a Fortran-to-C converter. The specific configurations of the benchmarks that were used in the performance study were constrained primarily by their memory and CPU requirements. Table 1 summarizes the relevant configuration information for the benchmarks. Each benchmark was executed for three target machine configurations. For example, LU was executed on 4, 8 and 16 processors.

4.2 Verification & Validation

The target programs and the simulators were executed for all processor configurations listed in Table 1. For each target and host processor configuration, each simulator was executed in four modes described in Section 4.3. The NPB 2 benchmarks are self-verifying, meaning that each benchmark after completion compares the computed results against precomputed results to ensure that it executed correctly. All target programs and simulators were found to verify correctly.

Figure 1 plots the target program execution time (solid line) and the execution time as predicted by the simulator (dashed lines) as a function of various target machine configurations; note that the simulator predicted times are plotted for each host configuration listed in Table 1. The graphs were nearly identical in all simulator modes, and consequently the figure shows only one mode: the NMP+CEP+Det mode. In the best case the predicted and measured times differed by less than 5% and in the worst by 20% lending reasonable credibility to the simulations.

4.3 Simulator Modes

A simulator can be executed in four modes. In three of these the simulation status is non-deterministic, but

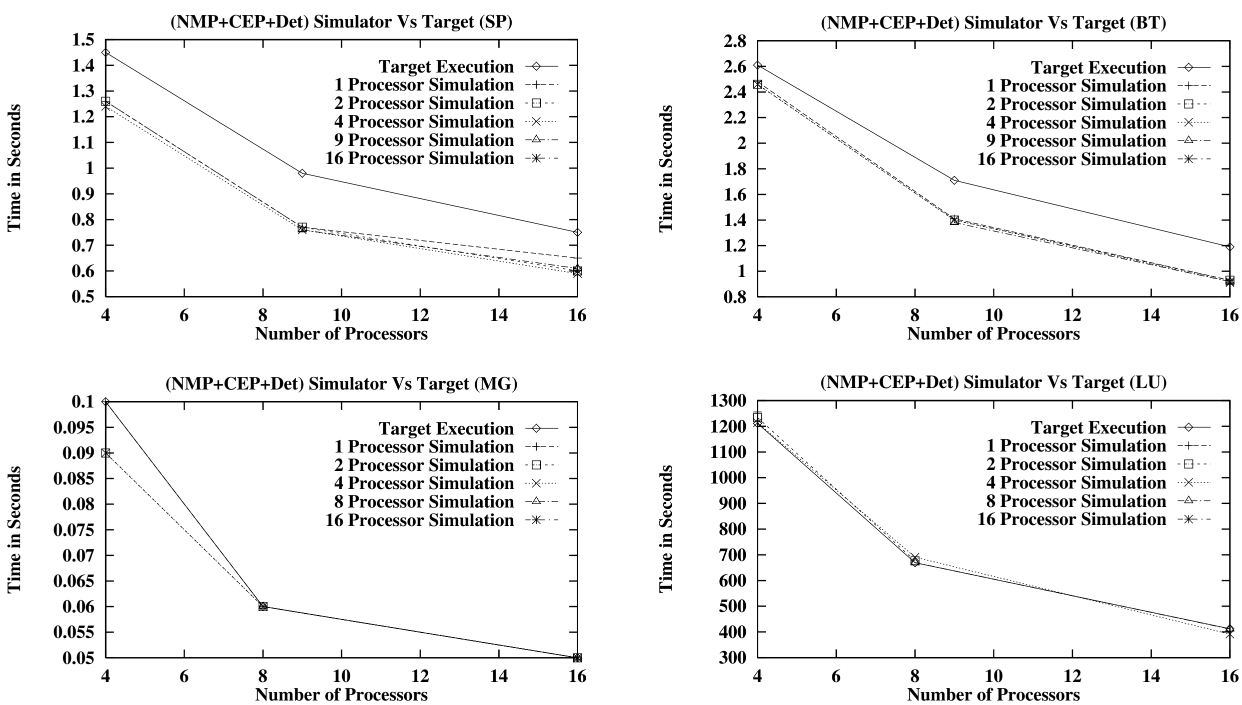

Figure 1: Target Execution Time vs. Simulator Predictions for NAS Benchmarks

they differ in the use of the protocol for EIT advancement. The CEP mode (uses conditional event protocol), NMP mode (uses null message protocol) and CEP+NMP mode (combines both). In the last mode the simulation status is deterministic and both the conditional event and the null message protocol (CEP+NMP+DET mode) are used. These simulator modes allow us to determine the contribution of each protocol and each optimization to the performance of the simulation.

4.4 Reducing Synchronizations

We compared all modes of each simulator against the traditional quantum protocol. Performance of the simulation protocol in each simulator mode is gauged by the number of rounds of protocol messages, R, sent per processor. The performance of the quantum protocol is measured as the number of global synchronizations it takes to simulate the same target program. A round of protocol messages is similar to a global synchronization, although it is frequently less expensive, since in many cases a processor does not need to wait to receive protocol messages from all other processors.

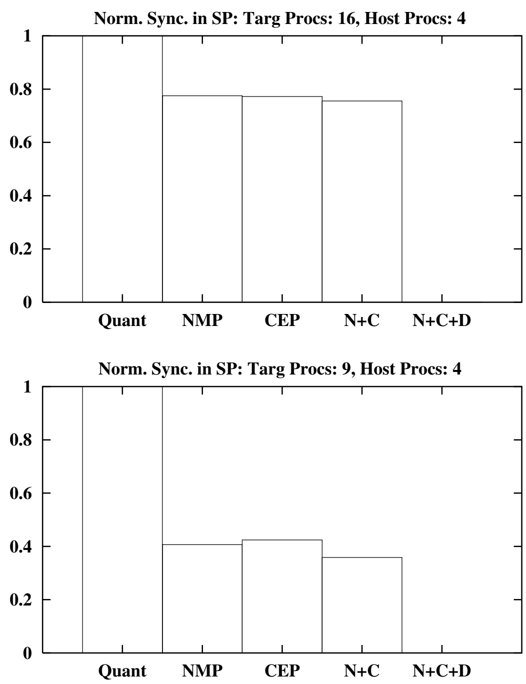

Figure 2: Performance of Simulators for SP

Given a target processor configuration, we found that R decreases only modestly as the number of host processors used to simulate the configuration is increased. Figures 2, 3, 4, and 5 show the variation of R with the simulator modes for two representative target and host processor configurations of each benchmark. In each graph, the number of rounds of protocol messages is normalized against the number of global synchronizations of the quantum protocol. The X-axis shows the simulator mode, where "N+C" refers to the NMP+CEP mode and the "N+C+D" mode refers to the NMP+CEP+Det mode.

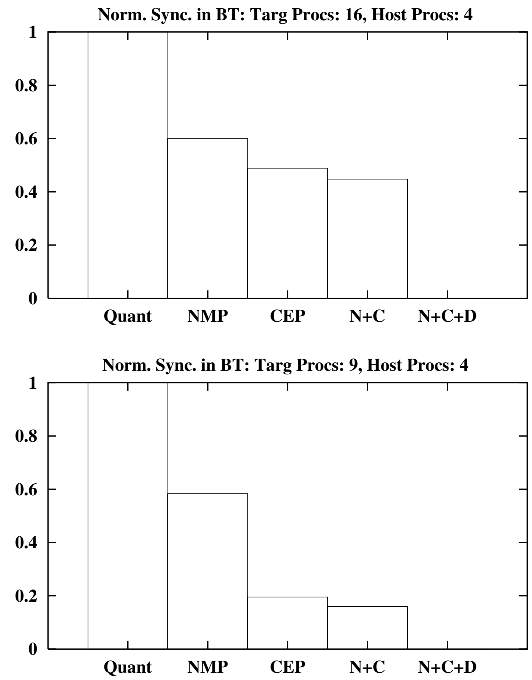

Figure 3: Perfomance for Simulators for BT

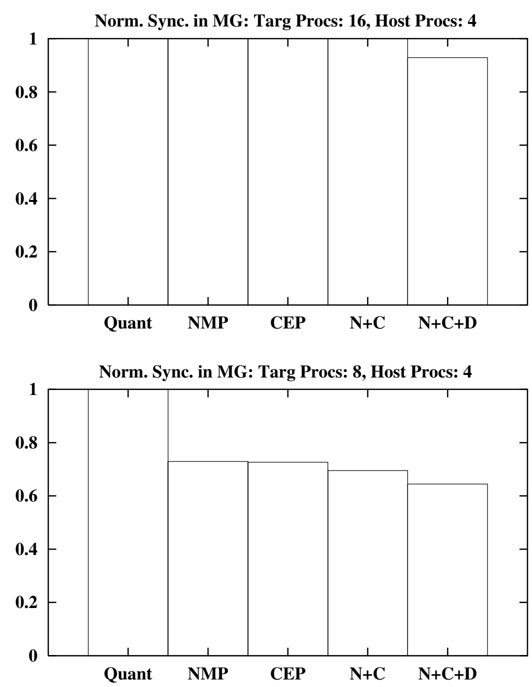

Figure 4: Performance of Simulators for MG

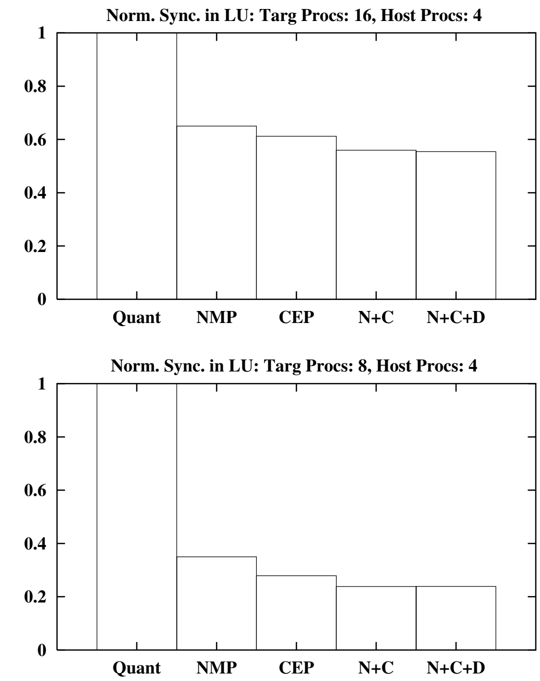

Figure 5: Performance of Simulators for LU

Consider only the CEP mode: the amount of improvement over the quantum protocol is strongly dependent on the average duration for which an LP (i.e. thread) executes before getting blocked. Table 2 shows this average duration for each benchmark and each target program configuration. L is the minimum message latency of the target machine. The 9-processor BT benchmark has the largest average uninterrupted execution time per thread, and in the simulation, the CEP mode is able to eliminate more than 80% of the global synchronizations of the quantum protocol. The NMP mode is able to eliminate only 40% of the global synchronizations of the quantum protocol. This is because the CEP significantly improves over the NMP when some LPs are far ahead of the others in simulation time, requiring the other LPs to exchange many rounds of null messages to update their simulation times. The 16-processor MG benchmark has the smallest average uninterrupted execution time per thread, and the NMP+CEP mode is unable to significantly reduce the number of global synchronizations of the quantum protocol.

Table 2: Average Uninterrupted Execution Time

Benchmark	16 Targ. Proc.	8 or 9 Targ. Proc.
LU	8.74L	11.77L
MG	2.79L	4.03L
BT	12.33L	24.81L
SP	4.61L	9.29L

Using our optimizations for exploiting the determinism in the program, we note that it is possible to eliminate *all global synchronizations* in the BT and SP benchmarks. The optimizations were not effective in significantly reducing the synchronizations from the MG and LU benchmarks as discussed in the next section.

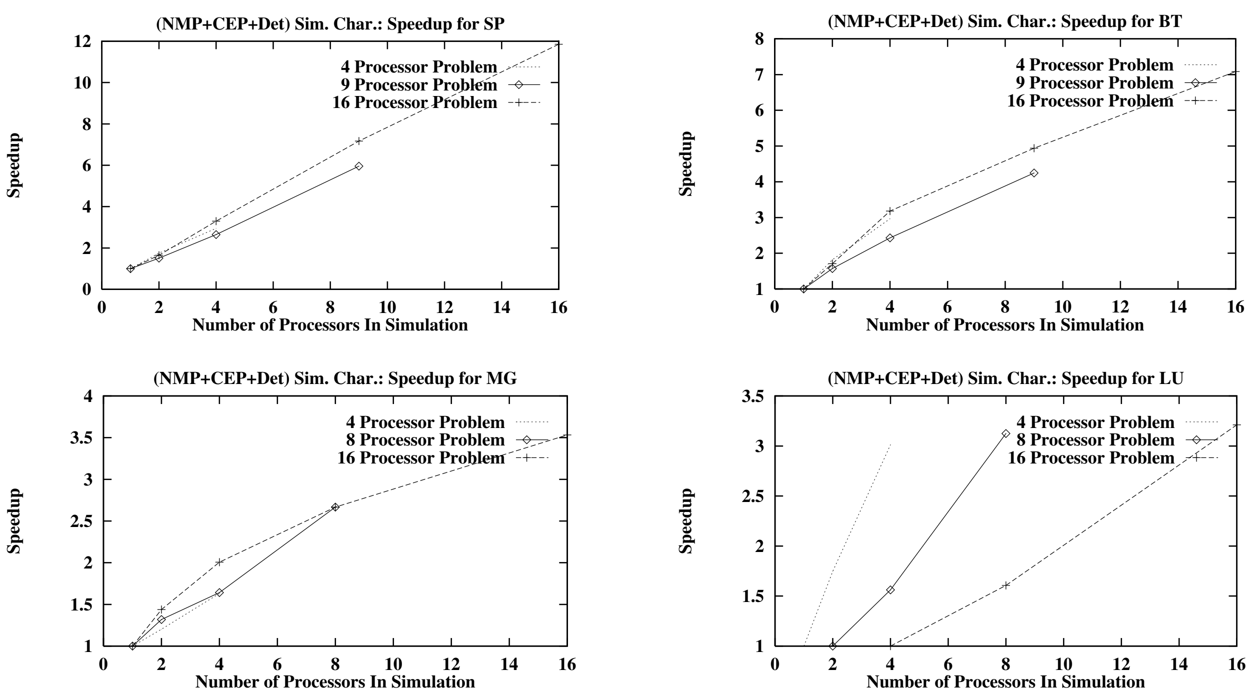

Figure 6: Fast Simulator Speedups

4.5 Reducing Simulator Execution Times

We present the speedup measured by executing the parallel simulator using the combined NMP and CEP algorithm as well as the deterministic protocol. A receive can be deterministic either if it specifies the source explicitly or it specifies an explicit tag and each source uses unique tags. Although the first type of determinism can be detected automatically by the current simulator, we have not yet implemented the second mode. Out of the four benchmarks used, SP and BT have the determinism of first type. The MG and LU benchmarks have determinism of second kind. Although this optimization is not automatically implemented in the compiler, we manually inserted the optimizations to evaluate the potential benefit that can be derived from exploiting this form of non-determinism. The final speedups obtained from the execution of all the benchmarks are presented in Figure 6. We measure speedup(N) by taking the ratio of the execution time of the sequential simulator to the execution time of the simulator using N processors. The speedups for the LU benchmarks are relative to the smallest host processor configuration that could be used to run the simulator. For example, the 8 target processor simulator could be executed on 2, 4 or 8 host processors. Hence, the reference execution time is of the 2-processor simulation. This *understates* the expected performance improvement for this application. Notice that the speedups achieved with the simulation are characteristic of the application itself, as the simulation overhead is relatively small.

5 RELATED WORK

Most simulation engines use sequential or parallel implementations of the quantum protocol. Among these are Proteus (Brewer et al 1991), a parallel architecture simulation engine, Tango (Davis et al 1991), a shared memory architecture simulation engine, Wisconsin Wind Tunnel (Reinhardt et al 1993), a shared memory architecture simulation engine and SimOS, a complete system simulator (multiple programs plus operating system). Two simulation engines which use approaches similar to ours are Parallel Proteus (Legedza and Weihl 1996) and LAPSE.

Parallel Proteus is the parallelization of the Proteus simulation engine, which uses the quantum protocol. The synchronization overhead caused by frequent barriers is reduced using two methods: (a) Predictive barriers and (b) Local barriers. Predictive barriers is a method for safely increasing the simulation quantum beyond L, the minimum communication latency of the target machine. This method uses runtime and compile time analysis to determine, at the beginning of a simulation quantum, the earliest simulation time at which any LP will send a message to any other LP. Consequently, the simulation quantum can be extended until that time. Runtime analysis involves simply running an LP until it communicates. If it stops at the equivalent of a receive statement, analysis performed at compile time is used to predict when it would have sent a message if it were instantly resumed. The method of local barriers uses statically available communication topology information (i.e. groups of LPs that communicate only within the groups they belong to) to reduce the global synchronization at the end of a simulation quantum to local synchronizations between groups of LPs.

LAPSE (Dickens et al 1994) is a parallel simulation engine for programs using the message passing library of the Intel Paragon. It uses a quantum protocol called WHOA (Window-based Halting On Appointments). Like Parallel Proteus, it uses runtime analysis to determine the size of the simulation quantum, but the runtime analysis is not supplemented with compile time analysis.

In comparison, we use the equivalent of runtime analysis since we execute an LP until it reaches a receive statement. The benefits of compile time analysis are achieved using the conditional event protocol, which is portable and does not need target instruction set analysis. In addition, our implementation of the null message protocol adapts automatically to the dynamically changing communication topology specified by the target program. Perhaps most importantly, it automatically recognizes (some forms of) deterministic code and switches off all synchronization while simulating it; automatic recognition of other forms of determinism are being added to the simulator. As seen in Section 4, this optimization helps us eliminate almost all the synchronization overhead in simulating many real applications.

6 CONCLUSION

In this paper, we have shown the usefulness of the null message and the conditional event protocols in the conservative parallel simulation of parallel programs. We have used application characteristics to optimize the performance of the null message protocol, and used the comparatively slower conditional event protocol only where the null message protocol fails. We have demonstrated that for deterministic sections of code, the simulation protocol can be bypassed completely without affecting the correctness of the simulation. These optimizations have been implemented in a simulation library (MPI-SIM) for a subset of MPI, an accepted standard for message passing parallel programs. MPI-SIM has been validated and shown to be fast for a subset of the NAS Parallel Benchmarks (NPB 2).

ACKNOWLEDGMENTS

This work was supported by the Advanced Research Projects Agency, DARPA/CSTO, under Contract F-30602-94-C-0273, “Scalable Systems Software Measurement and Evaluation” and by DARPA/ITO Contract N-66001-97-C-8533, “End-to-End Performance Modeling of Large Heterogenous Adaptive Parallel/Distributed Computer/Communication Systems.” All data was collected on the IBM SP2 at UCLA’s Office of Academic Computing, granted to UCLA by IBM Corporation under the Shared University Research Program.

REFERENCES

Brewer, E. A., C. N. Dellarocas, A. Colbrook, And W. E. Weihl., Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, Cambridge, MA 02139, 1991.

Bagrodia, R., S. Docy, and A. Kahn, Parallel Simulation of Parallel File Systems and I/O Programs. In *Supercomputing 97*, 1997.

Bailey, D., T. Harris, W. Saphir, R. V. D. Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. Technical Report Nas-95-020, Nasa Ames Research Center, Moffet Field, CA 94035-1000, December 1995.

Covington, R. G., S. Dwarkadas, J. R. Jump, J.B. Sinclair, and S. Madala. The Efficient Simulation of Parallel Computer Systems. *IJCS*, 1:31-58, 1991.

Chandy K. M., and J. Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering*, Pages 440-452, September 1979.

Chandy K. M., and R. Sherman. The Conditional Event Approach to Distributed Simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, Miami, Pages 93-99, 1989.

Davis, H., S. R. Goldschmidt, and Hennessey. Multiprocessor Simulation and Tracing Using Tango. In *Proceedings of ICPP '91*, Pages 99-107, August 1991.

Dickens, P., P. Heidelberger, and D. Nicol. A Distributed Memory Lapse: Parallel Simulation of Message-Passing Programs. In *Workshop on Parallel and Distributed Simulation*, Pages 32-38, July 1994.

Dickens, P. M., P. Heidelberger, and D.M. Nicol. Parallelized Direct Execution Simulation of Message-Passing Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 6(4):297-320, October 1996.

Feldman, S. I., D. M. Gay, Mark W. Maimone, And N. L. Schryer. A Fortran-To-C Converter. Technical Report No. 149, AT&T Bell Laboratories, Murray Hill, NJ, May 1990.

MPI Forum. MPI: A Message Passing Interface. In *Proceedings of 1993 Supercomputing Conference*, Portland, Washington, November 1993.

Fujimoto, R. Parallel Discrete Event Simulation. *Communications of The ACM*, 33(10):30-53, October 1990.

Jha V., and R. Bagrodia. Transparent Implementation of Conservative Algorithms In Parallel Simulation Languages. In *Winter Simulation Conference*, December 1993.

Legedza U., and W. E. Weihl. Reducing Synchronization Overhead in Parallel Simulation. In *Tenth Workshop on Parallel and Distributed Simulation PADS 96,* May 1996.

Misra, J. Distributed Discrete-Event Simulation. ACM Computing Surveys, 18(1):39-65, March 1986.

Prakash, S. Performance Prediction of Parallel Programs. Ph.D. Thesis, Computer Science Department, Ucla, Los Angeles, CA 90095, November 1996.

Rosenblum, M.. E. Begnion, S. Devine, and S. A. Herrod. Using The SimOS Machine Simulator to Study Complex Computer Systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1), January 1997.

Reinhardt, S. K.. M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of the 1993 ACM Sigmetrics Conference*, May 1993.

Rosenblum, M.. S. A. Herrod, Emmett Witchel, and A. Gupta. Complete Computer System Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology*. Vol. 3, No. 4, Winter 1995.

AUTHOR BIOGRAPHIES

SUNDEEP PRAKASH Sundeep Prakash received a B.Tech in Electrical Engineering from the Indian Institute of Technology, Delhi, India in 1989, an M.S. from the University of Florida in 1991 and a Ph.D in Computer Science from UCLA, Los Angeles in 1996. Since 1997, he has been a software engineer at TIBCO, Palo Alto. His research interests include algorithms for parallel and distributed simulation, compilation of parallel programs for shared and distributed memory machines, and messaging interfaces and protocols.

RAJIVE L. BAGRODIA is a professor in the Department of Computer Science at the University of California, Los Angeles. He holds an M.S. and Ph.D. in Computer Science from the University of Texas at Austin. His research interests include computer and communication networks, nomadic systems, and parallel languages.