Parallelized Network Simulators for Message-Passing Parallel Programs

Phillip M. Dickens
ICASE
NASA Langley Research Center
Hampton, VA 23681

Philip Heidelberger*
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

David M. Nicol†
Department of Computer Science
The College of William and Mary
Williamsburg, VA 23185

Abstract

This paper describes a software tool, LAPSE (Large Application Parallel Simulation Environment), for the simulation of large-scale programs written for the Intel Paragon, executed on smaller-scale machines. Two timing simulators are themselves parallelized; this paper describes the conservative synchronization algorithm we have developed to synchronize the simulation. We present performance data taken from LAPSE runs of an indirect linear system solver where we consider issues of simulated network complexity, and processor usage.

1 Introduction

All parallel codes have to be fine tuned by their programmers to obtain maximum performance. Since modern large-scale parallel machines simultaneously support many users running in independent partitions, performance tuning of this sort can be impractical for codes run (in production mode) on a large fraction of a machine's nodes. Likewise, one may wish to investigate performance on machines larger than any yet built, or using communication networks that have not yet been built. For these reasons, and others, there is benefit in a capability of simulating the execution of an application on a large *virtual machine*, using a smaller physical machine. There are also compelling reasons for simulating a virtual machine using the same size physical machine, to support intrusion-free instrumentation, or experimentation with (simulated) operating system policies.

This paper describes a software tool, LAPSE, for the direct-execution simulation of SPMD (Single Program Multiple Data) applications written for the Intel Paragon. For every processor in the simulated system LAPSE creates a Unix process that directly executes the code for that processor. All interactions of that code with the message-passing library of the virtual machine are trapped, and simulated. Of particular importance is the simulation of the communication network behavior, and operating system management of communication functions. To achieve high performance, LAPSE uses parallel processing. It distributes the set of application processes among nodes in a physical Paragon, and also distributes a set of simulator processes responsible for simulating the timing behavior of the application processes. Since the timing simulator is itself parallelized, we must pay special attention to the synchronization issues involved. LAPSE uses a new protocol, *Whoa* (Window-based halting on appointments) designed especially for this simulation problem domain. The protocol is used with two different network simulators, one that does not model contention, and one that does. In this paper we describe LAPSE's general structure, the *Whoa* protocol, and our computational experience with the tool.

A number of other direct-execution simulators for simulating parallel programs have been developed. The first (in 1983) was SIMON [6], which simulated message-passing programs. Like SIMON, RPPT [3] is a serial simulator for message-passing codes, but here the emphasis is on its use to evaluate network designs. A recently described tool for message-passing codes, HASE [7], focuses on the software one uses to design the simulator, leaving the simulation details to a commercial tool (by Jade Simulation). Jade's simulation engine supports parallel processing, although at the time of publication HASE had not yet been integrated with it. Likewise, the designers of Maya [1] intend to parallelize it, noting that its network simulator is the bottleneck. An entirely different set of tools simulate shared memory programs e.g., [2, 4, 9]. Among these, Proteus [2] also supports message-passing networks, and the Wisconson Wind Tunnel [9] is parallelized. The unique contribu-

*This work was performed while this author was on sabbatical at ICASE.

†This work was performed while this author was on sabbatical at ICASE. It is also supported in part by NSF grant CCR-9201195
.

tion LAPSE makes to the field is to create an efficient synchronization mechanism for parallelized simulation that takes advantage of message-passing codes' structure. This paper sketches the tool, discusses the addition of a new network simulator within the LAPSE framework, and then considers issues related to using the tool well.

2 LAPSE Usage

LAPSE (with only the no-delay network) is described more fully in [5]. Here we sketch its structure.

To use LAPSE a programmer simply modifies the code's "make" file to call LAPSE scripts instead of compilers, and sets up a file specifying LAPSE parameters such as the number of simulated and the number of actual nodes. The LAPSE system transforms copies of the code's files (which may be in C, or Fortran, or both) into a set of "application" and "simulation" processes. Each actual node multitasks one simulator process and multiple application processes. For every node in the simulated system, LAPSE creates a unique application process that executes functionally exactly the way the application code would in the simulated system (a technique called direct execution simulation). However, wherever the original application calls a message-passing system routine (e.g., a send, receive, probe, or clock call), the LAPSE modified application calls a LAPSE routine to perform that action, and to report it to the network simulator. The net effect is that the application processes communicate exactly the same messages as they do in the larger system, but the LAPSE simulation processes simulate the messaging activity in the simulated larger system. In doing so, the LAPSE simulators provide temporal information about the simulated system. For instance, whenever the application process calls a system clock (as it will when performance tuning), LAPSE will return to the application the time in the *simulated* system.

To help capture temporal information LAPSE gets execution time information from application processes. At compile-time LAPSE modifies application assembly code, inserting instruction counters at basic block boundaries. Whenever an application process calls a LAPSE routine, LAPSE recovers the number of instructions executed since the last call to a LAPSE routine. This provides us with a measure of the time spent executing the application in the simulated system. This measure, along with a description of the requested activity, is sent to the LAPSE simulation process responsible for this application process. The simulation processes are distributed across the actual system, one per node (usually, with exceptions discussed later), each one being responsible for simulating the timing behavior of all application processes also assigned to that node. Delicate synchronization between simulation processes is required since the simulation in one process can affect and be affected by activity in other simulation processes. However, one of the key insights identified by our work is that in many codes of practical interest LAPSE enjoys excellent "lookahead" capabilities since the application processes can be run largely as on-line trace-generators with little synchronization with the simulation processes. This has allowed us to develop a simple synchronization protocol based on windows (in simulation time), and appointments (dynamically established and updated lower bounds on times of future message send events).

3 Whoa

Our synchronization algorithm is based on the notion of "windows" in simulation time, and on the notion of *appointments*. An appointment [8] at time t from a processor i to processor j is a promise by processor i to processor j not to cause any event to be scheduled at j with time-stamp t or less. If a processor i can compute a lower bound b on the time of the next event to be scheduled at i, by any different processor, then i is free to execute all known simulation events up to time b. In LAPSE the joint application of windows and appointments allow processors to compute such lower bounds. We consider the construction/maintenance of appointments and windows separately.

Every LAPSE simulator process is responsible for simulating the timing behavior of a set of virtual processors (VPs). As the application processes call the interface, reporting execution bursts and virtual machine calls, the interface reports these to the appropriate LAPSE simulator. The simulator constructs for each of its VPs a *time-line* of future events. As the events are reported, initial time-stamps on the events are computed based on the reported bursts of execution time, and known lower bounds on times required to execute virtual system calls. To compute an appointment on the time that a future message might be sent, one need only analyze the time-line, summing execution bursts and overhead lower bounds. As the simulation progresses, time-stamps on future events may change, and so appointments may also change.

Whenever an application reports to the simulator a message send event involving inter-processor communication, the simulator computes a lower bound on the time of that message being received, and sends that bound to the recipient processor. This is an appointment for the event. As earlier events are simulated, that appointment may change. For instance, the simulation of a blocking receive will increase the times of future events for that VP by as long as the VP was suspended waiting for its message. A number of other circumstances modify the time-line event times, and hence the appointments. Under *Whoa*, changes in appointment times are reported as soon as the changes are discovered.

The only way LAPSE knows that one simulation process may affect another is by observation of an application message from which the interaction is derived. In order to account for the threat of interaction due to an as-yet-unobserved application message, *Whoa* constructs windows in simulation time. The basic idea is that if the simulation processes are all synchronized at simulation t, they choose a simulation time $w(t)$ which has the property that all future events whose time-stamps end up in $[t, w(t))$ have already been observed at the time $w(t)$ is computed. By globally synchronizing at time $w(t)$, simulation processes avoid advancing their clocks past a time of an interaction that was unknown at the point $w(t)$ was constructed. Within the window $[t, w(t))$, appointments are computed and maintained. A simulator maintains a *halting time* h, equal to the time of its minimum incoming appointment from a different simulator. The simulator halts (e.g., blocks) whenever its halting time is less than its current least-time event, *and* is less than $w(t)$. While halted the simulator accepts messages from other simulators, which ultimately increases the halting time and unblocks the simulator. Once a simulator's halting time is at least as large as $w(t)$ and the simulator executes all events known up to time $w(t)$, the simulator engages in the window construction protocol.

The basic idea behind the window construction protocol (invoked at time t) is to identify the largest time $w(t)$ at which we are assured (based on reported execution bursts and application events) that all events whose time-stamps end up in $[t, w(t))$ have already been reported (by the application processes) at the time the window is constructed. The computation is derived from information on each application's time-line. For each VP i that is not suspended at time t (e.g., waiting for a message) we compute a lower bound r_i on the execution time of the last known application event for VP i. For suspended VPs i we compute a lower bound s_i on the total execution time between the last known application event, and the VP's present state. For all VPs i we compute a lower bound n_i on the time when next the VP sends a message to any other. The processors cooperatively compute a global component-wise reduction on the vectors (r_i, s_i, n_i); let $r_{\min}$, $s_{\min}$, $n_{\min}$ be the results. Then

$$w(t) = \min\{r_{\min}, n_{\min} + s_{\min}\}.$$

4 Network Simulators

The basic Whoa algorithm suffices to safely synchronize a delay-only network model, because a message affects only the processor responsible for simulating the message's destination VP. To deal with contention we have developed a switch-level simulator that explicitly models packet contention for ports and queues in the Paragon switch. There is one switch per Paragon node, with bidirectional NEWS (North East West South) ports and with an entry port and and an exit port dedicated to the associated node. The packets of a message may then pass through multiple simulators, a problem for which the the synchronization strategy must account.

The synchronization strategy for the switch-level model builds upon Whoa. Exactly the same set of synchronization messages are sent, but with some additional logic, as follows. The appointment message a processor receives are seen as appointments on messages processors *have not yet released to the network*. Reports of message sends are seen as reports of messages entering the network. Processors synchronize as before to account for as-yet-unreleased messages. Further synchronization is needed to account for messages that are currently in the network that involve more than one simulation processor. Such a message is called a traveler. Towards this end the network is assumed to run synchronously, with all events occurring at multiples of the packet switching time Δ. Given the relatively large size of Paragon packets, Δ may be large; the experiments reported use the empirically determined value of $\Delta = 94.95$ μ-sec. At window construction time, every processor schedules an evaluation event at the earliest network clock time. The evaluation event does a global reduction to determine whether there are any traveler messages currently in the network and if not, a lower bound on the next instant when a traveler message may enter the network. If a traveler message is detected, the evaluation routine re-schedules itself Δ time in the future. If instead L is computed as a lower bound on the next traveler message entry time, an evaluation event at time L is scheduled. The overall effect is to have the network simulator time-step at increments of Δ when there is traffic crossing simulation processor boundaries, but to skip over periods of network idleness. While simple, the strategy takes advantage of the behavior of many scientific codes to proceed in alternating compute and communication phases. On the other hand, there is clearly room for more sophisticated optimizations.

5 Performance

Accuracy of the predicted performance is of course a critical concern. On a variety of codes we have found LAPSE to predict finishing time well within 10% of measured finishing times, running natively.

"Slowdown" measures how much more slowly an application runs using the simulator than it does running natively, as a measure of the simulation's overhead. LAPSE slowdowns depend a great deal on the characteristics of the application. In highly favorable circumstances we have observed slowdowns of only 1.2; in highly unfavorable circumstances we've observed a slowdown of 100. These slowdowns are actually low, compared to other slow-downs reported in the literature. "Speedup" is also an important metric, measuring the utility of using parallel processors. We have observed high speedups using LAPSE, for instance, a speedup of 47 using 64 processors (to simulate 256 processors).

In this paper we consider two issues, the extent to which the switch-level network simulator degrades performance, and the effect of using separate pools of processors for executing application processes, and simulation processes.

LAPSE's default configuration (given P processors) is to assign one simulation process to each, and evenly distribute the application processes among them all. One issue to examine is whether this policy always makes sense. The answer intuitively depends on the relative execution costs of the application processes and the network simulator. Thus, when approaching this question, we need to look both at different computation/communication ratios for the application processes, and look at different network simulators.

The experiment we report here uses LAPSE to simulate a code that solves the Poisson (heat) equation on the square, given well-behaved boundary conditions. The domain is partitioned into an 8×8 set of subdomains, assigned one to a (virtual) processor. Each iteration, each VP sends boundary values to and receives boundary values from its neighbors in the 8×8 mesh. Each VP then computes updates locally on its subdomain. At the end of the iteration the processors globally determine whether the solution has converged. The computation to communication ratio is controlled by a parameter G: each VP's subdomain is a $G \times G$ submesh. Increasing G increases the computation to communication ratio.

Our experiment uses a pool of 16 actual processors to simulate the 64 application processes, and varies (i) the number and distribution of the simulation processors, (ii) workload characteristics, (iii) the network simulator. We represent the results in terms of slowdown—the factor by which the simulator runs more slowly than the actual code, using 64 processors. These numbers are given in table 1. The notation 16 :: n means that 16 processors were dedicated to executing application processes, and n additional processors were dedicated to the simulation processes. 16 :: 0 and 32 :: 0 denote experiments where each processor hosted one simulation process and multiple application processes.

app :: sim	low comp./comm.		high comp./comm.	
	simple	switch	simple	switch
16 :: 0	32	348	8	18
16 :: 1	264	326	12	16
16 :: 2	104	260	7	10
16 :: 4	52	241	7	8
16 :: 8	36	236	7	7
16 :: 16	24	226	7	7
32 :: 0	24	230	6	18

Table 1: LAPSE Slowdowns on PDE solver as a function of simulation processors, application computation/communication ratio, and network simulator used. 64 Virtual processors are simulated.

The data shown helps us to understand the relative contribution the network simulation makes to overall performance. Consider first the experiments run with a low computation/communication ratio ($G = 50$). Here we expect the execution cost of the simulation processes to contribute heavily to the overall performance, and so anticipate that adding processors to their pool will decrease the slowdown. This is especially true for the simple network model. The performance gain between one simulation processor and two is disproportional, due to relieving of communication congestion. (And the tremendous performance penalty going from 16 :: 0 to 16 :: 1 is due to serialization of the network and communication with it. This is a measure of how strongly overall performance depends on that of the network simulator). Adding additional processors continues to significantly reduce the slowdown. The cases of 16 and 32 processors overlapping simulation and application processes bear special comment. The drop in slowdown between 16 :: 0 and 32 :: 0 shows, again, that the network simulator is the bottleneck. It is also interesting to compare 16 :: 16 and 32 :: 0, as both use 32 processors. There is little or no advantage to separating the simulation and application processes. Overall, the case of low computation/communication ratio and simple network argues for overlapping. The risks of fouling up a separated scenario are significant.

Now consider performance when the switch-level simulator is used, again under a low computation/communication ratio. Here a different story entirely emerges. First, the slowdowns are much larger than their simple network counterparts. Secondly, the marginal benefit of adding processors to the pool of simulation processors is much less. Finally, the previous argument for overlapping is completely negated. The best (in terms of cost-effectiveness) strategy appears to be using a small number (2 or 4) of dedicated simulation processors. The underlying reason is that simulation processors are much more tightly synchronized—they synchronize globally whereas in the simple network case global synchronization occurs only at window construction. Multitasking interferes with the synchrony (and the Paragon does not provide sophisticated gang-scheduling mechanisms that might alleviate this interference).

In the case of a high computation/communication ratio ($G = 250$) we are led to the same conclusions: use overlapping processes with the simple network model, and a few separated simulation processes with the switch-level model. However, for the simple network we reach the conclusion for a different reason. The application workload is so high that the network simulation—when done properly—scarcely affects performance. This is evidenced by the insensitivity to increasing numbers of simulation processors. Since the simple network simulation costs relatively little, it makes sense to overlap. For the switch-level network we again see small marginal returns for increasing network simulation processors, and the same deleterious effect due to interference by overlapping.

6 Conclusions

This paper discusses a software tool, LAPSE, which provides parallelized direct-execution simulation of codes for the Intel Paragon. We describe the overall structure of LAPSE, and describe the synchronization algorithms used for two parallelized network simulators. One simulator neglects the effects of contention, the other is careful to include it. We report on LAPSE's performance on a program that solves a partial differential equation, paying special attention to the effects on performance due to the network simulator used, and due to the strategy used in assigning processes to processors. We find that the strategy best suited for the delay-only simulator is to use all processors for both application and simulation processes, whereas the best strategy for the complex network is to dedicate a few processors to the simulation processes exclusively.

References

[1] D. Agrawal, M. Choy, H.V. Leong, and A. Singh. Maya: A simulation platform for distributed shared memories. In *Proceedings of the 8^{th} Workshop on Parallel and Distributed Simulation*, pages 151–155, July 1994.

[2] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS: A high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, September 1991.

[3] R. Covington, S. Dwarkadas, J. Jump, S. Madala, and J. Sinclair. Efficient simulation of parallel computer systems. *International Journal on Computer Simulation*, 1(1):31–58, June 1991.

[4] H. Davis, S. Goldschmidt, and J. Hennessy. Multiprocessor simulation and tracing using Tango. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages II99–II107, August 1991.

[5] P.M. Dickens, P. Heidelberger, and D.M. Nicol. Parallelized direct execution simulation of message-passing programs. Technical Report 94-50, ICASE, July 1994.

[6] R. M. Fujimoto. Simon: A simulator of multicomputer networks. Technical Report UCB/CSD 83/137, ERL, University of California, Berkeley, 1983.

[7] F.W. Howell, R. Williams, and R.N. Ibbett. Hierarchical architecture design and simulation environment. In *MASCOTS '94, Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 363–370, Durham, North Carolina, 1994. IEEE Computer Society Press.

[8] D.M. Nicol. Parallel discrete-event simulation of FCFS stochastic queueing networks. In *Proceedings ACM/SIGPLAN PPEALS 1988: Experiences with Applications, Languages and Systems*, pages 124–137. ACM Press, 1988.

[9] S. Reinhardt, M. Hill, J. Larus, A. Lebeck, J. Lewis, and D. Wood. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proceedings of the 1993 ACM SIGMETRICS Conference*, pages 48–60, Santa Clara, CA., May 1993.