A Technique for the Distributed Simulation of Parallel Computers

Livio Ricciulli

Computer Science Laboratory

SRI International

Menlo Park, California 94024, USA.

Phone: 415-8592969

Fax: 415-8592844

Email: livio@csl.sri.com

Keywords: Parallel simulation, Distributed simulation, Parallel computer simulation

Submitted to MASCOTS' 95

A Technique for the Distributed Simulation of Parallel Computers

Abstract: A new technique for the efficient asynchronous discrete event-driven simulation of parallel shared-memory computers is proposed. Our execution-driven methodology, while introducing minimal synchronization overhead to maintain a coherent distributed event causality relation, allows complete virtualization of the design at all levels and therefore is very flexible. We give a detailed description of the proposed technique together with some preliminary performance results we obtained by implementing our parallelization technique on a CM-5 computer.

1 Introduction

Distributed Shared Memory (DSM) computers constitute an important subset of parallel machines because they allow the software application developer to have a simple, abstract view of the underlying hardware. Complex hardware and software mechanisms support consistency of the global state of the distributed system. Coherence mechanisms, without programmer intervention, keep track of all the global state transitions and execute a protocol to maintain a consistent partial order of memory access events. The amount of user intervention, the kind of coherence protocol, the granularity of sharing and cost performance trade-offs are some key design issues that are being intensely investigated today. Most studies today are performed with a quantitative approach because this allows the designer to formulate hypotheses and to prove or disprove them in a systematic manner, thus allowing an exploration of the design space based on precise observable and reproducible measurements. Simulation is the most direct and inexpensive tool to understand and measure the behavior of complex systems. It can be used to directly evaluate design alternatives without having to build costly prototypes. As systems being designed and simulated grow both in complexity and size it is necessary to scale the simulation environment to accommodate the increased performance requirements. In this paper we propose a novel methodology to perform efficient distributed event-driven simulation of DSM machines exploiting the existing computational power offered by today's parallel computers. Our approach is based on the execution-driven simulation methodology and affords unprecedented efficiency by allowing the simulated application synchronization operations to directly synchronize the distributed event-driven simulation.

2 Background

In this section we will define some metrics for evaluating different simulation approaches and apply them to evaluate currently used simulation techniques.

2.1 Metric

The desirable attributes of a simulation environment in our opinion are:

1 Speed.
2 Flexibility.
3 Realism.

Simulation performance is a key issue because it can greatly impact the design process. Incremental improvements of a design are sequential and therefore are intimately tied to the turn-around time of simulations. Performance is also closely related to the size of the simulation and therefore has a direct effect on how big a system can be studied.

Flexibility is necessary to permit a design space to be extended as needed and therefore to allow new discoveries to be made. A simulation environment which is targeted for a particular class of designs will not allow the exploration of alternative options and therefore will restrict the designer.

Realism is necessary to keep the designer at a close enough distance between what is being designed and simulated and a real product which could be derived from the experiments.

2.2 Current Approaches to Computer Simulation

We will mention some of the most popular approaches that have been followed to simulate parallel machines and evaluate them in regard to the three attributes we described above.

Prototyping

This technique is hardly a simulation methodology. The obvious drawbacks of this methodology are its high cost and very limited flexibility. The USC testbed [1] for the study of cache coherent DSM machines is an example of a project which tries to increase the design flexibility by making heavy usage of Field Programmable Gate-Arrays. Although this methodology is useful for the careful exploration of a particular design space, its high cost and lack of generality make it a controversial solution. We feel that this technique is only applicable when a research effort is extremely mature and therefore the design search space is greatly restricted.

Direct Execution

This methodology allows the simulated system to be directly mapped to the hardware of an existing general purpose host. In this framework the virtual and the real processors are overlapped as much as possible to reduce the ratio of the number of real instructions executed per simulated instruction. The Winsconsin Wind Tunnel [2] is an example of this technique. The virtual processors are replicated across the nodes of a CM-5 and consistency is maintained by inserting synchronization rendezvous every 100 instructions thus yielding a lock-step parallel simulation. Although this approach is more flexible than prototyping, we feel that it is neither flexible nor fast enough for today's simulation needs. Only systems with an instruction set and architecture similar to the one of the host nodes can be simulated, therefore restricting the design search space. This methodology also requires a great amount of synchronization overhead to keep the virtual and real parts of the machine synchronized.

Sequential Execution Driven Simulation

This technique allows the simulation of a parallel system on a sequential machine by interleaving the execution of the different simulated processor in an optimal way [3] [11]. This widely used approach is very flexible and can be made quite realistic by carefully choosing the execution parameters. The only drawback of this technique is that it is not scalable to the simulation of very large systems or systems running "real-life" parallel applications.

Trace Driven Simulation

Trace driven simulation is quite fast and can be very useful to analyze the behavior of the memory subsystem of a given computer. Trace driven simulation is the least flexible technique because it hides all the details of both the system which is being simulated and the program being executed and therefore greatly limits the scope of the design exercise.

3 Asynchronous Distributed Event Driven Simulation

Distributed event driven simulation has been the subject of extensive studies for the past 15 years and has evolved into simulation schemes of extreme sophistication and complexity [14][15]. The modular nature of DSM machines allows us to greatly simplify these mechanisms to yield a highly asynchronous distributed event driven simulation methodology with great parallelism. Each node of the simulated machine is mapped to a Logical Process (LP). All LPs are homogeneously distributed on the nodes of the real hardware machine. Each LP keeps its own local time and executes the simulated parallel application within the context of execution driven simulation.

We can make an analogy between distributed event driven simulation and the execution of parallel programs on shared memory machines. We associate shared memory accesses to events that propagate across LPs. Current distributed event driven simulation schemes can be viewed as sequentially consistent according to the definition of sequentially consistent systems given by Lamport [4]:

Definition 1.

[A system is sequentially consistent if] the ***result*** *of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*

We recognize that Definition 1 is too restrictive to study the behavior of parallel computers. We have highlighted the word ***result*** because depending on how one interprets this word different kinds of simulation methodologies can arise. In case the observable behavior of interest is the ordering of all the simulated events of the distributed entities, the simulation needs to be totally ordered of the kind proposed by Chandy and Misra [9] or Jefferson [10]. In case the observable behavior of a system is not the ordering of events itself but it is its overall performance, we can greatly simplify the simulation methodology. We observe that it is inefficient to simulate an asynchronous system (a parallel computer) with a synchronous methodology either with parallel lockstep execution or with a serializable execution. We abandon the synchronous world and, unlike in other distributed simulation techniques, we maintain the causality of events only to preserve the correctness of the simulated application and not to emulate a sequential execution. This idea has appeared in the context of parallel programming as the weak ordering consistency programming model [5] where memory update messages need to be ordered only with respect to synchronization operations. In that ordering, coherence messages are sent only to maintain a correct execution rather than to emulate a serialization of global memory accesses.

We augment the simulated application synchronization primitives with directives to enforce a consistent causality relationship between the different distributed events. The efficiency of our approach derives from the fact that the virtual devices need to communicate only data and synchronization messages produced by the application being simulated and nothing more.

A very important consequence of our LP partitioning is that there is a one to one correspondence between simulated shared data movement and events which propagate between LPs. On the basis of this property our scheme can be viewed as a *weakly ordered* simulation scheme where simulation events appear sequentially consistent only to the application being simulated. As in the weakly ordered consistency model we distinguish between global data access events and synchronization events.

1 Global data access events are propagated between LPs to simulate data movement.

2 Synchronization events are propagated between LPs to control the execution of the LPs. These are all those events that could cause an LP to either be blocked or suspended or cause an LP to resume from a blocked or suspended state.

In some cases events devoted to data access are also synchronizing according to definition 2; in those cases we treat these events as synchronization events. For example a cache miss will cause a block fetch which carries data and also causes the missing LP to block.

In weak ordered distributed event-driven simulation global data accesses events have an explicit causality relation only with synchronization events and not among themselves. This model allows much more parallelism than previous ones because global data access events do not need to be ordered but are allowed to propagate asynchronously and therefore do not introduce overhead to enforce a sequential causality.

In order to state the necessary condition for the correctness of weak ordering simulation we need to introduce some definitions:

Definition 1. *Two global access events are* ***conflicting*** *if they access the same global location and at least a one of them is a write [6]*

Definition 2. *A simulation is* ***race free*** *if any two* ***conflicting*** *global data access events have at least one proper synchronization event between them. [7]*

We can say that a weakly ordered distributed simulation is correct if:

All synchronization events are sequentially consistent

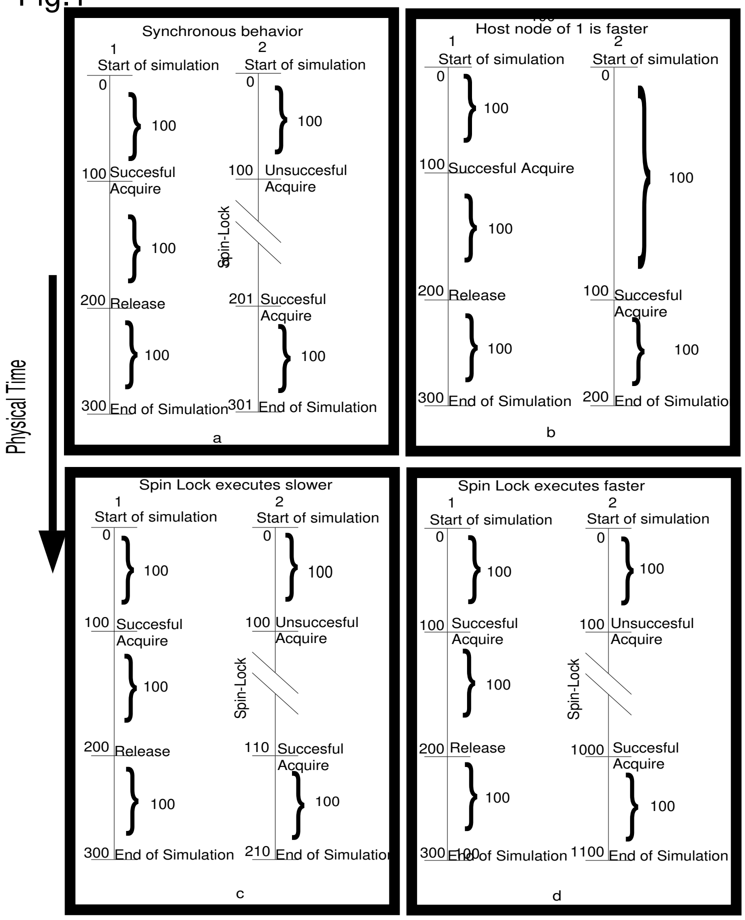
The simulation is race free

Since there is a one to one correspondence between shared data movement and global simulation events we can say that our simulation is correct if the system we are simulating is data-race free and its synchronization operations are sequentially consistent. This is of paramount importance because it lifts from the simulator the responsibility of enforcing a total order of simulation events by assuming that the application being simulated is data-race free. Although this might seem restrictive to a particular set of applications it should be noted that many parallel applications written today have these requirements and therefore would execute correctly within this paradigm.

4 Clock Synchronization

Correctness of the execution of the simulated application is not enough to make a simulated methodology usable. The simulation model described in section 3 does not provide a correct estimate of the execution time. The different LPs execute asynchronously and, although producing the correct data, will suffer from significant virtual clock skews introduced by busy-waiting operations. Fig. 1 depicts such phenomenon.; in all the examples LP 1 acquires a lock and processor 2 busy-waits on the lock. After completing the mutual exclusion section the processors are synchronized by a barrier before termination which takes the maximum of the execution times as the total execution time. Depending on the relative speed of the LPs and the host nodes, at the completion of the barrier, the two LPs might have an erroneous estimate of the simulated time which does not reflect a proper event causality relation. In Fig. 1.a the two processors behave synchronously and do not need clock adjustment. Fig. 1.b shows what would happen if the

Fig.1



host node of P1 is faster and allows P1 to complete the mutual exclusion section before P2 issues an acquire.

In Fig. 1.c P2's spin-lock executes faster and therefore P2's virtual time advances too much. In Fig. 1.d P2's spin-lock executes more slowly and its virtual time is therefore two low. In cases 1.a and 1.b we do not perform clock adjustment. In cases 1.c and 1.d we need a mechanism to synchronize the local clock of LP 2. In our methodology the clock synchronization solution is simple because the simulated application will enforce an ordering of events which will never need to be undone. Unlike in time warp mechanisms [10], we only modify the virtual clocks without having to restore previous states. In addition, 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. This can introduce some variance of the simulated execution time (as Fig. 1.b) but in our opinion if such an ordering was necessary, it should be enforced through the simulated application rather than by the simulator. This is analogous to the approach that it would be followed when running the application on a real machine.

We notice that the only instance when the virtual time of an LP gets out of synch is when the LP is blocked; therefore, given a particular ordering of synchronization operations, we synchronize blocked LPs by updating their virtual time to respect a causality relationship with the event that is causing the blocked LP to resume execution.

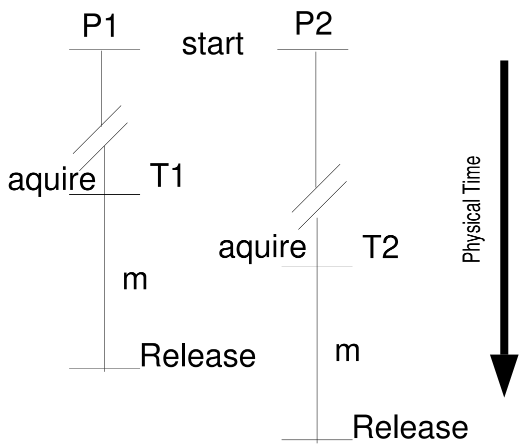
Assuming that:

1 LPs cannot change their virtual time except at the acquire of a resource for which they were blocked

2 LPs cannot change other LPs virtual times

3 A blocked process will not execute an acquire

The following accomplishes clock synchronization of two or more processes which



contend for a resource.

The virtual time of a blocked process acquiring a resource is the maximum of:

The virtual time of the blocked process when it first tried to acquire the resource for which it is blocked and the virtual time of the releasing LP at the release of the resource plus the propagation delay of the release-acquire messages.

Here is a simple argument to support the 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. P2 local clock must  respect a causality relationship with
the event order observed (P1 acquires the lock before P2). If
T1+m1<T2 than the causality relation is maintained without adjust-
ing T2. If T1+m1 >= T2 then the causality is maintained if T2 is
set to T1+m1+delta.  The total execution time for both processors
to complete their mutual exclusion sections is:

Texec =  T1+m1+m2+delta  if T2<T1+m1+delta or
Texec =  T2+m2           if T2>T1+m1+delta.
or more compactly Texec=Max(T2,T1+m1+delta)+m2

Since LPs can only change their clocks at the acquire of a resource
then it follows that for the total execution to yield
Max(T2,T1+m1+delta)+m2, P2 must  adjusts its clock at the beginning
of m2 to Max(T2,T1+m1+delta).
```

This will always work because an LP issuing a releasing event cannot be blocked and therefore must have a correct virtual time.

The above idea is quite general and can be used to synchronize virtual times on any event which causes a blocked LP to resume useful execution including, for example, clear operations on lock variables. This method could be sufficient to synchronize in a distributed manner the local virtual clocks of any system with synchronization primitives based on lock and clear operations (for example using the ANL macros [8]). To make this technique independent from particular synchronization schemes, however, we lift the mechanism one step up in the level of abstraction and introduce a central controller which will exchange messages with the LPs and determine a correct causality relation among synchronization events and will supply a correct estimate of the execution time.

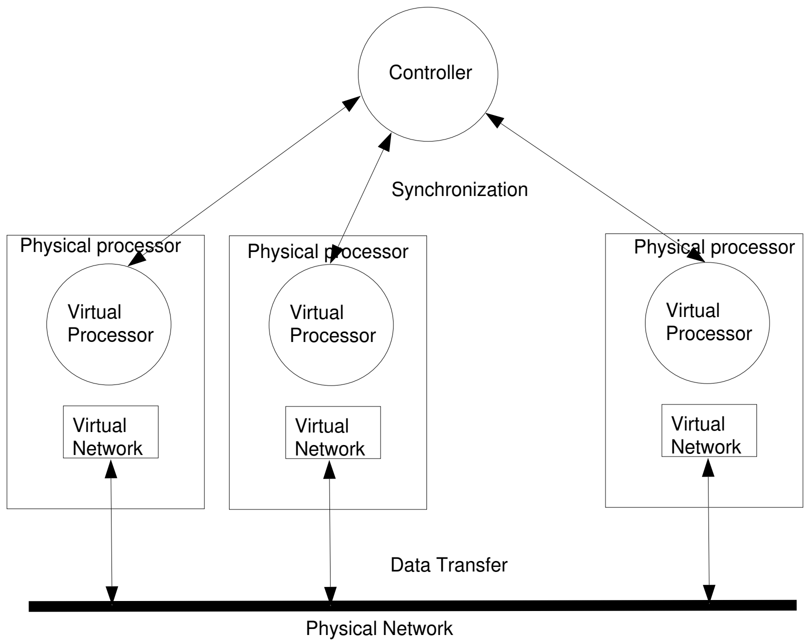


Figure 2 schematically illustrates our simulation methodology. The central controller implements our virtual time synchronization scheme and sends messages to the synchronizing LPs to adjust their clocks. The controller is only invoked when the virtual processors execute synchronization operations. All data movement between virtual processors happens directly between the physical nodes without the controller intervention.

As LPs enter a blocking synchronization primitive they notify the controller of their 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 by 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.

Figure 3 details how clock synchronization is maintained when a barrier is executed.

```
 Processor            Controller

barrier(p,barvar)     barrier(var)
```

```
{                      while(1) {
INC(barvar)            receive(p0,time[p0])
send(p,time)           increment(var)
while(barvar<N)nop;    if (var==N)for(p0=0;p0<N;p0++)
receive(p,local_time)             send(p,maximum(time[1..N])+delta)
 }                      }
```

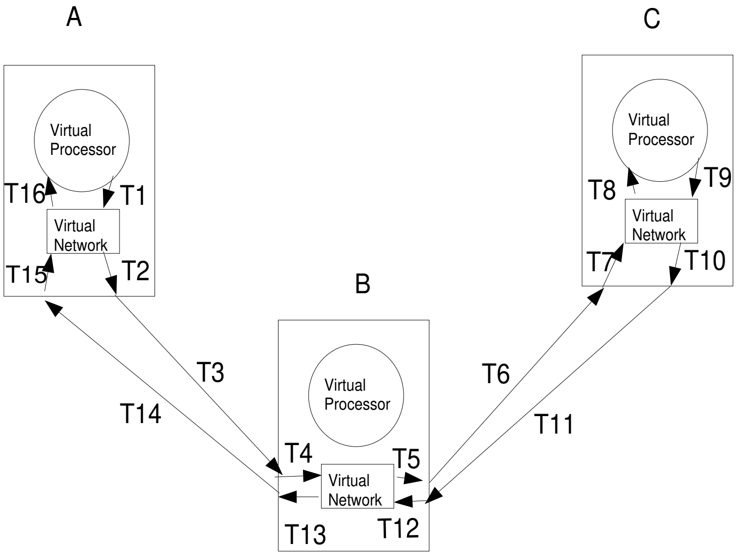
Where:
N is the number of virtual processors in the system.
barvar is the global counter used to implement the barrier.
INC is an atomic increment instruction.
p is the LP number issuing the barrier.
v*ar, p0* and *time[1..N]* are local variables of the controller and delta is a delay due to release acquire messages.

All virtual processors notify the controller of their local time 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 received.

Figure 4 details how clock synchronization can be maintained for mutual exclusion synchronization of two processors. The technique can easily be extended for multiple processors:

```
   Processor                    Controller
entermutex(p,i)              Lockcontrol(i)
{ waited=FALSE;              {
  send(p,i,time,LOCK);
  while (!TAS(lock[i])         receive(p0,i,time1,oper);
     waited=TRUE;              receive(p1,i,time0,oper);
  if (waited==TRUE)
  {
    receive(p,local_time);     if(oper==LOCK)
    Adjust_time(local_time);   {
  }
 }                               receive(p0,i,time0,oper);
 exitmutex(p,i)                  send(p1,max(time0,time1)+delta);
 {
                               }
   send(p,i,time,UNLOCK);      CLR(var[i]);
   CLR(lock[i]);
 }                            }
```

Where
i is the index of a global array *lock* used to implement mutual exclusion,*p* is the LP number issuing the operations, *waited* is a local variable, *Adjust_time* is a function that adjusts the local time of a processor*, TAS* is a test & set operation, *var[1..NLOCKS], p0,p1,time0,time1,oper* are local variables of the controller and delta is a delay due to release acquire messages.



As in the barrier case, virtual processors notify the controller when they enter a synchronization primitive and when exiting ask for the correct local time.

5 Virtual Network

The real network characteristics are masked from the simulation with a simplified version of the distributed event-driven simulation methodology proposed by Chandy and Misra [9]. We assume that the hardware network will be able to deliver messages between any of the hardware nodes but we do not make any assumptions on how long the messages take to propagate.

Fig. 5 depicts a read operation invoked by processor A routed through processor B to fetch a value from processor C. Each message is time stamped with the virtual 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.

We ignore T3, T6,T11,T14 and therefore mask the effects of the real hardware network. In contrast with Chandy and Misra's scheme, we only use timing information which measures relative propagation delays through the different parts of the simulated system and we always ignore the local time of the virtual devices the messages propagate through. For example processor B will add T4,T5,T12 and T13 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 with the propagation delays T7,T8,T9,T10.

In case a message reply is needed (as in the example above), the timestamp of the reply will reflect the time the returned message should have arrived to the original sender in its own time reference. At this point the method described in section 3 can be used to synchronize the clock of the original sender. Notice that, unlike a true time warp mechanism, we do not need to send anti-messages or restore states because coherence is assumed to be guaranteed by the simulated application synchronization; also, if the sender was not blocked the time stamp is discarded because we can assume that the time of the delivery of the message would only have an effect in a race condition situation and therefore is relevant only to the correctness of the simulated application rather than the to simulation itself.

6 The Trade-off

Because of the non-determinism introduced by the run-time behavior of the host, different execution behaviors might be observed for the same simulated application. We argue that this a desirable attribute rather than a flaw because it narrows the gap between what is being simulated and a real world implementation. Why should we introduce a great amount of overhead to make the behavior of a simulated parallel system deterministic when in reality this will never happen? An answer to that question is that determinism allows repeatability. This attribute however can lose its appeal if, for example, the execution behavior varies by 1% while the simulation runs 50 times faster. In addition, the amount of variance in behavior of several executions will give extra information to evaluate a particular application's dependance on run-time behavior. We acknowledge that repeatability is sometimes indispensable for debugging purposes, and for those cases a sequential deterministic simulation is the only alternative, but we also note that it is always possible to introduce extra synchronization in a parallel simulated application to restrict non-deterministic behavior and therefore to isolate problems. We plan to further investigate this important aspects in future studies.

7 Experimental Evidence

Here we report some initial experimental evidence to support the effectiveness of our approach.

7.1 RRM simulator

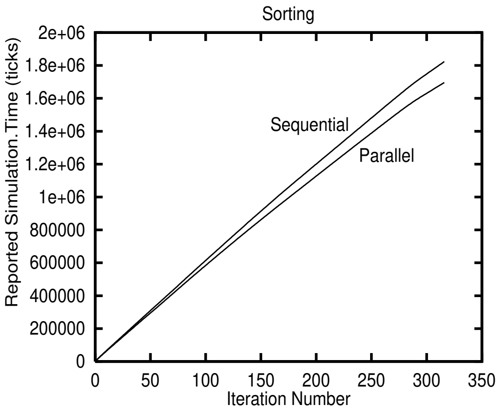
We have applied our ideas to the simulator for the Rewrite Rule Machine. The Rewrite Rule Machine (RRM) [12] is a Multiple Instruction Multiple Data/Single Instruction Multiple Data Stream (MIMD/SIMD) massively parallel computer currently being designed and simulated at SRI International. The simulator 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 [13]. This package is an extension of the C language which 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(mail-boxes). This simulation scheme is very similar to a Verilog Hardware Description Language (VHDL) type of behavioral simulation. 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. The parallelization of the RRM simulator was achieved by spawning multiple Unix processes each containing a Csim simulator and allowing the processes to communicate through Unix pipes. This methodology was used for the initial development of the parallel simulator and provided encouraging initial results on a 4-processors SPARC-server 670 MP yielding an efficiency up to 75%. A second version of the simulator was ported to the CM-5 computer in a host/node configuration were each CM-5 node holds one or more LPs and the CM-5 host simulates the simulator synchronization controller.

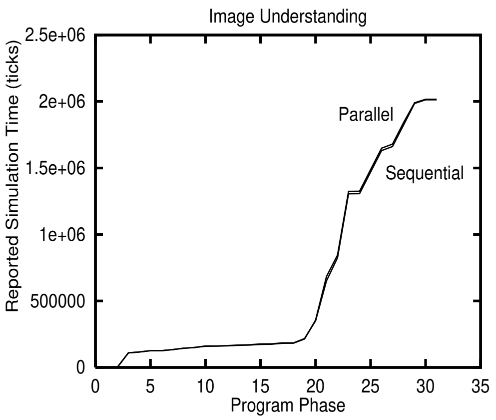
7.2 Comparison of sequential VS. Parallel

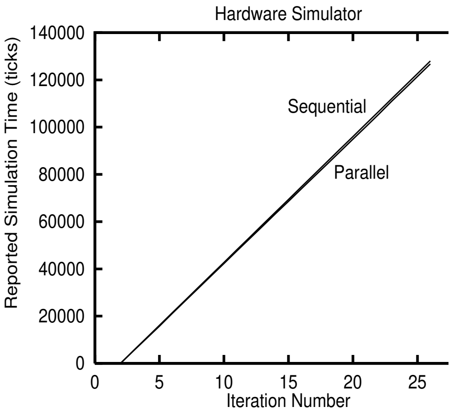
We have evaluated our methodology on the simulation of three applications: Sorting, Hardware gate level simulation and the (D)ARPA image understanding benchmark for parallel computers. All these applications follow the Single Program Multiple Data paradigm and were coded in RRM assembly language. The table below reports the performance of our parallel simulator on a 32-node CM-5 with one LP per physical processor. The SPARC-10 execution time is taken by executing the same simulator with 32 LPs mapped 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. The porting of the simulator from the Unix SPARC server to the CM-5 was quite straight forward thus pointing out that our approach is quite general and machine independent.

(sec.)	cm5	sparc-10	speedup
sorting	11247	69355	6.2
IU	6659	42481	6.4
hw sim	2601	9668	3.7

Notice that the speedup is calculated over the execution time of a SPARC-10. If one took into account the fact that the SPARC-10 is about twice as fast as each of the CM5 nodes, our speedup figures would double. The total size of the sequential simulator was about 30-35 Mbytes and we suspect it had quite 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 and therefore will give advantage to the parallel simulator. The hardware simulator has worse behavior because the test vectors for the gate level design simulation are stored in a Unix file therefore creating an access serialization through the CM-5 host.







These graphs depicts the reported simulated execution times of the sequential and parallel simulations. The fact that the parallel simulation is not totally ordered causes some small variations in the reported execution times but as one can see from the graphs the variations are quite small and in our view do not outweigh the performance gains obtained through our parallelization technique. Sorting shows the worst variation because most of the simulated time is spent waiting for intra-node communication resources and contention modeling is skewed by the fact that at any given time some of the messages are stored in the hardware network buffers rather than in the virtual network therefore partially reducing the negative effects of contention. Further experiments are needed to verify this hypothesis.

8 Conclusion

We have proposed a novel simulation methodology which introduces a minimal amount of overhead in the parallel simulation of parallel systems. This new methodology can be used to simulate large DSM systems without incurring in trashing. An important consequence of our complete virtualization is that the network simulation can be 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 desirable attribute will contribute to improving the robustness of the evaluation process of a given design by embedding it in a real environment. Relative ordering of simulated operations is truly asynchronous, shortening the path for detecting subtle synchronization problems. It will be difficult, for example,to make wrong assumptions about the atomicity of message delivery or deterministic scheduling behavior as can happen in a sequential event-driven simulation.

We would like to experiment more with this methodology on large distributed systems to measure and improve its performance. We also would like to measure the variance of simulation results for a wide variety of relevant real-life parallel applications to verify our view that these will be of minor significance and therefore are an acceptable characteristic of our simulation methodology.

References

[1] M. Dubois, L. Barroso, S. Iman, J. Jeong, K. Oner, K. Ramamurty. "The USC Multiprocessor Testbed project: Project Review", USC Technical Report CENG-94-15, Department of EE-Systems, University of Southern California, Los Angeles, CA90089.

[2] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, D. A. Wood. "The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers", Proceedings of the 1993 ACM SIGMETRICS Conference", May 1993.

[3] M. Brorsson, F. Dahlgren, H. Nilsson, P. Stenstrom. "The CacheMire Test Bench --- A flexible and Effective Approach for Simulation of Multiprocessors", Technical Report, Dept. of Computer Engineering, Lund University, Sweden, September 1992.

[4] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs", C-28(9):241-248, September 1979.

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

[6] D. Shasha, M. Snir. " Efficient and Correct Execution of parallel programs that share memory", ACM Transactions on Programming Languages and Systems, April 1988.

[7] S. Adve, M. D. Hill. "Weak Ordering- A New Definition", Proceeding of the 17th Annual Symposium on Computer Architecture, May 1990.

[8] 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.

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

[10] D. Jefferson, "Virtual Time", ACM Transactions on Programming Languages and Systems, July 1985.

[11] H. Davis, S. R. Goldschmidt, J. Hennessy, "Multiprocessor Simulation and Tracing Using Tango", Proc. of the Par. Proc.Conf., June 1990.

[12] P. Lincoln, J. Meseguer, L. Ricciulli, "The Rewrite Rule Machine Node Architecture and its Performance", to appear in proceedings of CONPAR 94, Linz, Austra.

[13] H. Schwetman "Csim: A c-based, process-oriented simulation language", MCC Technical Report.

[14] R. M. Fujimoto, "Optimistic Approaches to Parallel Discrete Event Driven Simulation", Transactions of the society for computer simulation, June 1990.

[15] R. M. Fujimoto, " Parallel discrete event simulation", Communications of the

ACM, Vol. 33, October 1990.