# Maya: A Simulation Platform for Distributed Shared Memories

Divyakant Agrawal* Manhoi Choy† Hong Va Leong* Ambuj K. Singh†
Department of Computer Science
University of California at Santa Barbara
Santa Barbara, CA 93106

## Abstract

Maya is a simulation platform for evaluating the performance of parallel programs on parallel architectures. It allows the rapid prototyping of memory protocols with varying degrees of coherence and facilitates the study of the impact of these protocols on application programs. The design of Maya and its simulation mechanism are discussed. Performance results on architectural simulation with different memory coherence protocols are presented. Parallel discrete event simulation techniques are adopted for the execution-driven simulation of parallel architectures.

## 1 Introduction

Recent interest in high performance computing has led to a greater need for adequate simulation tools and models for evaluating parallel architectures and parallel programs on these architectures. Several simulation systems have been developed [1, 5, 6, 9, 10, 17] and based on the granularity of the simulator, the simulation techniques used in these simulators can be broadly classified into three categories: *trace-driven* or statistical simulation, *functional* or instruction level simulation, and *execution-driven* simulation with direct execution. Trace-driven simulation systems [1, 4, 10] have been used for a long time. They are fast but suffer from a low accuracy. Functional simulators such as the ASIM simulation system [6] provide a high degree of accuracy of a simulated execution with respect to the real-time execution. However, these simulators are usually slow. Execution-driven simulators, on the other hand, represent a compromise between the two extremes. These simulators sacrifice the speed of trace-driven simulations for greater accuracy. The representatives of this class are the Tango system from Stanford [9], the Proteus system from MIT [5], and the Wisconsin Wind Tunnel [17]. These simulation systems provide a reasonable level of accuracy between simulated and real-time executions with a moderate slowdown of about one to two orders of magnitude. In this paper, we describe an execution-driven simulation system, Maya[1], developed at the University of California at Santa Barbara.

Maya is a parallel programming system which supports execution-driven simulation on distributed memory architectures. The programming paradigm of Maya is that of shared memory. The current version of Maya is based on the PVM communication library [19], and can be ported to any environment that supports Unix and PVM. We have so far ported Maya to a network of Sun workstations and the Intel Paragon. Maya is capable of simulating the execution of a number of such memories including *causal memory* [3], and *pipelined random access memory* [16]. Preliminary evaluation results for several user applications appear in [2].

This paper is organized as follows. An overview of the design of Maya is presented in Section 2. In Section 3, the simulation environment and the modeling of parallel architectures are discussed. Section 4 contains some architectural simulation results and performance results. We conclude with a brief discussion in Section 5.

## 2 An Overview of Maya

Maya is designed to be a versatile tool for parallel programming with a variety of features. First, it is capable of simulating a target parallel architecture on a different host machine. Second, it is useful as a programming environment for rapid prototyping of parallel programs. Finally, it is intended as a test-bed for experimenting with a variety of shared memory protocols. To provide these features, Maya operates principally in two different modes: the *simulation* mode and the *native* mode. The simulation mode is used for simulating execution of a parallel program on the target architecture with a specific memory coherence protocol. In the native mode, the target and the host architectures are identical. Parallel programs are executed directly and no simulation is performed. In either mode, it is easy to replace one memory coherence protocol by another.

We assume that the host architecture consists of a set of processors each with its private memory (referred to as a node) connected by a network. The Maya environment on each node comprises of three layers: the user program or application layer, the memory subsystem layer, and the communication subsystem layer. The memory subsystem in Maya acts as an interface between the user processes and the network to support shared memory in a distributed environment. The accesses to the shared variables result in macro calls to the memory subsystem. Currently the interface between the users and the memory subsystem is based

*Work supported in part by NSF grant IRI-9117094 and LANL.
†Work supported in part by NSF grants CCR-9008628 and CCR-9223094.

[1] Maya in Sanskrit means "illusion".

on messages. The memory subsystem executes the underlying distributed shared memory protocol by communicating with the memory subsystems on other nodes. The distributed static manager scheme [14] is implemented as the base case memory protocol (referred to as "atomic memory" in this paper) for both comparison and testing purposes. The memory subsystem in Maya is extensible. Currently, we have developed a library to support distributed shared memory protocols based on *atomic memory*, *causal memory*, and *pipelined random access memory* [2].

The primary responsibility of the communication subsystem is to facilitate communication among the memory subsystems, which cooperate with one another to implement distributed shared memory. The communication subsystem in Maya is based on the message passing library PVM [19]. Since PVM is available on most distributed memory architectures, Maya can be supported on a variety of hardware platforms and on a network of heterogeneous Unix compatible machines.

# 3 Architectural Simulation

Maya can be configured as a simulation environment for parallel architectures. Functioning as a simulator, Maya automatically transforms both the parallel user program and the memory coherence protocol into simulation programs using a set of macros. Monitoring codes are inserted for each communication event in the coherence protocol and cycle counting codes are added to maintain timing information in the simulated environment. A number of useful statistical information can be obtained from Maya in the simulation mode.

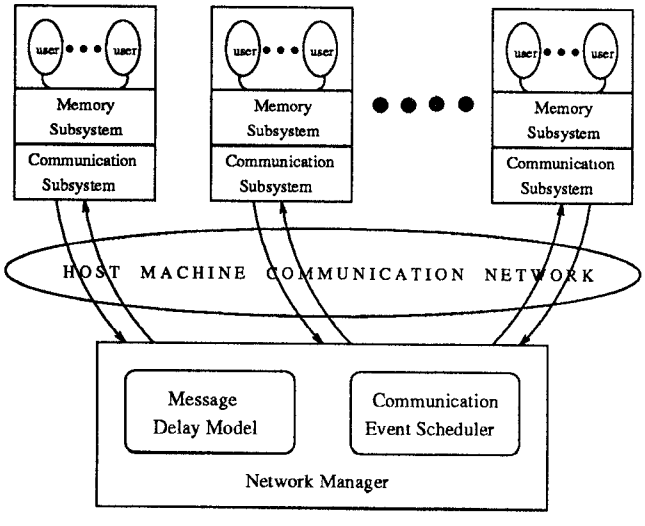


Figure 1: Architectural simulation in Maya

The configuration of Maya in the simulation mode is shown in Figure 1. The memory subsystem is responsible for scheduling user processes on the same node. The *network manager* is responsible for scheduling the simulation events due to messages sent over the network and is currently implemented as a centralized process. Two major components exist in the network manager: message delay model and communication event scheduler. The message delay model is used to determine the arrival time of a message sent over the target architecture communication network. The communication event scheduler schedules the delivery of messages and resolves any deadlock. Maya simulation ensures *repeatability*: repeated execution of the same user program with the same input will yield exactly the same output and simulated execution time. This feature is useful for rapid prototyping of parallel programs by making it easier for debugging timing anomalies. In the following subsections, we will discuss the modeling of target parallel architectures, the event scheduling mechanism, and the modeling of network contention in greater detail.

## 3.1 Modeling Parallel Architectures

Several models exist for parallel architectures and computations [8, 11, 13]. The traditional Parallel Random Access Machine model [11, 13] is not accurate enough since many important architectural aspects such as communication overhead are missing. The LogP model [8] characterizes a parallel architecture by the communication delay $L$, the communication overhead $o$, the communication bandwidth $g$, and the number of processors $P$. Various aspects of the architecture are approximated by these four parameters. Our architectural simulation model is similar to the LogP model. Parameters used in Maya can be categorized to model the delay, overhead and bandwidth. The number of processors $P$ is not explicitly used in our model.

In Maya, the characteristics of the parallel architecture are modeled by two sets of parameters. The first set of parameters captures the computational aspects of the parallel architecture, whereas the second set delineates the communication aspects. The parameters on computational speed are categorized based on instruction groups such as integer addition, floating point division, procedure call, etc. Cycle counting statements are inserted into the user program to maintain the timing information for local computation. The simulation clock is advanced as the simulation proceeds. To cater for the communication delay between nodes, we require the simulation model designer to provide a function *netdelay* that models the target communication network. Details of this function is discussed in Section 3.3. In the current prototype of Maya, the network manager schedules the message delivery events using *netdelay* since it has the complete knowledge of all communication events occurring between nodes.

## 3.2 Event Scheduling

In order to mimic the execution of the parallel program on the target architecture, events and processes need to be scheduled correctly. This execution is modeled in terms of a set of simulation events which correspond to the accesses to shared variables and the resulting message transmission, delivery, and processing in the memory subsystem. Maya attempts to schedule all these events in the same order as they would appear in a real execution on the target architecture by tagging each event with a timestamp from the simulation clock and executing them in timestamp order. Simulation events local to a user process are executed in timestamp order. Non-local simulation events such as transmissions and deliveries of messages are scheduled by the network manager. Logically, all internodal messages are sent to their destination node via the network manager, which maintains a queue of relevant events ordered by their timestamps. Based on the message transmission timestamps from senders, the network manager computes the timestamp of the next message delivery event by adding the network delay estimated by the message delay model and redirects the message to the receiver. This scheme results in a consistent interface for both the native and the simulation mode.

While the network manager is responsible for the scheduling of communication events across memory subsystems,

memory subsystems are responsible for the scheduling of events on the node. These events include shared access requests from user processes and the receipt of messages from other memory subsystems. Each memory subsystem maintains an ordered queue for these simulation events in the same manner as the network manager. For events due to messages from other memory subsystems, the arrival timestamps of the events are supplied by the network manager. For events due to shared access requests, a special function *time_mapping* is used to determine the arrival timestamps. Function *time_mapping* models the specific scheduling algorithm employed by the operating system on the target architecture. It takes the local timestamp of an event which is supplied by the user process and the relevant past history of the execution on the node and returns a modified local timestamp. This modified timestamp is an estimation of the actual scheduling time of the event after taking operating system scheduling overhead and delay into account. Different scheduling schemes require different *time_mapping* functions. For the applications we are simulating, there is only a single user process residing at each node. Consequently, function *time_mapping* simply models the overhead and delay due to a memory subsystem.

Maya's event scheduling follows the framework of parallel discrete event simulation [12], in which simulation events are scheduled in the same order as in actual executions. This can be guaranteed by observing the *input waiting rule* and the *output waiting rule* as in the conservative approach for parallel discrete event simulation [7]. In Maya, the input waiting rule is enforced by requiring the network manager to wait until it receives a message from each memory subsystem. The output waiting rule is enforced by requiring the network manager to schedule a message only when the simulation clock has advanced to the send time of the message. Conservative approaches in parallel discrete event simulation may lead to deadlocks. One way to avoid deadlocks is to flush all the communication channels periodically with *null messages* [7]. In Maya, the deadlock resolution scheme is attached to the centralized network manager. The deadlock resolution scheme is a combination of deadlock avoidance and deadlock detection. The network manager keeps track of the number of outstanding messages of each memory subsystem. When the number of outstanding messages drops to zero, the network manager calculates the largest time $T$ such that no memory subsystem will receive another message with send timestamp smaller than $T$. The value of $T$ is then broadcast to all memory subsystems. Memory subsystems are allowed to schedule local events only if they are issued before time $T$. If $T$ is updated and broadcast frequently, this scheme tends to avoid deadlocks before they actually occur. Furthermore, if deadlock does occur, the number of outstanding messages among the memory subsystems will eventually become zero. (This is obvious if the deadlock involves all the memory subsystems. On the other hand, if only a subset of memory subsystems are deadlocked, the local simulation clocks of the deadlock memory subsystems will not proceed any further. Eventually, the network manager will not be able to advance the value of $T$ any more and all memory subsystems will stop sending messages.) Subsequently, the network manager will be able to detect the deadlock and resolve it by sending another value of $T$. To compute the largest possible value of $T$, lookahead techniques [15] are adopted. In particular, suppose the smallest send timestamp of all the messages the network manager has received is $t$ and the minimum message delay on the target architecture is $d$. It can be shown that when the network manager finds that the number of outstanding messages is zero, no memory subsystem will receive any message before time $t+d$. Consequently, $T$ can be taken as $t+d$. Our scheduling scheme is further optimized in the case where successive shared accesses are separated by a large block of local access and computation. In this situation, the application process is setup to report the largest timestamp before which it will not make any shared access. Event scheduling on other nodes may be accelerated based on this additional information.

### 3.3 Modeling Contention and Communication Delay

It is important to take the congestion level of the target communication network into account in order to obtain a realistic simulation of a parallel program. To simplify the modeling process, we assume the existence of a function *netdelay* that takes into account the contention level in the network and computes the message delay. Function *netdelay* takes an event of message transmission and returns the time at which the message should be received by the receiver. For some network contention models, the arrival time of a message depends not only on the current network contention, but also on the traffic in the immediate future. The function *netdelay* is allowed to return $\perp$, if there is inadequate information to determine the arrival time of a message. Eventually, when sufficient information becomes available, an arrival time is returned. The amount of information needed to estimate the receipt time of a message varies for different accuracy levels. In our prototype, we assume that the scheduling of messages on most networks is independent of the local computation on the nodes of the target machine. Therefore, a separate network manager is used to schedule messages during simulation.

In the current prototype of Maya, the ideal delay of a message is modeled as the sum of the sender overhead, the receiver overhead, and the transmission delay. Sender and receiver overheads are assumed to be constant. Transmission delay is modeled as a linear function proportional to the length of the message. The actual delay of a message is the ideal delay plus a factor due to network contention. A collision-based model is introduced to model the effect of contention. This model is useful for communication networks that exhibit the behavior of message collisions. Let the *propagation interval* of a message transmitted at time $t$ be the interval $[t, t+d]$ where $d$ is the calculated transmission delay of the message. A collision is said to occur between two or more messages transmitted over a common channel if their propagation intervals overlap. We handle collision by rescheduling messages whose propagation intervals overlap. In our collision-based model, when a collision occurs, the message with the smallest transmission time succeeds and all the others are re-transmitted. Note that this results in recomputation of the propagation intervals for the delayed messages when they are re-transmitted. A penalty $e$ is charged to delayed messages so that they are re-transmitted $e$ units of time after the succeeded message.

## 4 Experimental Results

The current version of Maya includes a library of coherence protocols: *atomic memory, causal memory, pipelined random access memory*, and some of their variations. Several application programs including the Gaussian elimination and matrix inversion (Gaussian inverse), all pairs shortest path, the traveling salespersons problem, the Jocabi iterative synchronous linear equation solver, and the Cholesky factorization from the SPLASH benchmark [18] have been tested. Preliminary results of the performance of some of

these coherence protocols and application programs when used in the native mode have been obtained for a network of Sun workstations and an Intel Paragon [2]. In this section, we concentrate on the results from the simulation mode. These results are based on one user application: a Gaussian inverse algorithm, two kinds of coherence protocol: atomic memory and causal memory, and two different architectures: a network of Sun workstations and an Intel Paragon. The results obtained from the simulation mode are compared with the results obtained from the native mode. Finally, the performance of Maya is evaluated by varying the degree of concurrency.

### 4.1 Results of Architectural Simulation

The input to the Gaussian inverse problem is a matrix of size $N \times N$ and the output is the inverse of the input matrix. The $N$ rows of the matrix are evenly assigned to the $n$ participating user processes. In phase $i$, the user process in charge of row $i$ selects one of its rows as a pivot row. All the user processes cache this row and use it for the local computation on their assigned rows. Due to the use of barriers and the variable sharing patterns in the algorithm, coherence protocols weaker than atomic memory can be used. We use atomic memory and causal memory in our experiments. Copies of the memory pages may migrate dynamically and are kept track of by a distributed directory [14]. In the case of causal memory, each memory subsystem keeps a copy of the shared memory. Reads to the shared memory are served locally. Writes to the shared memory are broadcast and performed on each node in a causal order.

A network of Sun workstations is used as the host machine in our simulation experiments. The target machines being simulated are the same network of Sun workstations and an Intel Paragon. In the network of Sun workstations, an Ethernet is used to connect a set of Sparc LX workstations. The number of user processes being simulated is denoted by $n$. Each simulated user process and the corresponding simulated memory subsystem are assigned to one workstation so that there are a total of $n$ workstations. On the Intel Paragon, a partition of $2n$ nodes is used and the simulated user processes and simulated memory subsystems are assigned to different nodes of the machine. To obtain the necessary parameters for simulating the two target machines, we have written simple benchmark programs to estimate the speeds of the processors for local computation, the communication overheads when using PVM as the underlying communication interface, and the bandwidths of the networks.

Figure 2 compares the actual execution times and simulated times running Gaussian inverse with $N = 500$ and $n$ equal to $2, 4, 6$, and 8. As shown in figures 2*a* and 2*b*, the simulated times obtained for causal memory match the actual execution time within a small percentage on both the Paragon and the network of workstations. In the case of atomic memory, the simulated times obtained also agree with the actual execution time on the Paragon (Figure 2*c*). However, the simulated times obtained in the case of the network of workstations are relatively low compared with the actual execution time (Figure 2*d*).

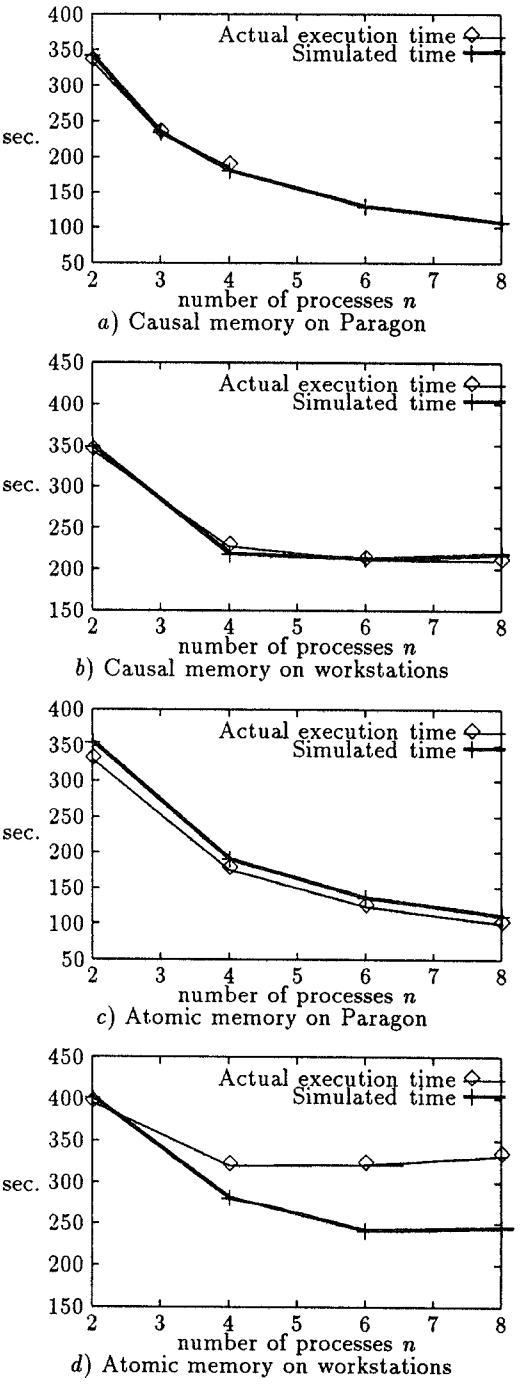


Figure 2: Native mode versus simulation mode for Gaussian inverse with $N = 500$

### 4.2 Performance of Maya

Figure 3 shows the speedup of performing simulation in Maya by varying the number of processors used in the host environment. In this experiment, we fix the number of user processes $n$ to 32. We assume that each user process resides on a different processing unit of the target machine and there is a memory subsystem corresponding to each user. We simulate the same Gaussian inverse algorithm using one or more workstations. Since the bottleneck of the simulator is in the network manager, only moderate speedup is obtained as more workstations are used.

The slowdown of performing simulation in Maya with 2 workstations and a varying number of user processes is depicted in Figure 4. As the number of user processes is in-

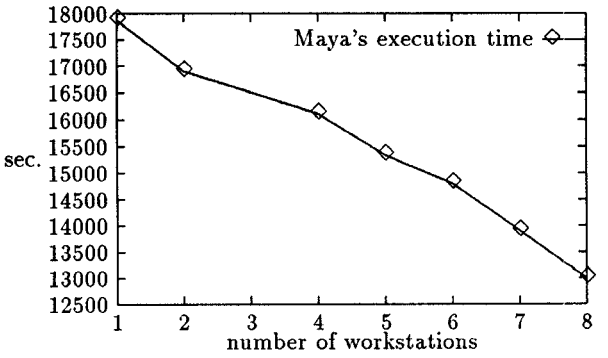


Figure 3: Simulating using various number of machines ($N = 500, n = 32$)

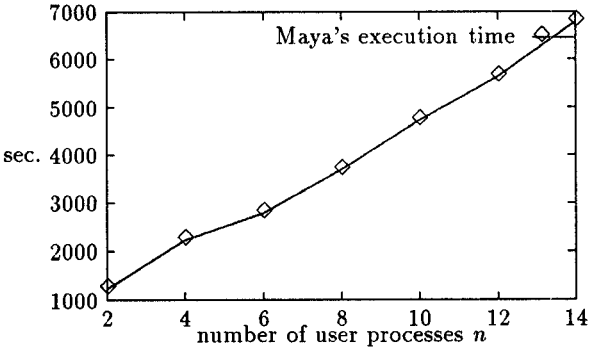


Figure 4: Simulating varying number of user processes using 2 workstations ($N = 500$)

creased, the number of memory subsystems and the number of processors in the target machine are also increased correspondingly. The slowdown is almost linear to $n$, which is expected as more computation and communication are needed to simulate larger number of user processes.

## 5 Discussion and Future Work

In this paper we have presented the system overview of Maya which is a simulation platform for evaluating distributed shared memory protocols and parallel architectures. Maya is an execution-driven simulator and can be ported to a variety of distributed memory architectures. It can be used for rapid prototyping of parallel programs employing the shared memory paradigm. Furthermore, Maya facilitates shared memory parallel program development based on different notions of memory consistency. We feel that this is a better approach of developing parallel programs instead of exposing the low-level message passing paradigm to the users. Finally, users can experiment with a variety of parallel architectures which can be simulated through Maya.

The memory subsystem is being integrated with the user process in the next generation of Maya. This approach will result in minimal access time for locally available shared variables. Another issue that we are investigating is to decentralize the current implementation of the network manager, which is the major bottleneck of the system.

## References


[1] A. Agarwal, J. Hennessy, and M. Horowitz. Cache performance of operating system and multiprocessing workloads. *ACM Transactions on Computer Systems*, 6(4):393–431, November 1988.

[2] D. Agrawal, M. Choy, H.V. Leong, and A.K. Singh. Evaluating weak memories with Maya. Technical Report TRCS-93-23, University of California at Santa Barbara, Department of Computer Science, 1993.

[3] Mustaque Ahamad, James E. Burns, Phillip W. Hutto, and Gil Neiger. Causal memory. In *Proceedings of the 5th International Workshop on Distributed Algorithms*, pages 9–30. LNCS, October 1991.

[4] A. Borg, R.E. Kessler, and D.W. Wall. Generation and analysis of very long address traces. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 270–279, 1990.

[5] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl. Proteus: A high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, MIT, Laboratory for Computer Science, September 1991.

[6] D. Chaiken, B.H. Lim, and D. Nussbaum. *ASIM User Manual.* ALEWIFE Systems Memo#13, August 1990.

[7] K.M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, 5(9):440–452, September 1979.

[8] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 1993.

[9] H. Davis, S.R. Goldschmidt, and J. Hennessy. Multiprocessor simulation and tracing using Tango. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume II, pages 99–107, August 1991.

[10] S.J. Eggers, D.R. Keppel, E.J. Koldinger, and H.M. Levy. Technique for efficient inline tracing on a shared-memory multiprocessor. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 37–47, May 1990.

[11] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10th Annual ACM Symposium on the Theory of Computing*, pages 114–118, 1978.

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

[13] P.B. Gibbons. A more practical PRAM model. In *Proceedings of the 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 158–168, 1989.

[14] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[15] Y.B. Lin and E.D. Lazowska. Exploiting lookahead in parallel simulation. *IEEE Transactions on Parallel and Distributed Systems*, 1(4):457–469, October 1990.

[16] Richard J. Lipton and Jonathan S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, Department of Computer Science, September 1988.

[17] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 48–60, May 1993.

[18] J.P. Singh, W.D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared memory. Technical report, Stanford, Computer Systems Laboratory, 1991.

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