Execution-Driven Distributed Simulation of Parallel Architectures

Livio Ricciulli, Patrick Lincoln, and José Meseguer

Computer Science Laboratory

SRI International

Menlo Park, California 94025, USA

Phone: 415-859-2969

Fax: 415-859-2844

E mail: {livio, lincoln, meseguer}@csl.sri.com

Keywords: Performance evaluation, Distributed simulation of computer architectures, Execution-driven simulation, Distributed shared memory computers, Relaxed memory consistency.

Abstract: A new methodology for the asynchronous discrete event-driven simulation of parallel computers is proposed. This methodology integrates sequential and distributed simulation in a unified paradigm and is applicable to the simulation of all classes of parallel computer architectures. In our own simulation work we accelerated simulations by more than an order of magnitude with parallel execution speedup efficiencies in the order of 60-70%. When simulating in parallel, our approach has the important benefit of testing the robustness of a simulated design by not hiding the asynchronous nature of the system being studied (our simulation model preserves the non-deterministic behavior of certain parallel executions). Unlike other distributed simulation methodologies, our execution model does not rely on a global view of virtual time to maintain coherent distributed event causality relations. The simulator correctly executes a given parallel application that observes a particular synchronization model of choice without a notion of virtual time. We then estimate a global virtual time in which the execution could have been carried out. We give a detailed description of our simulation methodology, the computational models that it can implement, and the conditions for its correctness. We also give some preliminary performance results obtained by implementing our parallelization technique to simulate a massively parallel machine on a CM-5 computer and on a heterogeneous network of workstations.

1 Introduction

As parallel computer systems being designed and simulated grow in both complexity and size, it becomes increasingly harder to perform simulation experiments with a sufficient level of detail within reasonable time limits. Indeed, given the performance currently offered by sequential workstations, parallel execution often becomes the only viable alternative for realizing detailed simulations of sufficiently large systems. However, effectively using a parallel computing environment to drastically increase the performance of an architectural simulation is a hard problem, for which new techniques are needed.

We propose a novel methodology for the distributed event-driven simulation of parallel architectures that can effectively exploit the existing computational power offered by today's parallel and distributed computing environments. Our approach is based on the execution-driven simulation methodology and affords unprecedented efficiency by enforcing only application-specific synchronizations to synchronize the distributed event-driven simulation. Our simulation methodology is general enough to efficiently simulate any parallel MIMD or MIMD/SIMD architecture in a distributed manner. In the case of MIMD multicomputers our methodology applies to both distributed memory machines with asynchronous message passing [2] and Distributed Shared Memory (DSM) multicomputers. For the sake of concreteness and because of their intrinsic interest, this paper focuses on the simulation of DSM machines.

Because parallel computers have explicit synchronization requirements, their simulation can be decoupled in two distinct parts:

1. Correctly executing a simulated parallel application on a simulated architecture
2. Generating a simulated time estimate in which the above execution could have been carried out

This decomposition gives more flexibility in choosing an appropriate level of simulation detail to suit one's particular needs. In one limiting case, by ignoring virtual time, one could perform simulations testingonly the correctness of an architecture and of the application being simulated without incurring the overhead of having to keep track of virtual time. In another limiting case, by using a trace-driven approach, one could completely abstract the execution of the parallel application and perform, for example, detailed timing measurements of the latency and throughput of the inter-processor interconnection network.

In the experiments we have performed, we chose to have the execution simulation at a fairly high level of detail and to keep the time estimation detail at a level that, while still providing useful information, allowed very efficient parallel simulation. By slightly reducing the accuracy of our timing estimator (±4-5% error) we were able to obtain a highly parallel simulation platform that, in addition to yielding good performance when executed on parallel machines, is ideal for implementation on heterogeneous networks of workstations where global lock-step synchronization is prohibitively expensive. In our own architecture work, this technique has greatly increased our experimentation turnaround time by more than an order of magnitude. For the particular parallel applications with which we have experimented, we have found that our technique achieves an average efficiency of 61% of ideal parallel speedup on a 64-node CM-5, and 71% of ideal speedup on a 16-node heterogeneous network of workstations.

We have thus far experimented using our approach to simulate one particular parallel architecture at the Register Transfer Level (RTL). We believe, though, that our approach is directly applicable to other parallel machines not only for RTL simulations, but also for higher-level symbolic simulations [10] and for lower-level simulations of parallel computer designs directly specified in VHDL [5]. This latter kind would have the enormous benefit of avoiding the possibility of translation errors by integrating architectural studies directly into the digital design process. Potentially, very costly errors, such as Intel's Pentium/90 problems in a multiprocessing environment [14], given enough computational power, could have been discovered by using this kind of methodology.

Our methodology is also quite general with regard to the physical platforms on which it can efficiently run and and allows seamless integration of sequential and parallel simulation. The same description of our simulated machine has been successfully and efficiently run on a single sequential workstation, a 4-processor SPARC-server 670 MP, a CM-5, and a heterogeneous network of workstations. The network of workstations we used in our experiments included multiprocessor SPARC-servers, single-processor SPARC-10s, and a Pentium/90 box running Linux, distributed over two separate Ethernet domains and interfaced using standard Ethernet hardware. We believe that this approach is therefore well suited to effectively utilizing department-level clusters of workstations to obtain more than an order-of-magnitude performance gain.

2 Our approach

Our proposed distributed event-driven simulation methodology has been designed with the explicit purpose of meeting the needs of parallel architecture simulation. We can capture the essence of what is different about our methodology by illustrating schematically the casual relation between the execution of simulation events and the simulated time at which they appear to have executed.

Conventional : Time $\Rightarrow$ Execution

Our method: Time $\Leftarrow$ Execution

In the conventional case, time directly drives the execution and event causality is strictly defined; events are executed or committed only if the global simulation time permits. Our approach starts from an execution, which is guaranteed to be legal (produces correct results), and generates a timing estimate in which the execution could have been carried out. The simulated application directs the simulated execution in such a way as to always produce correct results (this also tests the soundness of the design), and the timing estimator produces a timing estimate that conforms to this execution.

By using this approach the parallel simulated execution is easy to implement efficiently because, once all necessary synchronization mechanisms of a given synchronization model are in place, the parallel application being simulated will itself synchronize the simulation with minimal overhead. In previous distributed simulation techniques virtual time is used to guarantee the correctness of the execution by giving a total order of simulated events regardless of the fact that, as we will see in Section 3, this total ordering is not always necessary.

The accuracy of the timing estimation can be adapted to one's particular needs, therefore leaving more freedom to the simulator designer. As we will see in Section 4, given a particular execution, it is possible to very accurately predict at runtime the timing performance of a distributed system with very simple mechanisms. These mechanisms can be installed in an arbitrary number of places, depending on the degree of accuracy that one wants to obtain. In general these mechanisms can be activated whenever a logical process acquires a resource produced by another logical process.

2.1 Parallel Execution Simulation

In this section we outline which programming models can be simulated using our approach and show how to implement a simulator that correctly executes parallel programs written under the Release Consistency synchronization model.

In the context of shared memory parallel programming a variety of relaxed memory consistency models [1,6,8] have been proposed. These models define conditions under which shared memory parallel programs appear to the programmer as having sequentially consistent executions as defined by Lamport [9], but allow the hardware to schedule coherence messages more freely. Instead of emulating a serialization of global memory access, the hardware guarantees a fixed set of coherence properties that are understood by the software as a synchronization model. If the software then observes the constraints of the given synchronization model, then the architecture produces executions that are sequentially consistent [1]. Such relaxed consistency models allow much more parallelism than previous ones because global memory access operations do not always need to be ordered but in some cases are allowed to propagate asynchronously without introducing overhead to enforce sequential causality.

When a simulation's observable behavior of interest is the relative ordering of all the simulated events of the distributed entities, the simulation must enforce such ordering with simulation techniques of the style proposed by Chandy and Misra [4] or Jefferson [8]. Instead, if the relevant observable behavior of a system is not the ordering of events itself but its final state and its overall performance, we can greatly simplify the simulation methodology. In our methodology it is sufficient to maintain causality of events only to preserve the correctness of the simulated application and not to emulate a given deterministic execution.

Our methodology correctly simulates the execution of a parallel program that obeys a particular synchronization model, provided that the simulator correctly implements that model in a distributed manner. For example, a program written under the Release Consistency model [7] executes correctly on a simulator of the hardware mechanisms required by this memory consistency paradigm and therefore yields a sequentially consistent simulation.

To accomplish this, each node of the simulated machine is mapped to an autonomous Logical Process (LP) that keeps its own local time and executes the simulated parallel application within the context of execution-driven simulation. We allow the simulator to consist of as many LPs as needed, and we map the LPs on the nodes of the physical parallel or distributed system in an optimal way. If multiple LPs are mapped on the same physical processor, they can be spawned as separate UNIX processes, or they can be grouped as a single UNIX process sharing the same discrete-event list. Allowing groups of LPs to be mapped in a single UNIX process somewhat increases the mapping complexity but minimizes the UNIX overhead in transferring data among LPs and achieves smooth integration of parallel and sequential simulation; *in fact when all LPs are mapped to a single UNIX process sharing the same event list, the simulation becomes sequential***.** By using this property we were able to seamlessly compare our parallelization technique with a corresponding sequential deterministic simulator.

A very important consequence of our LP partitioning is that there is a one-to-one correspondence between simulated shared data movement and messages that propagate between LPs. Under this kind of mapping, the implementation and simulation of relaxed memory consistency models will generate three kinds of events: local non-coherence simulation events, (simulated) global data access events, and (simulated) synchronization events:

1. Local noncoherence simulation events do not directly influence global consistency and are needed for the simulation to make forward progress.
2. Simulated global data access events simulate application data movement.
3. Simulated synchronization events control the execution of the LPs and can cause an LP to be either blocked or suspended, or to resume from a blocked or suspended state.

When data access events are also synchronizing events according to our definitions, we treat them as synchronization events. For example, a cache miss causes a block fetch that carries data and also causes the missing LP to resume; in this case we would consider a block fetch to be a synchronization event.

The simulation of a particular synchronization model classifies all events generated by the execution of a parallel application to be of one of the three above types. The simulator then enforces consistency by observing the following rules:

1. Local events of type 1 and coherence events of type 2 are always executed once they are generated by the simulated program.
2. Coherence events of type 3 can cause an LP to execute a series of 'null' events that could effectively block the execution of the application for that LP. In addition, these events trigger specific mechanisms of a particular synchronization model of choice to maintain sequential consistency.

Given the above rules, a simulation based on our methodology yields correct results if the following two conditions hold:

1. The application being simulated satisfies the restrictions imposed by the given synchronization model M.
2. The simulator correctly implements the synchronization model M.

Application requirements to satisfy condition 1 totally depend on the architecture and memory consistency model that are being simulated and therefore are specific to the particular instantiation of our technique. Condition 2 essentially requires the simulator to be correct.

The above requirements do not impose any particular computational model or architecture for the applicability of our methodology. Because of its simplicity and potential for exploiting application parallelism, we have used Release Consistency [7] as the synchronization model of the particular simulation experiments we have performed. For this model we can translate the correctness conditions given in [7] to further specify the correctness conditions 1 and 2 mentioned above:

1. The application being simulated is properly labeled (it is race free).

2.1 All synchronization acquire events performed by a processor must be performed before any subsequent global data access event can be issued.

2.2 A synchronization release event can be observed by a processor after all preceding global data access events are performed.

2.3 Synchronization events need to be processor-consistent.

Condition 1 basically requires that any two conflicting events (two operations with the same address, of which at least one is a write [12]) must be ordered by a synchronization operation between them. Condition 2.1 specifies that an LP must block (generate null events) until it acquires needed synchronization resources. Condition 2.2 specifies that all memory operations must be committed before a release can cause an LP to resume execution. Condition 2.3 requires that LPs perform synchronization operations in program order.

These conditions ensure that the logical view of the distributed simulation is sequentially consistent under the release consistency model. Furthermore, reasonable assumptions about the actual simulation platform along with these properties imply that the result of the actual simulation running on the real hardware does not depend on vagaries of timing or race conditions in the simulation platform. Thus, assuming that a parallel application is race-free is of paramount importance, since it lifts from the simulator the responsibility of enforcing a total order of simulation events.

2.2 Clock Synchronization

In most cases, ensuring the correctness of a simulated application's execution is not enough to make a simulation methodology usable. The simulation model described in Section 2 does not, as yet, provide a correct estimate of the execution time. The different LPs execute asynchronously and, although producing the correct data, do not provide a realistic estimate of global virtual time. In our methodology the clock synchronization solution is simple, because the simulated application will block the execution of an LP and therefore enforce an ordering of events that will never need to be undone. Unlike Time Warp [8], we only modify the virtual clocks without having to restore previous states.

For the purpose of clock synchronization we view a parallel execution as a series of resource requests, acquisitions, and releases. This unifying view allows us to break down all synchronizing events of a parallel execution as being of one of these three types. This view is general enough to model all computational paradigms of interest, and it abstracts the behavior of a parallel program in a way that allows us to easily generalize our clock synchronization mechanism. A resource can be any object (hardware or software) that a process must acquire to make forward progress in the computation; for example a buffer segment is a resource needed by an LP before it can send a message, and a mutual exclusion lock is an object needed before a critical section is entered.

Given a particular resource assignment ordering, resource acquisitions are the only synchronization points at which clocks must be adjusted to yield a correct estimate of global time. In the usual case, the virtual time of a process that acquires a resource is either the time at which the resource is made available or the time the resource is requested if the resource was already available. Complications arise when the clocks of the process that last released the resource and the clock of the process that requests the resource are not synchronized.

We use lock contention as an example to first motivate this clock synchronization methodology and then to explain and validate the mechanism we use. Fig. 1 depicts what can go wrong if the clocks are not synchronized; in all the examples, LP1 acquires a lock before LP2 with respect to physical time. After completing the mutual exclusion sections the processors terminate, and the total concurrent execution time is taken to be the maximum of the two LPs' execution time. Depending on their relative speed, the two LPs might terminate with an erroneous estimate of the simulated time that does not reflect a proper event causality relation.

In Fig. 1a the two processors behave synchronously and do not need clock adjustment. Fig. 1b shows what would happen if the host node of P1 is faster and allows P1 to complete the mutual exclusion section before P2 issues an acquire message. The resulting total virtual time of 201 is wrong because it is derived by executing an impossible schedule (the time estimate implies that p1 and p2 execute the mutual exclusion at the same time).

In case 1b we need a mechanism to synchronize the local clock of LP 2. We assume that

1. LPs cannot change their virtual times except at the acquisition of a resource.
2. LPs cannot change other LPs' virtual times.

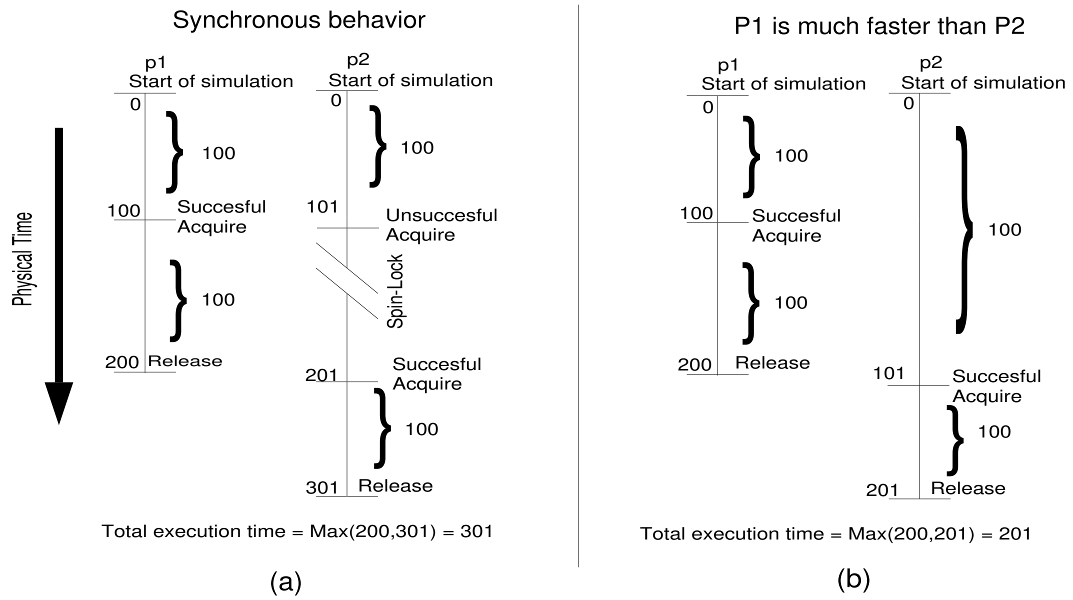

Fig. 1. Some possible behaviors of two virtual processors contending for a lock variable. (a) The processors execute synchronously. (b) P1 is much faster than P2; P1 and P2 may appear to execute the mutual exclusion section at the same time.

3. A blocked LP cannot do anything other than acquiring the resource for which it is blocked.

The following accomplishes clock synchronization of two or more processes contending for a resource.

The virtual time of an LP after it acquires a resource R is the maximum of

1. Its local time at the first attempt to acquire the resource R

2. The local time of the LP that last released the resource R plus the propagation delay of the release-acquire messages

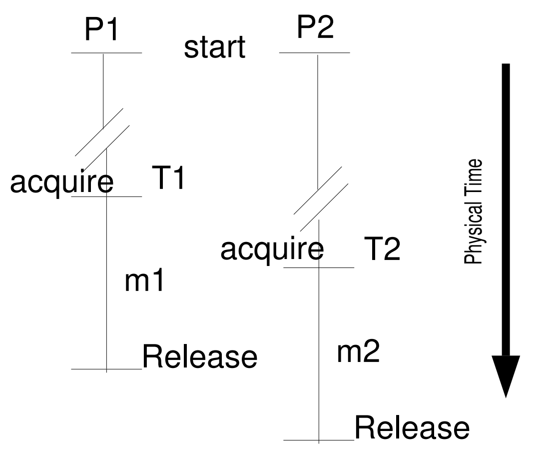

Fig. 2. Two processors contending for a mutual exclusion section.

A simple argument supports our claim. Suppose that two processes P1 and P2 are contending for a common lock at virtual local times *T1* and *T2,* respectively, and that they execute a series of events of duration *m1* and *m2,* respectively, after they acquire the lock, and that *T1* happens before *T2* in physical time. Assume also that *delta* is the overhead time for the propagation of the release-acquire messages. P2's local clock must respect a causality relationship with the real event order observed (P1 acquires the lock before P2). If $T1+m1<T2$, then the causality relation is maintained without

adjusting $T2$. If $T1+m1 \geq T2$, then the causality is maintained if $T2$ is set to $T1+m1+delta$. The total virtual execution time for both processors to complete their mutual exclusion sections should be

$Texec = T1+m1+m2+delta \quad \text{if } T2 \leq T1+m1+delta.$

$Texec = T2+m2 \quad \text{if } T2 > T1+m1+delta.$

or more compactly

$Texec = Max(T2,T1+m1+delta)+m2$

Since LPs can change their clocks only at the acquisition of a resource, it follows that for the total execution to yield $Max(T2,T1+m1+delta)+m2$, the second LP, P2, must adjust its clock at the beginning of $m2$ to $Max(T2,T1+m1+delta)$. This always works because an LP issuing a releasing event cannot be itself blocked and therefore must have a correct virtual time.

This idea is quite general and can be used to synchronize virtual times on any event that causes a blocked LP to resume useful execution, including, for example, clear operations on lock variables. This method could synchronize, in a distributed manner, the local virtual clocks of any system with synchronization primitives based on lock and clear operations (e.g., using the ANL macros [3]). In a distributed implementation each resource has associated with it a time Tr that indicates the local virtual time of the last LP to release that resource. An LP successfully acquiring a resource uses Tr to compute a new local virtual time.

3 Our Implementation

One of the main advantages of our simulation methodology is that it allows a great degree of freedom in selecting the level of detail one wants to achieve. Here we explain the particular choices we have made in our own architecture simulation work and explain how these techniques also allow to effiently simulate a complex interconnection network in a very parallel way.

3.1 The simulated architecture

We have applied our ideas to the simulation of the Rewrite Rule Machine (RRM) [15]; a novel MIMD/SIMD parallel architecture currently being designed and simulated at SRI International. The RRM system we simulated consists of 64 SIMD processors, each with its own separate controller executing in MIMD mode. The simulator holds a very detailed description of all the hardware down to the register level; it uses the libraries provided by the general-purpose simulation package Csim [11]. This package is an extension of the C language; it allows very efficient process-oriented event-driven simulations. Each device of each node is a separate process that interfaces with other processes through synchronization lines (events) and hardware queues (mailboxes). Contention is carefully taken into account at all levels, and timing (the amount of time each process takes to perform a given operation) is derived from a careful analysis of the hardware as it would be implemented with realistic high-end microelectronics technology. This architecture is difficult to simulate sequentially because of its massive parallelism (the system simulated consisted of 36,864 processing elements). Individual sequential runs of our optimized sequential simulator typically take weeks of SPARC-20 CPU time.

3.2 Timing Estimation

As mentioned in the introduction, we try to subordinate timing accuracy to

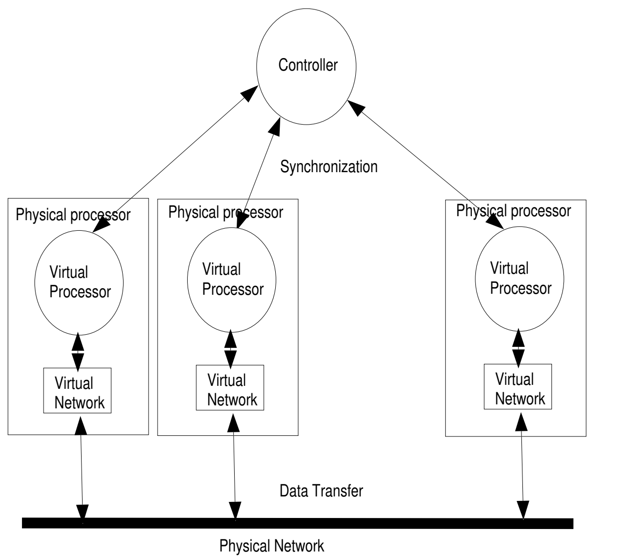

Fig. 3 A central controller coordinates clock synchronization while data transfer occurs in a distributed manner.

performance. To achieve the maximum degree of parallelism we choose to synchronize the local clocks of the various LPs only when they execute synchronization operations. In addition, to simplify our implementation, we introduce a central controller to arbitrate the clock synchronization operations. The controller exchanges messages with the LPs to determine a correct causality relation among synchronization events and supplies LPs with a correct estimate of the execution time. Although our implementation utilizes a central controller, our general methodology, as explained above, requires neither a centralized control nor any notion of current global time. For truly large distributed simulations, very fine grain applications or very detailed simulations with timing estimation mechanisms installed for all hardware resources, a central controller can become a bottleneck, but for the applications we have simulated and the level of our timing accuracy we have not found the controller to be a critical bottleneck in utilizing as many as 256 physical nodes.

Fig. 3 schematically illustrates our simulation methodology. The central controller implements our synchronization scheme and sends messages to the synchronizing LPs to adjust their clocks. The controller is invoked only when the virtual processors execute

LP

```
Barrier(p)
{
    INC(barvar)
    send(HOST,local_time)
    while(barvar < N) { nop }
    receive(time)
    local_time = time
}
```

Host

```
Barrier_control
{
    var = 0
    while( var < N)  {
        receive(time[p])
        var = var +1
    }
    release_time = MAX(time[1..N]) + delta
    for(p=0;p<N;p++)  send(p,release_time)
}
```

Fig. 4. Barrier clock synchronization mechanism. **N** is the number of LPs; **barvar** is the global counter used to implement the barrier; **INC** is an atomic increment instruction; **p** is the LP number issuing the barrier call; **var**, **time[1..N]**, and **release_time** are local variables of the controller; and **delta** is a fixed delay due to the release-acquire messages.

synchronization operations. All data movement between virtual processors happens directly between the physical nodes, without the controller intervention.

As an LP enters a blocking synchronization primitive it notifies the controller of its intent. At the release of a resource, a virtual processor notifies the controller of the time the resource was released. After an LP acquires a resource, it receives from the controller the appropriate local time (the maximum of the releaser and requester times). Notice that we still simulate the data movement required by the simulated synchronization operations as it would happen in the real system.

Fig. 4 shows how clock synchronization is maintained when a barrier is executed; all virtual processors notify the controller of their local times as they enter the barrier. Once the controller detects a release (barrier is complete), it adjusts the virtual times of the virtual processors to the maximum of the times it has received.

LP	Host

Enter_mutex(p)
{
  send(HOST,(p,local_time,LOCK,i))
  while(T&S(lock[i]==0) { nop }
  send(HOST,(p,local_time,ACQUIRE,i))
  receive(time)
  local_time = time
}
Exit_mutex(p,i)
{
  send(HOST,(p,local_time,RELEASE,i))
  CLR(lock[i])
}
	Lock_control()
	loop {
	  receive((p,t,operation,i))
	  if operation == LOCK time[p]:=t
	  if operation == ACQUIRE
	    send(MAX(time[p],release_time[i])+delta)
	  if operation == RELEASE release_time[i]:=t
	
	}

Fig. 5. Mutual exclusion clock synchronization mechanism. **i** is the index of a global array of lock variables used to implement mutual exclusion; **p** is the LP number from 1 to N issuing the operations; **T&S** is a test-and-set operation; **p**, **time[1..N]**, **t** and **operation** are local variables of the controller; and **delta** is a fixed delay due to the release-acquire messages

Fig. 5 shows how clock synchronization can be maintained for mutual exclusion synchronization. The technique works for any number of processors and any number of mutual exclusion variables. As in the barrier case, virtual processors notify the controller when they enter a synchronization primitive and ask for the correct local time when exiting.

These synchronization techniques assume a fixed delay delta for the release-acquire overhead. More sophisticated algorithms that actually record the propagation delays encountered by the release-acquire messages could be used if such an approximation were inappropriate.

3.3 Network Modeling

Our network modeling methodology, although slightly inaccurate, yields a very fast and distributed way of simulating large networks. Because we do not synchronize the local virtual clocks of the LPs on network resources acquisitions, network contention may be modeled differently than in a global time view. Given the asynchronous nature of the simulation, messages with the same virtual time may traverse the virtual network at different real physical times or messages with different virtual timestamps may contend for a common resource at the same physical time. To model network contention more accurately all network resources like simulated buffers or communication lines could trigger resource-acquisition clock synchronization mechanisms like the ones used for the other synchronization constructs.

We use only timing information that measures relative propagation delays through the different parts of the simulated system, and we always ignore the local time of the virtual devices through which the messages propagate. Fig. 6 depicts a read operation invoked by processor A routed through processor B to fetch a value from processor C. Each message is timestamped with the local time of the original sending process. As messages propagate through the virtual network, the timestamp is updated by adding the latencies of the network being simulated. For example, processor B will add T3 and later T9 to the timestamp of the message as it travels through its simulated hardware. Likewise, processor C will record the local time at which the message arrives and will add to the timestamp of the message the propagation delays T4, T5, T6, T7, and T8.

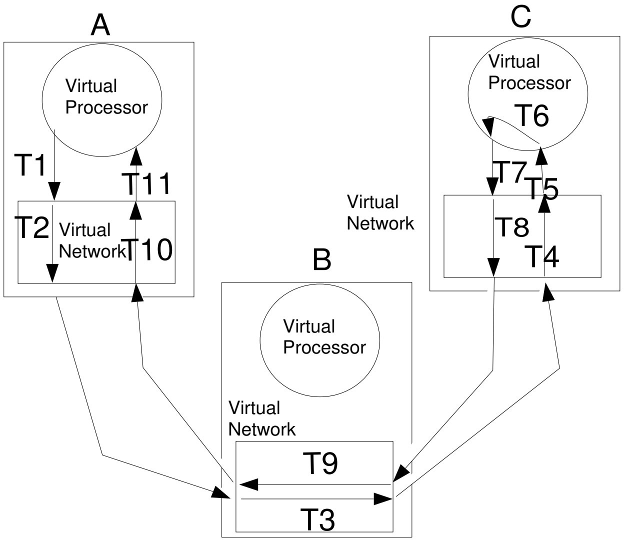

Fig. 4. A and D read a value from C, and the network simulation routes the request through B.

When a message reply is needed (as in the example above), the timestamp of the reply will reflect the time the message should have returned to the original sender in its own time reference and, therefore, the local clock of the original sender can be directly adjusted to the time of the timestamp. Note that if the sender is not blocked, as would happen with nonblocking memory reads, the timestamp of the reply can be discarded because we can assume that the time of the delivery of the message does not influence the LP total execution time.

4 Experimental Evidence

In reporting on the results of some experiments aimed at evaluating our technique, we compare sequential simulation performance with executions on both a CM-5 multicomputer and a heterogeneous network of workstations. This initial experimental evidence supports the effectiveness of our approach.

4.1 The Simulated Applications

As in the Release Consistency programming model [7], we do not enforce a total ordering of the synchronization operations because this would place a very strong burden on the simulator, thus reducing its efficiency. The nondeterminism introduced by this choice can cause variance in the simulation's results, but in our view if such determinism is necessary, it should be enforced by the simulated application rather than by the simulator. This is exactly the approach that would be followed when the application is run on a real machine and is in fact a reflection of the indeterminacy existing in some real executions.

We have evaluated our methodology on the simulation of three applications: sorting, hardware gate-level simulation, and the (D)ARPA image understanding benchmark for parallel computers [13]. All these applications follow the Single Program Multiple Data (SPMD) paradigm and were coded in assembly language.

4.2 Comparison of Sequential vs. Parallel

Tables I and II report the performance of our parallel simulator on a CM-5 and on a heterogeneous network of workstations. In the CM-5 experiments we mapped one LP per physical processor; in the network-of-workstations experiment we mapped either one or four LPs per workstation. The sequential execution times were taken by executing the same simulation on the same simulator with 16 or 64 LPs mapped on a single UNIX process run on a single processor. We spent considerable time in optimizing the sequential version of the simulator, but did not pursue several CM-5-specific optimization possibilities. Given that Ethernet sockets are already at a fairly low level, we anticipate that the most fruitful optimization would be network improvements; we plan to experiment in the near future with an ATM switch. The porting of the simulator to the different platforms was straightforward, thus giving us further evidence for the generality and the machine independence of our approach.

TABLE I

Performance on a 64-node CM-5

	SPARC-10/41	CM-5/64
Sorting/64		
Physical Execution Time (sec.)	265,437	11,020
Speedup		24.0
Virtual Execution Time (cycles)	3,646,017	3,356,069
Image Understanding/64		
Physical Execution Time (sec.)	101,910	5,688
Speedup	1	17.9
Virtual Execution Time (cycles)	2,012,109	2,016,990
HW Simulator/64		
Physical Execution Time (sec.)	22,678	1,320
Speedup	1	17.1
Virtual Execution Time (cycles)	127,982	126,728
Average Speedup		19.6
Average virtual time % variation		2.6

Table I reports the results obtained by running our parallel simulator on a CM-5. Notice that the speedup is calculated over the execution time of a SPARC-10. If one were to take into account the fact that, for our simulator, the SPARC-10 is about twice as fast as each of the CM-5 nodes, our speedup figures for each experiment would double, yielding on the average 69% of ideal speedup.

Table II summarizes the results obtained by running our simulator on a heterogeneous network of workstations. The experiments were performed with one SPARC-20, two 2-processor SPARC-20s, two SPARC-10/61s, five SPARC-10/51s, four SPARC-10/41s, and one Pentium/90. The workstations were distributed on two separate 10-Mbit Ethernet

TABLE II
Performance on a Network of Workstations

Number of Workstations	1	4	16
Sorting/16			
Physical Execution Time (sec.)	20,892	5,748	2217
Speedup	1	3.6	9.4
Virtual Execution Time (cycles)	1,383,016	1,378,09	1,348,737
Image Understanding/16			
Physical Execution Time (sec.)	14,825	4,202	1,560
Speedup	1	3.5	9.5
Virtual Execution Time (cycles)	1,322,434	1,263,301	1,319,723
HW Simulator/16			
Physical Execution Time (sec.)	9,214	2,069	630
Speedup	1	4.45	14.6
Virtual Execution Time (cycles)	262,796	256,734	256,857
Average Speedup		3.85	11.4
Average virtual time % variation		1.79	1.63

domains and interfaced with standard Ethernet hardware. The speedup was calculated over the execution time of a SPARC-10/41, since the total parallel execution time for our applications is primarily determined by the execution time of the slowest workstation. The average speedup of 16 workstations yields 71% efficiency over ideal speedup.

The total size of the sequential simulator was 22 to 35 Mbytes, and we suspect that it had good locality. As the system being simulated grows in size we expect the speedup to greatly increase, because larger data sets will not fit well within the single processor cache or even real memory and therefore will give advantage to parallel simulators with much larger total cache size, total real memory, and total CPU-memory bandwidth.

Because the parallel simulation is not totally ordered, some small variations result in the virtual execution times, but as shown in Tables I and II the variations are quite small and in our view do not outweigh the performance gains obtained through our parallelization technique. One possible explanation for the fact that virtual execution times are consistently lower in distributed simulations than in sequential simulations is that contention modeling is skewed because, in the case of the parallel executions, at any given time some of the messages are stored in the physical network buffers rather than in the virtual network, therefore partially reducing the negative effects of contention. Further experiments are needed to verify this hypothesis. We find these experiments very encouraging since we were able to speed up our simulations' turnaround time from several days to hours, with only a slight degree of timing variance in the simulated time; other performance parameters, such as bus utilization and average memory access performance, although not reported, were also minimally affected.

5 Conclusion

We have proposed a novel simulation methodology that introduces a minimal amount of overhead in the parallel simulation of parallel systems. This new methodology can be used to effectively simulate large parallel computers executing real-life applications. The increase in simulation power has enabled us to begin studying the interaction of our hardware design with the operating system. This fact is of paramount importance because it gives the designer a much better picture of the whole design and allows testing of system-level design issues, which in most cases are left untouched until the final realization. An important consequence of our complete design virtualization is that network simulations can be flexibly distributed across all the hardware nodes, thus permitting a fast distributed simulation of arbitrary network topologies. Our methodology is realistic because, although virtualized, the execution is ordered by the synchronization interactions of a real parallel system. This improves the robustness of the evaluation process of a given design by embedding it in a real environment. We have directly benefited from this desirable attribute in the process of evaluating our technique; we found two instances in our thoroughly debugged applications where data races causing inconsistent results surfaced only when the applications were simulated in parallel. Relative ordering of simulated operations is truly asynchronous, shortening the path for detecting subtle synchronization problems. For example, it becomes more difficult to make wrong assumptions about the atomicity of message delivery or deterministic scheduling behavior, as it can happen in a sequential simulation.

We plan to experiment more with this methodology on a variety of simulated parallel architectures to measure the effects of varying the granularity of sharing and the computation to communication ratios. We also plan to measure the variance of simulation results for a wide variety of relevant real-life parallel applications running on each simulated architecture to verify our view that these are of minor significance and therefore are an acceptable characteristic of our simulation methodology. Finally, we plan to explore the application of our methodology to higher-level symbolic simulations and to lower-level simulation of hardware designs specified in VHDL.

References

[1] S. Adve, M. D. Hill, "Weak Ordering— A New Definition", Proc. 17th Annual Symposium on Computer Architecture, May 1990.

[2] W. C. Athas, C. L. Seitz, "Multicomputers: Message Passing Concurrent Computers", IEEE Computer Magazine, August 1988.

[3] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, R. Stevens, "Portable Programs for Parallel Processors", Holt, Reinhart and Winston, Inc., 1987.

[4] K. M. Chandy, J. Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations", Communications of the ACM, April 1981.

[5] A. Costa, A. De Gloria, P. Fababoschi, M. Olivieri, "An evaluation system for Distributed-time VHDL Simulation", Proc. of the 8th Workshop on Parallel and Distributed Simulation 1994, Edinburgh, Scotland, UK.

[6] M. Dubois, C. Scheurich, F. Briggs, " Memory Access Buffering in Multiprocessors", Proc. 13th Annual International Symposium on Computer Architecture, June 1986.

[7] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, J. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors", Proc. 17th Annual Symposium on Computer Architecture, May 1990.

[8] D. Jefferson, "Virtual Time", ACM Trans. Programming Languages and Systems, July 1985.

[9] L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs", IEEE Trans. on Conputers C-28(9):241-248, September 1979.

[10] S. Narain, R. Chadha, "Symbolic Discrete-Event Simulation", Discrete-Event Systems, Manufacturing Systems and Communication Networks, Editors: P.R. Kumar and P. Varaiya, LNCS, Springer Verlag 1994.

[11] H. Schwetman, "Csim: A C-based, Process-oriented Simulation Language", MCC Technical Report.

[12] D. Shasha, M. Snir, " Efficient and Correct Execution of Parallel Programs that

Share Memory", ACM Trans. Programming Languages and Systems, April 1988.

[13] C. Weems, E. Riseman, A. Hanson, "The DARPA Image Understanding Benchmark for Parallel Computers". J. Parallel and Distributed Computing, 11(1), 1991.

[14] B. Crothers, "Multithreading Gets Lost on P100 Systems", Infoworld, cover page, November 1994.

[15] P. Lincoln, J. Meseguer, L. Ricciulli, "The Rewrite Rule Machine Node Architecture and Its Performance", Proc. Conpar 94 - VAPP VI, Lintz, Austria, LNCS 854.