Parallelized Direct Execution Simulation of 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

August 30, 1994

Abstract

As massively parallel computers proliferate, there is growing interest in finding ways by which performance of massively parallel codes can be efficiently predicted. This problem arises in diverse contexts such as parallelizing compilers, parallel performance monitoring, and parallel algorithm development. In this paper we describe one solution where one directly executes the application code, but uses a discrete-event simulator to model details of the presumed parallel machine, such as operating system and communication network behavior. Because this approach is computationally expensive, we are interested in its own parallelization, specifically the parallelization of the discrete-event simulator. We describe methods suitable for parallelized direct execution simulation of message-passing parallel programs, and report on the performance of such a system, LAPSE (Large Application Parallel Simulation Environment), we have built on the Intel Paragon. On all codes measured to date, LAPSE predicts performance well, typically within 10% relative error. Depending on the nature of the application code, we have observed low slowdowns (relative to natively executing code) and high relative speedups using up to 64 processors.

*Research supported by NASA contract number NAS1-19480, while the authors were in residence at the Institute for Computer Applications in Science & Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23681.

†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 .

1 Introduction

Performance prediction and/or analysis of parallel programs is currently an important area of research, especially as parallel computers are coming to dominate the high performance computing arena. Writers of parallel compilers would like to be able to predict performance as an aid towards generating efficient highly parallel code. Users of performance instrumentation and tuning tools are interested in predicting and observing parallel performance, but must deal with the fact that instrumentation code may perturb their measurements. Developers of new parallel algorithms are interested in predicting how well the performance of a new algorithm scales up with increasing problem size and machine architecture. General users may be interested in performance tuning their codes for large numbers of processors (which are only infrequently available) using fewer, more readily available resources. Designers of new communication networks are interested in evaluating their designs under realistic workloads.

The method of direct execution simulation[4, 5, 9, 11, 16] of application codes offers a solution to each of these problems. Under a direct execution simulation all application code is directly executed to obtain information about the application's execution behavior, but all references by the application code to the simulated virtual machine are trapped by the simulator. From the point of view of the application code, it is running on the virtual machine. Thus, when the application executes temporal calls such as "what is the wallclock time *now*", or, "is there a message of type T available *now*", the response depends on the state of the virtual machine simulator at the simulated time of the call's placement. From the point of view of the simulator, the application code is a driver, describing the activity to be simulated. A detailed direct execution simulator for parallel programs offers the potential for accurate prediction of parallel program performance on large, possibly as-yet-unbuilt systems. The approach does require a great deal of computation, but is a good candidate for parallelization. Execution of the application processes is a clear source of parallelism; one easily envisions a system where the discrete-event virtual machine simulator resides on one processor while a pool of other processors host the directly executing application processes. This solution will perform poorly though in situations where either the communication path to the simulator becomes a bottleneck, or the simulator execution itself is a bottleneck. It is important then to consider the problem of parallelizing the virtual machine simulator.

This paper considers the problem of parallelizing the virtual machine simulator for message-passing parallel programs. The methods we describe have been implemented in a tool named LAPSE (Large Application Parallel Simulation Environment), implemented on the Intel Paragon, for Paragon codes. While pieces of LAPSE are specific to the Paragon, the synchronization algorithm we describe in this paper is applicable to general message-passing systems. LAPSE accepts as input only a "makefile" describing how to build the application, the application source code, and a LAPSE initialization file describing the size and characteristics of the virtual machine and the mapping of the virtual machine onto the physical machine. From these LAPSE automatically builds, loads, and executes a direct execution simulation of the application code. LAPSE supports applications written in C, or Fortran, or a mixture of the two. We describe LAPSE's performance on four parallel applications: two linear system solvers (one indirect, one direct), a continuous time Markov chain simulator, and a graphics library driver. Measuring speedups (relative to simulations using only one processor) and slowdowns (relative to natively executing code) we observe a wide range of performance characteristics, depending in large part on the nature of the code being simulated. LAPSE has proven to accurately predict performance, typically within 10%. While we have simulated as many as 512 virtual processors using only 64 actual processors, the main limitation we've encountered is limited physical memory. This problem is specific to the Paragon, and should not be an issue on parallel architectures that better support virtual memory.

Several other projects use direct execution simulation of multiprocessor systems. Among these we find two pertinent characteristics, (i) the type of network being simulated, and (ii) whether the simulation is itself parallelized. Table 1 uses these attributes to categorize relevant existing work, and LAPSE.

Tool	communication	simulator
HASE [12]	message-passing network	parallel
LAPSE	message-passing network	parallel
MaxPar[3]	shared memory (no cacheing)	serial
Proteus[2]	cache-coherent shared memory	serial
RPPT[4]	message-passing network	serial
Simon[9]	message-passing network	serial
Tango[7]	cache-coherent shared memory	serial
WWT[24]	cache-coherent shared memory	parallel

Table 1: Direct Execution Simulation Tools.

Among most current simulators other than our own, simulation of cache-coherency protocols are an important concern. LAPSE is implemented on the Intel Paragon[13], which does not support virtual shared memory. Coherency protocols complicate the simulation problem considerably, but are a facet LAPSE need not deal with. However, existing work has identified context-switching overhead as a key performance consideration, and it is one that directly affects us. As much as an order of magnitude improvement has been observed when a direct-execution simulator uses its own light-weight thread constructs to accelerate context-switching (for small grain sizes). The thread packages available to us do not support the appearance of independent virtual address spaces necessary to our approach, nor are we able (in the context of a shared machine in a government lab) to modify the operating system kernel to support this ourselves; LAPSE processes are by necessity OSF-1 Unix threads, are subject to that operating system's mechanisms for scheduling, and are subject to its costs for context switching.

The Wisconsin Wind Tunnel (WWT) is to our knowledge the only working multiprocessor simulator that uses a multiprocessor (the CM-5) to execute the simulation. (HASE was not operational in parallel at the time [12] was published. The intent in HASE is to use a commercially available simulator based on the optimistic Time Warp synchronization protocol.) It is worthwhile to note the differences between LAPSE and the WWT. The first is a matter of purpose. LAPSE's primary goal is to support scalability and performance analysis of Paragon codes. The WWT is a tool for cache-coherency protocol researchers, being designed to simulate a different type of machine than its host. A second difference is a matter of lookahead, the ability of a conservative parallel simulation to predict its future behavior. In a cache-coherent system, a processor may interact with the communication network on any cache miss, or may have its cache affected by a write at any time by another processor. The lookahead is apparently poor. The WWT deals with this by keeping things in close synchrony. The WWT exploits an assumption that any communication between processors requires at least $B \approx 100$ number of cycles. Application object code is altered to cause WWT application processes to synchronize every B cycles. Any communication is deferred until the next barrier with the assurance that the barrier occurs before the communication can have affected its recipient. (This method of synchronization is a special case of the YAWNS [19, 22] protocol.) The WWT ignores any network contention by assuming that the latency of every message is fixed, and known. By contrast, Paragon processors are less tightly coupled. In a cache-coherent setting any memory reference might generate a network event; in a message-passing setting only an explicit call to a message-passing subroutine can influence network behavior. This fact allows a less rigid approach to synchronization. In particular, many large numerical programs alternate between a long computation phase where no communication occurs and a communication-intensive phase. Because of better lookahead possibilities, LAPSE can avoid synchronization during long periods of network idleness whereas the WWT cannot. The lookahead available in LAPSE comes from the observation that, in many applications, long portions of the execution path are independent of time. In such cases the application code can be executed well in advance of actually simulating the timing. Where the execution path is not independent of time, lookahead can still be obtained provided there is a lower bound on the operating system overhead required to send or receive a message. Finally, the WWT uses a customized operating system that cleverly exploits CM-5 idiosyncrasies to recognize misses in the simulated cache. LAPSE runs purely as an application.

Individual elements of LAPSE have been proposed before, e.g., parallelized direct execution, and support for different network simulators. Our contributions are to show how to effectively synchronize parallelized direct-execution simulations of distributed memory programs, to demonstrate the feasibility of our approach by actual implementation and testing on non-trivial codes, and to observe sometimes excellent performance. This combination of features makes LAPSE unique among its peers.

Section 2 gives an overview of the LAPSE system. Section 3 describes how LAPSE transforms a massively parallel code into a direct execution simulation and then Section 4 details our synchronization strategy. Section 5 describes our experiments and their results with respect to validation, slowdown, and speedups, while Section 6 presents our conclusions.

2 An Overview

A parallel program for a distributed memory machine is comprised of N application processes, distributed among $n \leq N$ processors. Most parallel programs are constructed so that $n = N$, an equivalence we presently assume. We assume the mapping is static. Application processes communicate through message passing using explicit calls to system library routines. For example, the **csend** call in the Intel **nx** library sends a message. The calling application passes arguments defining the message type (a user-defined integer), the message base address and length, the process id and processor id of the recipient. Control is returned to the application as soon as the memory area occupied by the message is available for reuse. **crecv** is called to receive a message; arguments are the message type, base address to place the message, and maximum message length. Control is returned to the application process once the message is received. **irecv** is an asynchronous version of **crecv**. If the receive routine is called before the anticipated message arrives, the incoming message will transfer directly from the network into the location specified by the user. Otherwise the message transfers from the network into a communications buffer; a subsequent receive call, **msgdone**, copies the message into the user's buffer.

Figure 1(a) illustrates how an application process views time. It runs for a period, then calls a system message-passing routine to send or receive a message. The message transaction is complete upon return of control to the application. The time the system spends handling messages is invisible to the application. An application process knows about execution durations, e.g., process 1 can measure or predict durations $a - 0$, $c - b$, and $e - d$ under the assumption that it is not interrupted; these are analogous to service times in a queueing network. If we can assume that such durations are independent of network activity (again assuming lack of interruptions, a facet we do deal with), these durations give us information we can exploit in a parallel simulation synchronization protocol. Interrupt durations and message-passing overheads are determined in part by the network state, as illustrated in Figure 1(b). In Figure 1(b), a message is sent from process 1 to process 2 starting at time a. At time a, control is passed to the operating system on processor 1. Because of the operating system overhead required to prepare the message for transmission over the network, control is not returned to application process 1 until time b. The message begins coming out of the network on processor 2 at time f, thereby interrupting process 2, which was in the middle of an execution block. At this point the operating system on processor 2 gains control to handle the interrupt. After the message has been completely received by the operating system, control is returned to application process 2 (at time g). When process 2 finally reaches the code (at time h) that explicitly receives the message (in this case copying it from a system buffer to user space) an additional overhead is incurred; this message receipt overhead is completed at time i at which point process 2 begins executing again. By contrast, process 1 reaches its receive statement (at time c) before the message from process 2 arrives. The arriving message moves directly into the user buffer. Application process 1 continues execution after the receive has been completed (at time d). The application processes are unaware of these timing details. It falls to the simulator to assign virtual times to event times a, b, c, and so on, as a function of the execution durations

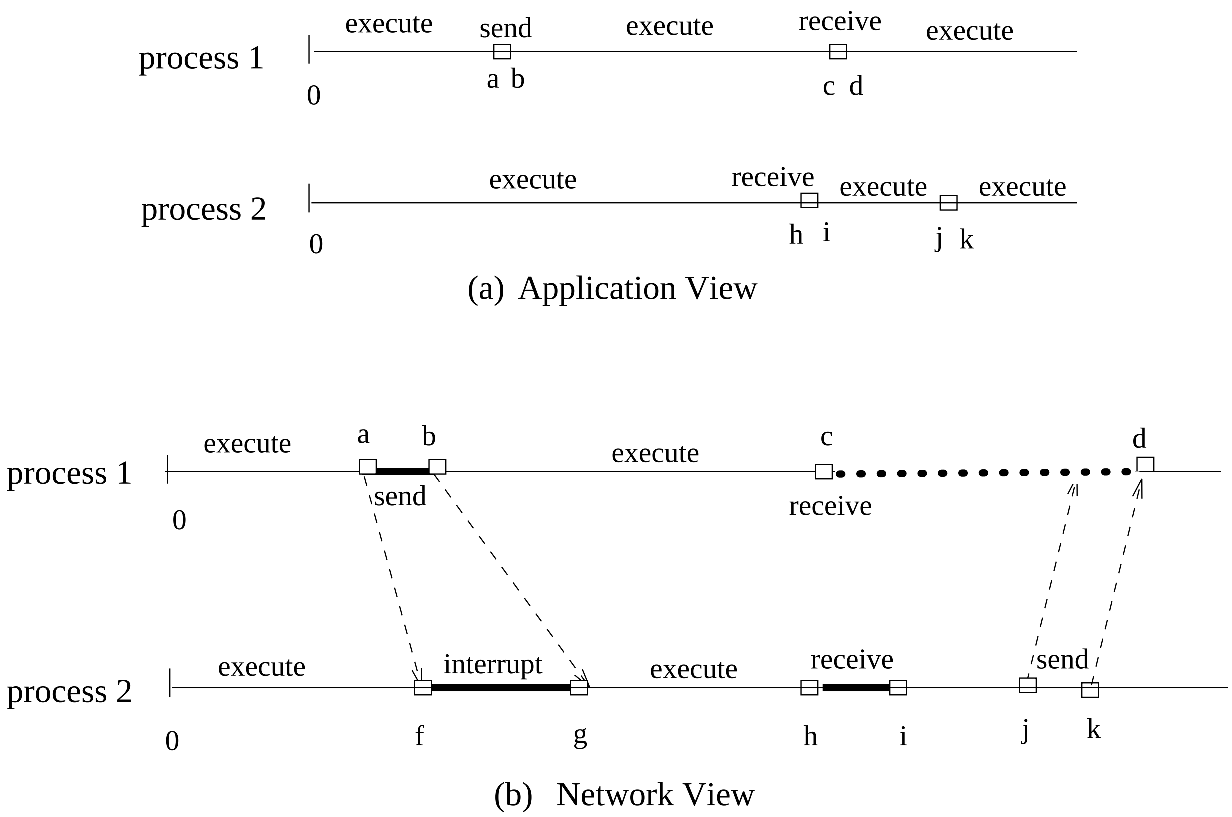

Figure 1: Application and Network views of parallel application timings.

reported by the application processes, an evaluation of message-passing delays, assumed operating system overheads for message-passing and interrupt handling, and a model of how message-passing activity affects the execution of application processes. Indeed, an application simulator in LAPSE maintains for each application process a data structure reflecting Figure 1(b), called a *time-line*, that records observed application events and assigns simulation times to them. A time-line is essentially a future-events list for an application process simulator, where the time-stamps on future-events are modified as the simulation progresses, depending on the simulation activity. The simulator is conceptually separated into an application simulator and network simulator, although the set of future application and networks events is essentially a single event list.

Figure 2 gives an overview of LAPSE's communication structure. An application submitted to LAPSE is recompiled with LAPSE macros that redirect application message-passing calls to corresponding LAPSE routines. The set of all such routines is known as the LAPSE interface. The interface code is linked to the application, becoming resident in the same address space. Application calls to interface routines typically trigger some interaction between application processes, through the interface routines. For instance, an application call to send a message is trapped by an interface routine which sends the message—to be received by an interface routine corresponding to a matching message receive in another application process. An interface routine also communicates with some

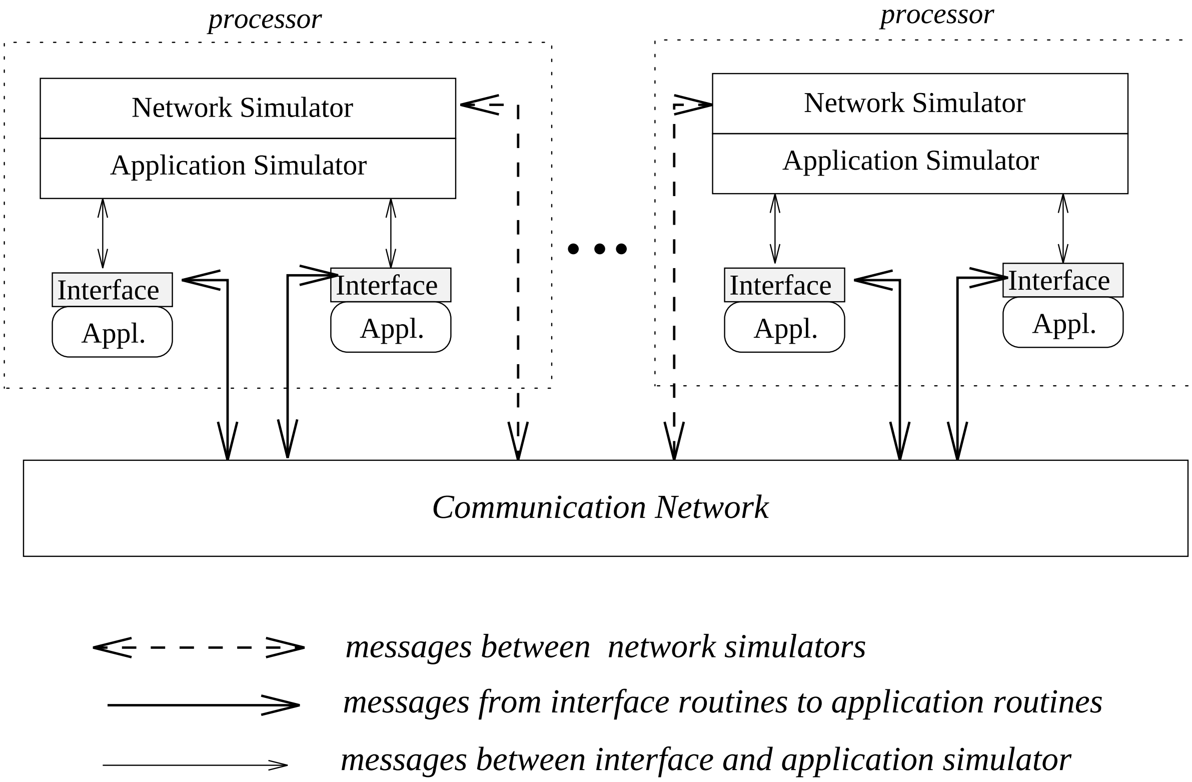

Figure 2: The LAPSE communication structure.

LAPSE application simulator, notifying it of the activity it has performed (or that the application wishes to perform). A LAPSE application simulator is responsible for receiving descriptions of application "events" from application interface routines, and for assigning simulation times to events on time-lines like those illustrated in Figure 1. Each application simulator interacts with a network simulator responsible for simulating activity in some portion of the virtual communication network. An application simulator and the network simulator it interacts with form a single process responsible for simulating a number of application processes that are typically (but not always) resident on the same physical processor. The collection of all application and network simulators cooperatively synchronize as a parallel discrete-event simulation.

The interface between an application simulator and its corresponding network simulator is simple, supporting the integration of different network simulators. The performance data in this paper is taken from a pure-delay simulator that ignores any contention. As will be seen, the no-contention assumption is not deleterious to LAPSE's ability to predict performance.

3 Application/LAPSE Interaction

In this section we briefly describe how an application code is transformed into a LAPSE simulation code.

LAPSE preprocessing is invoked by modifying an application's makefile (i.e. the input file for the Unix **make** command). All references to the compiler or linker are replaced by references to LAPSE scripts. The compilation script prepends to each source file a list of macro definitions that remap every message-passing routine call to a corresponding LAPSE routine. In this way all interactions by the application code with the virtual machine environment are trapped (excepting asynchronously triggered message handlers, whose use is discouraged on the Paragon). Next each source file is compiled to assembly code, which is instrumented with code that increments (at basic block boundaries) an accumulating count of the number of instructions executed so far. The instrumented code is then compiled. The linker script causes the additional linkage of LAPSE interface code that includes all routines to which message-passing calls have been remapped. The instrumented code and attached LAPSE interface code become one OSF-1 Unix process, which we call the application process. Each of these is represented by a time-line and attendant data structures as a virtual processor (VP) in the simulation.

Every call to an interface routine is temporally *sensitive*, or *insensitive*. An insensitive call is one whose result or effect does not depend on the state of the virtual machine at the point of the call. Put another way, the application's execution path does not depend on insensitive calls, but it may depend on the results of a sensitive call. Most calls are temporally insensitive, including those corresponding to the transmission or receipt of messages (e.g., **csend**, **crecv**, **isend**, **irecv** and their extended versions, on the Paragon), and those that cause the calling processor to block until some condition is satisfied (**msgwait**, **gsync**, all global reductions). Sensitive calls include those to real-time clocks (**dclock**), asynchronous probes for the existence of particular messages (**iprobe**, **iprobex**) and queries after the status of anticipated messages (**msgdone**). To appreciate the importance of this distinction, compare the two code fragments below.

Fragment 1	Fragment 2
crecv(MSGTYPE, MsgAdrs, MsgSize);	MsgCode = **irecv**(MSGTYPE,MsgAdrs, MsgSize);
ProcessMsg(MsgAdrs);	idx=1;
csend(MSGTYPE+1,MsgAdrs,1,10,0);	while(**msgdone**(MsgCode)==0) BusyWork(idx++);
	csend(MSGTYPE+1,MsgAdrs,1,10,0);

Both fragments wait for a message of type MSGTYPE to appear, process it, and then send a message themselves. Fragment 1 simply blocks and waits for the message. It then processes the message and then sends a message (of 1 byte to process 0 on node 10). Fragment 2 polls the Boolean valued function **msgdone** to determine if the message is present. If not, a routine BusyWork is called with a parameter that changes with every call. The number of times BusyWork is called, and the parameters passed to it depend on the responses from **msgdone**. While none of the calls in Fragment 1 are temporally sensitive, Fragment 2's call to **msgdone** *is* temporally sensitive. In LAPSE, the answer to the **msgdone** query will come from the simulator, based on whether the simulator observes the message of interest. After the simulated message arrives (in the simulator), the simulator answers the **msgdone** affirmatively by tell the LAPSE interface that the message is available. The interface then waits until the corresponding application message actually does arrive (from another application process). In this way, the application calls the **msgdone** routine exactly as many times as it would on the simulated virtual machine. In this case there is close linkage between the simulator and the application execution. Fragment 1 needs no such interaction with the simulator for its execution, and therefore Fragment 1 application code can be run well in advance of when the simulator eventually simulates the timing of events corresponding to Fragment 1.

At runtime, LAPSE loads simulator and application processes onto the physical machine as specified by an input file. Typically, each physical processor is assigned one simulator process and a number of application processes, although arrangements separating simulation and application processes are also possible. In either case, each application process is notified of the identity of a single simulator process with which it will interact. Likewise, each simulator process is initialized with the identity of all application processes with which it will interact. The application processes are then permitted to execute.

Whenever a LAPSE interface routine is called it recovers the number of application instructions that have executed since the last call to a LAPSE routine. This count is inflated by an input parameter (effective clock ticks per instruction) obtained from the input file to produce an estimate of the length of time the VP would have run without interruption on the virtual machine, between the last two interactions with the machine. A mechanism for determining that input parameter is described in Section 5. Next, if the call is temporally insensitive its parameters are transformed from virtual machine coordinates into physical machine coordinates, and the requested operation is performed. In the case of message sends and receives we take advantage of the address space co-residence of the interface routine and application, and leave memory address parameters unaltered. Temporally sensitive calls do not involve actual execution of the request by the interface. In either case, subject to flow-control considerations the interface routine next sends a message to its assigned simulator, reporting the measured execution burst, and the requested operation. At this point the interface routine for a temporally insensitive call completes the operation (e.g., blocks on a **crecv** call until the anticipated message arrives), and then returns control to the calling application process. A temporally sensitive call waits for a response from the simulator, and upon receiving it returns the response to the application.

The Paragon library **nx** includes a rich collection of global reduction operations. LAPSE maps these to LAPSE routines that implement them using more primitive send/receive calls, and instruments such routines as though they were part of the application. To account for the cost of global synchronization, for instance, the original application call is redirected to a LAPSE routine that implements a barrier in the standard tree-fashion, in virtual machine coordinates. Since that code is itself remapped and instrumented exactly like application code, the cost of the synchronization is obtained by executing a LAPSE version of what the Paragon library function does. LAPSE supports almost all **nx** calls consistent with the Single Program Multiple Data (SPMD) paradigm. The principle omissions are calls for asynchronous message handling (**hrecv**), message cancellation calls, and message calls involving type masks. These calls are rarely used in our computing environment.

To obtain accurate timing of the application, the operating system overhead for sending and receiving messages needs to be properly accounted for. LAPSE currently estimates such operating system overheads. For example, the overhead (in clock ticks) to execute a **csend** can be modeled as $a + b \times L$ where a is a startup cost, L is the message length and b is the cost per byte. Estimates of a and b can be obtained by measurements of the operating system. In principle, it would also be possible to instrument the operating system itself with the instruction counting code and to run this instrumented code as part of the simulator. This would eliminate the need for using estimated path lengths, however, it requires access to the operating system source code as well as authorization to run the modified kernel.

4 Synchronization

A strictly serial simulation executes events in monotone non-decreasing order of event times. We view a parallel simulator as a collection of individual discrete-event simulators, each run on a separate processor, each having the ability at any time of scheduling an event on a *different* simulator than itself. We desire that events on each processor also be executed in monotone increasing order, or at least compute the same state as though such monotonicity were achieved. If the earliest event on a processor has time-stamp t, before the processor executes that event it must either be certain that no other simulator can still schedule on it an event with smaller time-stamp $s < t$, or it must be prepared to rollback and begin recomputing at time s if such an event is later scheduled.

There is a rich literature of solutions to this synchronization problem. Good introductory surveys are found in [10, 25], and a survey of the state-of-the art is found in [20]. *Conservative* synchronization protocols prohibit a simulator from executing an event if there is any possibility of an earlier event being scheduled there later. *Optimistic* protocols allow out-of-sequence event processing. Each style has its strengths and weaknesses, and each has been shown to perform well on some applications, and poorly on others. Regardless of the method, good performance is always dependent on a model's tendency towards relatively infrequent scheduling of events by one processor onto another. Conservative methods frequently work well if that slackness can be identified ahead of time by computation, i.e., if each simulator is able to continuously compute and distribute lower bounds on the future times at which it may schedule events on other processors. The ability to make such predictions (called lookahead) is very model dependent. Optimistic methods basically work by assuming the slackness exists, and correcting errors made when the assumption proves to be false. Optimistic methods have the potential for being more general, but carry with them the overheads necessary for recovering from errors. Optimistic methods are also very much more difficult to implement correctly.

LAPSE uses a new conservative synchronization protocol, Whoa, (Window-based halting on appointments), tailored to the characteristics of the simulation problem. Suppose simulation time t was previously chosen as a global synchronization point (initially $t = 0$). Once all simulators have simulated up to t, they cooperatively establish a simulation time $w(t) > t$, and after doing so simulate all events with time-stamps in $[t, w(t))$. Within the window simulators may still synchronize with each other, but do so in a pairwise fashion; global synchronization occurs only at the boundaries of the windows. Once every simulator has reached time $t' = w(t)$, another time $w(t') > t'$ is chosen, and the window $[t', w(t'))$ is simulated. This process continues until the simulation terminates. Given lower window edge t, the value $w(t)$ is computed to ensure that all message send events whose time-stamps will eventually fall in $[t, w(t))$ exist already on their applications' time-lines.

Synchronization within a window is governed by the dynamic computation and distribution of lower bounds on future times at which one simulator may affect another. The only interaction between simulators is through *remote send* events, i.e., a message send call whose destination is a VP managed by a simulator other than that of its source. These lower bounds are called *appointments* [21]. There is an appointment associated with every remote send event, on every time-line; it is initially computed when the event is received, and it is updated as the simulation progresses. The appointment reflects the best known lower bound on when the associated message reaches the network hardware. The application simulator passes these appointments to its corresponding network simulator, which transforms it into an appointment between itself and the network simulator for the message's destination. Depending on the network being simulated, appointments with other network simulators may also be in order. A network simulator computes a *halting* time as the minimum incoming appointment time. Neither application nor network simulator ever processes an event whose time-stamp is larger than the simulator's current halting time. As the simulation progresses, a simulator either sends the fore-warned messages, or increases the lower-bound on network entry times. As these modifications occur the network appointment times either disappear or increase, and each simulator's halting time must increase until it is at least as large as $w(t)$. As the halting time increases, the application and network simulators are free to execute events with smaller time-stamps. The protocol will not deadlock—if $h' < w(t)$ is the least halting time among all simulators, then the simulator whose appointment defines h' must have a simulation clock value strictly less than h', and must have an event with time-stamp strictly less than h', which it can safely execute.

We now elaborate on Whoa, in two steps. First we discuss how appointments are computed, how they are used, and when they are updated. Secondly we discuss construction of windows.

4.1 Appointments

LAPSE exploits the tendency for message-passing codes' execution paths to be largely insensitive to timing. The key observation is that one can execute the application processes largely as on-line trace generators, and maintain on the time-lines a potentially long list of future-events. Whoa builds appointments and windows around these lists. In this subsection we describe two types of appointments, application appointments from application simulators to network simulators, and network appointments between network simulators.

Consider Figure 3 illustrating a VP's time-line and a situation where simulation time (for the simulator as a whole) has advanced to t. The time-line records execution bursts with lengths A,B,C,

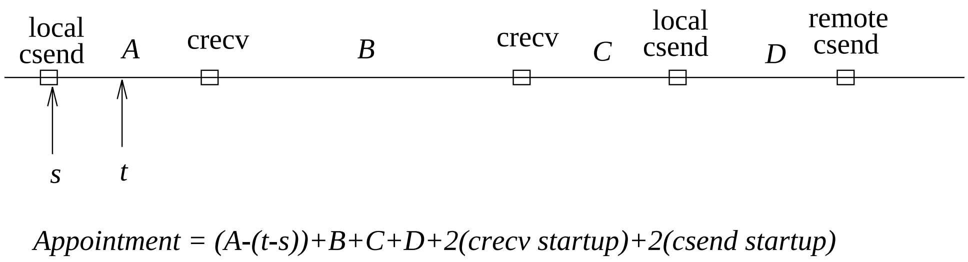

Figure 3: Application appointment calculation on a VP's time-line.

D, and application events; we wish to compute a lower bound on the network entry time of the remote message. An event's application appointment is a lower bound on the instant when it will be scheduled on the network simulator to model network entry. It is constructed as the sum of the residual time in the current execution or startup block, plus all execution burst lengths, plus lower bounds on startup costs of intervening application events, plus the lower bound on the startup cost of the event itself. As application events are processed the appointment may increase.

An application appointment is made for a remote send shortly after that event is reported to the application simulator. Whoa strives to keep appointments current as the simulator state evolves. Appointments change specifically in two cases. First, it may happen that the duration of an application event startup is larger than the lower bound assumed. This usually arises to include some cost—like that of copying a message—that is not always suffered and hence is not included in the lower bound. When the event is executed and the additional cost is included, appointments for all remote send events on the affected VP are increased by the amount of the additional cost. The appointments may change in the presence of blocking. For example, if simulation of an **crecv** in Figure 3 fails to find the anticipated message, the VP becomes suspended. As simulation time advances without the message appearing, the appointments computed by the suspended VP must increase. Likewise, a VP that is interrupted by the arrival of a message becomes suspended until the message is completely received, and its appointments must be updated during the period of its suspension. At the point a VP becomes suspended, say at t, we find the VP's minimum outgoing appointment time t_a, compute the difference $d = t_a - t$, and then periodically schedule a "VP appointments update" event every d units of time, until the VP becomes unsuspended. Execution of this event advances all of the VP's appointments by d units. The update event is removed from the event list when the VP is released from suspension, at which point updated appointments are passed.

Individual simulators synchronize using network appointments. A network appointment at time t' from simulator i to j, for message m, is a promise that the event reporting m's arrival at the subnetwork managed by j will not be scheduled at j before time t'. Given m's application appointment, simulator i's network simulator constructs the network appointment by adding the network latency time (which depends on the distance the message travels). LAPSE currently sends that network appointment every time the network simulator receives a new or updated application appointment (this is overkill, we are developing a version that very much reduces the communication volume associated with appointments). Because LAPSE presently uses a contention free network model, no further updating of network appointments is needed. However, more sophisticated updating of network appointments will be required for network models that capture contention.

The transmission of a simulated message from one network simulator to another removes its associated appointment. Upon receipt, the target VP is taken to be suspended for as long as it takes the message to completely arrive (which depends on the number of packets into which it has been decomposed). The target network simulator notes the time at which the message will be completely received, data needed for the window computation to be described.

Every network simulator maintains a single "halting time" for both the application and network simulators; at any instant the halting time is the time of the current minimum incoming network appointment. Any event with a time-stamp less than the current halting time may be safely executed. The halting time increases as appointments change, and messages are received. Eventually the halting time increases past the upper edge of the current window, all remaining events in the window are executed, and the simulator engages in computing the upper edge of the next window.

4.2 Window Construction

Suppose that time t is the upper edge of a window. A simulator engages in the window construction protocol once its halting time is t or larger, and it has no remaining events with time-stamps less than t. The upper edge $w(t)$ of the next window is computed to ensure that it is a lower bound on the time-stamp on the next unknown (i.e., not on the time-line) remote send event that any simulator may execute. $w(t)$ is thus a measure of how far into the future, globally, the application processes have advanced ahead of the simulation processes. We desire that many events be found on each VP's time-line between times t and $w(t)$, to better amortize the cost of computing $w(t)$.

Towards this end, a simulator's first action is wait until every one of its VP's time-lines has as many events as is apparently possible (it will always wait until at least one event is present). For reasons involving the Paragon's management of communication buffers, a user specified flow-control parameter F is involved in the decision. The general rule is the simulator will wait for more events until either the application processes reports that it is blocked, or at least F application events exist on the time-line already. The application process reports its current blocked/unblocked state with every event it reports; the state is also reported separately when it changes from blocked to unblocked. There are three ways an application process may become blocked. First, it will be blocked if the last call it made to an interface routine was temporally sensitive. To become unblocked it must wait until the the simulator executes the temporally sensitive call and responds to it. Secondly, the application process may be blocked at a temporally insensitive call, such as **crecv** or **msgwait** which both wait until an application message is received from another application process. At such time as the application process becomes runnable, the interface routine notifies the simulator of the change. Finally, the application may become temporarily blocked for purposes of message flow-control.

Once all of its time-lines are filled, a simulator enters a global vector (component-wise) min-reduction, the results of which permit each simulator to compute $w(t)$. Each simulator offers a 3-tuple to the reduction. The first element is L, a lower bound on the last time-line event of a unsuspended VP; the second element S is a lower bound on the last time-line event of a suspended VP, computed under the assumption that it will be released from suspension immediately. The third component R is a lower bound on the time at which a suspended VP will be released. R is itself a minimum of three values—the minimum completion time of a message in the midst of being received, the minimum incoming network appointment time for the simulator, and a lower bound on the minimum time at which an unsuspended VP next sends a message to any other VP (this includes local sends). Given the global minima $L_{\min}$, $S_{\min}$, and $R_{\min}$, each simulator computes

$$w(t) = B + \min\{L_{\min}, R_{\min} + S_{\min}\},$$

where B is a lower bound on the startup cost of a send event.

By construction we are assured that any remote send event whose time-stamp ultimately falls in $[t, w(t))$ is resident on a time-line at the point the window is constructed. This property is key to Whoa, for it ensures that an appointment for every remote send event in the window has already been established, and that no synchronization other than appointment management is needed. It allows us to use the appointments in $[t, w(t))$ to define a dynamically evolving pairwise synchronization schedule that will not deadlock.

The effect of temporally sensitive calls on window construction deserves remark. Presence of a temporally sensitive call on a time-line indicates that the application process is blocked—the call will always be the last one on its VP's time-line. This means that the values L and R offered by its simulator to the reduction will be small, so that the difference between $w(t)$ and t will also be small. In extreme cases where most calls are temporally sensitive and tend not to occur simultaneously (between VPs), the net effect is to serialize the simulation of communication events.

5 Experiments

In this section, we report on experiments using LAPSE. After describing a set of four scientific applications that we used for experimentation, we first address the issue of validation, i.e., quantifying the accuracy of LAPSE timing predictions. We next quantify overheads; these are captured by the application *slowdown*, which is defined to be the time it takes LAPSE to execute an application with N virtual processors on N physical processors divided by the time it takes the application, running natively, to execute on N physical processors. We then characterize the simulation *relative speedup*, which is defined to be the time it takes LAPSE to execute an application with N virtual processors on n physical processors divided by the time it takes LAPSE to execute the same application on one physical processor. This relative speedup is representative of *absolute speedup*, the execution time of a hypothetical optimized serial simulator divided by that of LAPSE on n processors. The justification for this belief is that LAPSE avoids most unnecessary overheads when running serially. Essentially the only overhead that LAPSE does not avoid is maintenance of the appointment data structures. However, this overhead is small compared to the actual execution of the application (with its attendant instruction counting) which would have to be done on any serial simulator. In fact for one of our applications (SOR with the high computation to communication ratio), LAPSE runs a four VP problem on four processors only 1.8 times slower than the native application on four processors. We will see that the slowdowns and speedups are strongly affected by the amount of lookahead present in the application, as well as by the application's computation to communication ratio. In this context, application lookahead refers to how far in advance of the timing simulator the application is able (or allowed) to run.

5.1 Applications

We experimented with four parallel applications that are representative of a variety of scientific and engineering workloads (although no claim is made or intended that these comprehensively span all such possible workloads). The applications arise in physics, computational fluid dynamics, performance modeling of computer and communications systems, and graphics.

The first application, **SOR**, solves Poisson's boundary value equation in two dimensions (see Chapter 17 in [23]). This is a partial differential equation (PDE) in two dimensions with a given set of values on a boundary (in our case a rectangle). The PDE is discretized, resulting in a large, sparse system of K^2 linear equations where K^2 is the number of discretized grid points. The equations are solved using simultaneous over relaxation (SOR) with odd-even ordering; at each iteration the solution at (an interior) grid point (i, j), $x(i, j)$, requires $x(i+1, j)$, $x(i-1, j)$, $x(i, j-1)$ and $x(i, j+1)$. If there are N processors arranged in a square grid, each processor is assigned a sub-grid of size $G \times G$ where $G = K/\sqrt{N}$. The computation thus results in a NEWS (North East West South) communications pattern. The number of instructions executed between the NEWS exchange is of order G^2 and the size of each pairwise exchange is of order G bytes. Thus by varying G, we can adjust the computation to communications ratio of the application; we call this ratio "very low" when $G = 25$, "low" when $G = 50$ and "high" when $G = 250$. This application results in a highly regular communications pattern. The message passing calls are all synchronous, **csend** or **crecv**. Thus the application execution path is timing independent, and thereby has good application lookahead.

The second application, **BPS**, is a domain decomposition solver for finite difference or finite element discretizations of two-dimensional elliptic partial differential equations [14]; this type of problem arises frequently in computational fluid dynamics. The algorithm is of Bramble-Pasciak-Schatz substructuring type consisting of a conjugate gradient method preconditioned by an approximation to the inverse of the matrix operator obtained from discretizing the PDE [1]. The conjugate gradient method applied to a sparse matrix is highly parallelizable, requiring at each iteration only nearest neighbor communication in the formation of matrix-vector products, and a global reduction operation in the formation of inner products. The domain of the PDE is partitioned into nonoverlapping subdomains. The subdomains are separated by a "wire basket" consisting of edges and vertices. Once values are specified for these edges and vertices, independent problems may be solved for each subdomain interior. This is done in the final iteration of the algorithm. The objective of the earlier iterations is to arrive at sufficiently accurate values for the unknowns on the wire-basket itself. This process requires independent solutions on each edge at each iteration, as well as a small global solution of a problem defined on the vertices only. The specific test case is Poisson's equation on a unit square with Dirichlet boundary conditions, and specific advantage is taken of the existence of fast Poisson solvers for the subdomain interiors. The code uses synchronous calls such as **csend**, **cprobe**, **crecv**, and global reductions and thus exhibits good application lookahead. The code runs on numbers of processors that are powers of 4 and two levels of computation to communication ratio were obtained by varying the size of the $G \times G$ subdomain assigned to each processor; the "high" ratio has $G = 256$ while the "low" ratio has $G = 64$. This code, which is written in a combination of C and Fortran, was provided to us by David Keyes and Ion Stoica of ICASE and Old Dominion University.

The third application, **APUCS**, is a continuous time Markov Chain discrete event simulator using the parallel algorithm described in [18]. The specific system simulated is the queueing network model of a large distributed computing system described in [18]. This application has highly irregular communications patterns involving any-to-any pairwise messages. This application also uses synchronous message passing calls (**csend**, **irecv** and **msgwaits**) as well as global reductions. The parameter settings were those of "Set I" of [18] with the "high" computation to communication ratio achieved by setting $\mu_f = 1$ and the "low" computation to communication ratio achieved by setting $\mu_f = 64$.

The fourth application, **PGL**, is a parallel graphics library written to support visualization of scientific data [6]. In the sample driver program for this library, each processor generates some number of randomly spaced and colored triangles of a certain size. The processors then rotate and shade the vertices of their triangles. Display scan lines are distributed in an interleaved fashion over the processors. The endpoints of the scan lines of each triangle, called a span, are then sent to the processor responsible for that scan line. For each scan line, the pixels are colored by interpolating between the endpoints of the spans and a Z-buffer algorithm is used for hidden surface removal. This process is repeated for each of F frames. This code uses any-to-any pairwise communications and is highly asynchronous, frequently using **irecvs** and the temporally sensitive **msgdone** call. Because of this the simulator has very little application lookahead. This code, which consists of approximately 14,000 lines of C, was written and provided to us by Thomas Crockett of ICASE. The "high" and "low"computation to communication ratios were achieved by assigning 1000 and 500 triangles per processor, respectively. However, for this application it is harder to adjust the computation to communication ratio, and the designations "high" and "low" are used mainly to distinguish between two different workloads.

5.2 Validation

In this section we describe the process by which we validated LAPSE timing predictions, and describe the results of our validation experiments. In order for LAPSE to make reasonable predictions, we must know the overhead of system message passing calls (e.g. **csend**) and interconnection network transit times, as well as the time the application spends between each message passing call.

We obtained estimates of the system overheads by measuring the system using test programs that were written specifically to determine such overheads. Overheads for calls that depend on the message length, e.g., **csend** were modeled as $a + b \times L$ where L is the message length and a and b are constants estimated by regression on the measurement data. Interconnection network transit times can be similarly measured. We have found that, except for very long messages, the software overheads for sending and receiving messages are far greater than the hardware transit times. Thus predictions using our simple network simulator, which does not model link contention, can be expected to be (and are) accurate.

To obtain the application time between message passing calls, we proceeded as follows. First, we ran the application natively (i.e., without LAPSE) on a small number of nodes (typically 2×2) with a certain data size per node (e.g., G in SOR or number of triangles per node for PGL). The Unix **time** command gives us an estimate of the amount of user time U consumed by the application. We next ran the same application (with the same data size) under LAPSE, which reports the total number of user instructions I. From this, we compute a "conversion factor" $c = U/I$ which is the average time per (user) instruction. This conversion factor incorporates, in an average case sense, the instruction mix, cache hit ratio, etc. of the application. The conversion factor is then used in subsequent runs of the application under LAPSE for timing predictions. For example, if there are J application instructions executed between two message passing calls, LAPSE computes the time between these calls as $J \times c$. For a given application and data size per processor, we use the same conversion factor c measured on the small number of nodes to predict timings on a large number of nodes. For example, in SOR with a fixed value of G, the number of instructions, the instruction mix, and the data access patterns executed between message passing calls are approximately the same on a 2×2 grid of processors as on an 8×8 grid of processors. Thus, for a fixed G, we expect the conversion factor to be approximately independent of the number of processors. However, for a different value of G, the conversion factor may be different and separate conversion factors were computed for each G (or computation to communication ratios for the other applications). In cases for which the application execution path is timing dependent, several iterations of computing conversion factors may be required; in the results presented below at most two iterations were used. The application was then run both natively and under LAPSE for a variety of numbers of nodes, and the timings compared.

Table 2 presents the percentage differences between LAPSE predictions of execution times and actual native execution times. As can be seen in the table, the maximum error is 6%. We have observed larger prediction errors, but these have always occurred for very short runs in which initialization effects may be present. For example, the PGL-High error on 64 nodes is -12% when generating 10 frames; the $+3\%$ error reported in Table 2 is for 20 frames. These runs span a range of application efficiencies. For example, LAPSE estimates that SOR on 16 processors spends 47% of its time executing user instructions when $G = 25$ (as opposed to executing system instructions or waiting for messages), while it spends 90% of its time executing user instructions when $G = 250$.

These results show that LAPSE can provide accurate timing estimates for a range of scientific and engineering applications.

5.3 Slowdowns

We next investigated the amount of overhead involved in running LAPSE. There are several types of overheads. First, there is the overhead involved in counting application instructions. Second, there is the overhead in actually doing the timing simulation (in parallel). Third, there is the operating system overhead that arises from managing multiple processes per node; an SPMD application running natively has only one process per node. All of these overheads can be captured in a single measurement called the slowdown, which is defined to be the time it takes LAPSE to execute the application on N nodes divided by the time it takes to execute the application natively on N nodes.

Our measurements have indicated that the instruction counting overhead generally ranges between 30% to 80%. These overheads were obtained by direct comparison of the time to natively execute the application to that when the application is augmented with instruction counting, but not timing simulation.

It is harder to separate simulation overhead from operating system overhead, nor shall we attempt to do so here. However, some measurements are presented in [8] indicating that OSF-1 on the Paragon has process management overheads that increase superlinearly as the number of processes per node increase. We do note that, parallel simulation overheads aside, LAPSE must send at least twice as many messages as the natively executing application; each message sent in the application results in both an application to application message as well as an application to simulator message (although these may be between processes on the same node). We now consider the overall slowdown of LAPSE. We begin by demonstrating the effect that application lookahead has on simulation speed. LAPSE attempts to run the application well in advance of the simulator, provided the application is able to do so. As explained earlier, this results in larger windows. User-supplied flow-control parameter F determines how the maximum number of message passing events the application is permitted to run in advance of the simulator. By running LAPSE on an application that has good lookahead (i.e., one with only synchronous sends and receives), we can study the effect of lookahead in a controlled manner by varying the parameter F. Table 3 presents the results of such an experiment for APUCS running on 8 processors with one VP/processor. This table reports the slowdowns as a function of F, as well as the average number of application events executed per window per VP, A. (An application event is defined to be a call to the LAPSE interface). For the low computation to communication ratio the slowdown decreases from 30.4 down to 9.3 as F increases from 2 to 16. At the same time A increases from only 0.3 to 8.0. For the high ratio the slowdown decreases from 12.0 to 3.7 and A increases from 0.3 to 4.6 as F increases from 2 to 16. This demonstrates that slowdown is strongly dependent upon application lookahead.

Table 3 also illustrates an effect of the flow-control algorithm used in LAPSE. For the low ratio, as F increases past 16 the slowdown increases. A contributing factor to this behavior is as follows. Each remote send event has an associated appointment time. As simulation time advances, these appointments may be updated, resulting in additional communication. Thus if there are a large number of events on the time-line, the appointments overhead increases. Increasing F increases the maximum allowable number of events on a time-line, and also tends to increase A, the average number of application events per window. However, there is not always an increase in slowdown for large F; see APUCS-High. In this case, A levels off as F increases. (The reason for this leveling off is not entirely clear; there are a number of factors interacting in a complex manner. These factors include operating system process scheduling, algorithms, the computation to communication ratios of the application and the simulator, the window construction algorithm, etc.)

We next consider slowdowns on the set of four applications described earlier. We compared LAPSE to native application time for the codes running on 4, 16, 32, and 64 processors. For the LAPSE runs, there are two processes per node; one application process and one simulator process. F was set at moderate values so as to obtain good lookahead whenever the application so permitted. The results of these experiments are reported in Table 4. Observe first that the slowdown increases as the number of processors increases. This is an effect of using a global window. The cost of computing a new window increases (logarithmically) as the number of processors increases, but more importantly we have observed that the average size of the window decreases as the the number of VPs increases. This occurs simply because $w(t)$ is computed as a function of minimum values taken over all VPs, and increasing the number of VPs increases the likelihood of low values submitted to the reduction. Observe next that, for a given application and number of processors, the slowdown typically increases as the application's computation to communication ratio decreases. With a high ratio, most of the time is spent executing application code and thus the simulation overheads are less important.

The slowdowns for SOR, BPS and APUCS are quite modest (between 1.8 and 28.0) and are considerably at the low end of slowdowns reported by other execution-driven simulators [5, 24]. Recall that these codes all have good application lookahead since they do not make much use of temporally sensitive message passing calls.

However, for PGL the slowdowns are significantly higher. As described earlier, this application has little lookahead since by executing many **msgdone** calls it continually forces synchronization between the application and simulation processes. In LAPSE, we deal with this type of temporal question by blocking the application process until the simulator has advanced simulation time up to the time at which the **msgdone** question was asked. This severely limits the size of the windows constructed by the windowing algorithm, and thereby slows simulation speed. Thus in PGL, application processes are frequently blocked and there are few application events per simulation window. With only one simulation process per node, the window construction algorithm becomes expensive relative to the amount of simulation work done in the window. These factors can be counteracted (to some extent) by placing multiple application processes per node, simulating multiple VPs per node, and separating the application processing nodes from the simulation nodes. For example, consider the PGL-high slowdown of 139 on 64 nodes. By configuring LAPSE to run on 48 nodes with 32 nodes for application processing and a different set of 16 nodes for simulation, the slowdown decreases to 117. Similar reductions in slowdown are obtained for PGL-low.

We next measure (relative) speedups by running LAPSE on the Paragon using N virtual processors on n physical processors where $N \geq n$. We again ran the applications for a variety of combinations of N and n. For these applications, operating system and physical memory constraints typically limit the level of multiprogramming to at most 8 VPs/processor; in the case of BPS-High at most 4 VPs/node could be handled. This effectively limits the size of N, especially for LAPSE running on one processor.

We first consider the case of simulating a small number of VPs on a small number of nodes; specifically simulating 8 VPs on from 1 to 8 nodes. (For BPS the number of nodes must be a power of 4 so in this case we simulate 4 VPs on from 1 to 4 nodes.) Table 5 shows the relative speedups for these experiments. Here relative speedup is defined to be the LAPSE time on one node divided by the LAPSE time on n nodes. For the applications with good lookahead, SOR, BPS, and APUCS, the speedups increase monotonically. For these applications, relative speedups on 4 nodes are between 2.4 and 3.7 (relative efficiencies between 0.60 and 0.92), while on 8 nodes the relative speedups range between 3.5 and 4.7 (relative efficiencies between 0.43 and 0.60). For PGL, the maximum relative speedup is 1.7, and most of that is obtained when increasing from 1 to 2 nodes.

We next consider simulating 64 VPs on from 8 to 64 nodes. Table 6 shows the relative speedups for these experiments. Here relative speedup is defined to be the LAPSE time on 8 nodes, with 8 VPs per node, divided by the LAPSE time on n nodes. (For BPS-High the maximum number of VPs per node was 4, so those speedups are stated relative to LAPSE time on 16 nodes with 4 VPs per node.) Again, the speedups are higher for those applications with good lookahead; LAPSE runs between 3.9 to 7.0 times faster on 64 nodes than on 16 nodes for SOR, BPS, and APUCS. The maximum PGL speedup is 2.5.

We were also able to run a 512 VP case of SOR-Very Low on 64 nodes. Since we did not have access to 512 nodes, an actual slowdown could not be computed. However, LAPSE ran only 100 times slower than the time LAPSE predicted it would take the application to run on 512 processors. (Remember that each LAPSE processor has 8 VPs and hence at least 8 times as much work to do as in the hypothetical larger system.)

Application	Comp./Comm. Ratio	Number of Processors 4	16	32	64
SOR	Very Low	-4%	+3%	+2%	+1%
SOR	Low	0%	+4%	+4%	+4%
SOR	High	+1%	+1%	+1%	+2%
BPS	Low	-6%	+1%	-	-4%
BPS	High	-1%	-2%	-	-3%
APUCS	Low	-1%	-2%	-3%	-1%
APUCS	High	-2%	-2%	-1%	0%
PGL	Low	+1%	-2%	+1%	+5%
PGL	High	-1%	-2%	-1%	+3%

Table 2: LAPSE validations: percentage errors in predicted execution times.

Comp./Comm. Ratio	Maximum Application Lookahead (F)	Slowdown	Avg. Application Events per Window per VP
Low	2	30.4	0.3
	4	14.2	1.1
	8	10.2	3.3
	16	9.3	8.0
	24	9.9	12.8
	32	10.9	17.5
	40	12.1	21.6
High	2	12.0	0.3
	4	6.1	1.3
	8	4.3	2.8
	16	3.7	4.6
	24	3.5	5.5
	32	3.5	6.0
	40	3.5	6.2

Table 3: LAPSE slowdowns on APUCS as a function of application lookahead. LAPSE execution time divided by native execution time using 8 processors.

Application	Comp./Comm. Ratio	Number of Processors 4	16	32	64
SOR	Very Low	17.5	28.4	24.0	28.0
SOR	Low	6.5	11.5	13.2	16.7
SOR	High	1.8	4.0	4.1	6.1
BPS	Low	3.0	7.5	-	11.9
BPS	High	1.9	2.4	-	3.0
APUCS	Low	7.5	11.1	13.5	18.8
APUCS	High	3.0	5.9	7.6	12.9
PGL	Low	15.1	61.6	99.2	148
PGL	High	17.4	70.2	116	139

Table 4: LAPSE slowdowns: LAPSE execution time divided by native execution time using the same number of processors.

Application	Comp./Comm. Ratio	Number of Processors 1	2	4	8
SOR	Very Low	1.0	1.8	3.1	4.7
SOR	Low	1.0	1.8	3.0	4.5
SOR	High	1.0	2.0	3.4	4.6
BPS	Low	1.0	1.7	2.4	-
BPS	High	1.0	2.0	3.7	-
APUCS	Low	1.0	1.7	2.5	3.8
APUCS	High	1.0	1.6	2.4	3.5
PGL	Low	1.0	1.5	1.6	1.7
PGL	High	1.0	1.4	1.4	1.6

Table 5: LAPSE speedups on an 8 VP problem (on a 4 VP problem for BPS).

Application	Comp./Comm. Ratio	Number VPs / Processor 8	4	2	1
SOR	Very Low	1.0	2.0	3.5	5.9
SOR	Low	1.0	2.0	3.5	6.2
SOR	High	1.0	1.7	4.2	7.0
BPS	Low	1.0	1.9	2.9	4.4
BPS	High	-	1.0	1.7	2.7
APUCS	Low	1.0	1.9	3.0	4.1
APUCS	High	1.0	1.8	2.8	3.9
PGL	Low	1.0	1.9	2.2	2.5
PGL	High	1.0	1.8	2.1	2.3

Table 6: LAPSE relative speedups on a 64 VP problem. Times are relative to 8 processors with 8 virtual processors per processor. (For BPS high, times are relative to 16 processors with 4 virtual processors per processor.)

6 Conclusions

This paper describes a tool, LAPSE, that supports parallelized direct execution and simulation of parallel message passing applications; we describe a synchronization protocol, Whoa, suitable for the direct-execution simulation of general message-passing systems, and provide performance data on LAPSE's implementation in the Intel Paragon multicomputer. The synchronization protocol is conservative, and exploits the observation that message-passing codes frequently exhibit long periods where their execution paths are insensitive to temporal considerations.

Using LAPSE, timing predictions of applications running on a large number of processors can be made by executing the application on a smaller number of processors and simultaneously running a timing simulation of the larger machine. LAPSE timing predictions were validated for four scientific and engineering applications. Typically the predictions were within 10% of the actual execution times; often they were within 5%.

The simulation speed was shown to depend on several factors: the computation to communication ratio of the application and the amount of application lookahead. For applications with good lookahead, slowdowns are modest and good simulation speedups are obtained. For applications without good lookahead, namely those whose execution paths depend on the answers to temporally sensitive questions, the slowdowns can be quite high. However, there are a number of possibilities for increasing lookahead and potentially increasing simulation speed for such applications. Consider first an application that calls clock routines. The current approach in LAPSE is to block the application until simulation time has advanced to the point at which the clock call is made, and then report the simulation time. However, in many applications, clock values are not used until well after they are set (e.g., in timing intervals). For such applications, it would be possible to continue running the application beyond the clock call, accept the simulation clock returns asynchronously, and only block the application when an unreturned clock value actually gets used (and then, only if simulation time has not yet advanced to the calling time). Transparent support of this mechanism would be costly since it would involve checking when certain storage locations change values (namely those into which the clock values are stored). However, a simple set of routines can be designed to read and subsequently use clock values. These would have to be inserted into the application in place of clock calls; however this should not pose a problem in the context of an automated instrumentation system that would most likely desire frequent clock calls. The second type of temporally sensitive call that can be handled is a query about the status of messages. For example, a probe asks "is there a message of a certain type here *now*?" LAPSE currently blocks the application until simulation time has advanced until the time at which the probe call was made, and then answers the probe. However, this approach is too conservative in certain cases. Suppose an application probes at some (as yet potentially unknown) time, say 100, and suppose that the simulator has advanced to time 50 at the time of the probe. If the probed for message is already present at time 50 (and messages can't be canceled), then it is guaranteed to be there at time 100. The simulator could so inform the application thereby unblocking the application. This requires somewhat more elaborate data structures and application/simulation flow-control than are currently in LAPSE (since, e.g., message types may be reused and the simulator must keep track of which message is destined for which probe). However, our experience with the PGL code indicates that such an optimization (which is really a more sophisticated form of lookahead) is well worth pursuing in order to accelerate the simulation.

There are a number of worthwhile extensions to LAPSE we are pursuing. First, LAPSE currently runs on the Paragon and provides Paragon timing estimates. We are porting LAPSE to work in conjunction with a software package, **nx-lib** [15], that provides the Paragon message-passing library on networks of workstations. **nx-lib** provides the workstations with Paragon functionality, our port will augment functionality with timing. Second, our model of the Paragon's operating system and hardware is currently fairly crude. For example our current network model is a pure delay network. We have implemented a packet-by-packet parallel simulator of the Paragon's mesh interconnection network and are in the process of integrating it with LAPSE. Our model of the Paragon's operating system is also simple and does not include models of some of the Paragon's internal algorithms, e.g., models of the way in which the Paragon manages communications buffers. We are planning to investigate how to incorporate such features into our model, and especially how lookahead calculations need to be changed for the resulting (more complex) models. In addition, we plan to investigate how to port LAPSE to run under and model the newly emerging MPI (Message Passing Interface [17]) standard on other parallel platforms besides the Paragon.

Acknowledgments

We are grateful to Jeff Earickson of Intel Supercomputers for his generous assistance in helping us to better understand the Intel Paragon system.

References

[1] J.H. Bramble, J.E. Pasciak, and A.H. Schatz. The construction of preconditioners for elliptic problems by substructuring, i. *Math. Comp.*, 47:103–134, 1986.

[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] D.-K. Chen, H.-M. Su, and P.-C. Yew. The impact of synchronization and granularity on parallel systems. In *Int'l. Symp. on Computer Architecture*, pages 239–248, May 1990.

[4] 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.

[5] R.C. Covington, S. Madala, V. Mehta, J.R. Jump, and J.B. Sinclair. The Rice Parallel Processing Testbed. In *Proceedings of the 1988 SIGMETRICS Conference*, pages 4–11, May 1988.

[6] T.W. Crockett and T. Orloff. A parallel rendering algorithm for MIMD architectures. Technical Report 91-3, ICASE, NASA Langley Research Center, Hampton Virginia, 1991.

[7] 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.

[8] P.M. Dickens, P. Heidelberger, and D.M. Nicol. A distributed memory LAPSE: Parallel simulation of message-passing programs. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS)*, Edinburgh, Scotland, 1994. The Society of Computer Simulation. to appear.

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

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

[11] R. M. Fujimoto and W. B. Campbell. Efficient instruction level simulation of computers. *Transactions of the Society for Computer Simulation*, 5(2):109–124, April 1988.

[12] 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.

[13] Intel Corporation. *Paragon User's Guide*, October 1993. Order Number 312489-002.

[14] D.E. Keyes and W.D. Gropp. A comparison of domain decomposition techniques for elliptic partial differential equations and their parallel implementation. *SIAM Journal on Scientific and Statistical Computing*, 8(2):s166–s202, March 1987.

[15] S. Lamberts, G. Stellner, A. Bode, and T. Ludwig. Paragon parallel programming environment on Sun workstations. In *Sun User Group Proceedings*, pages 87–98. Sun User Group, December 1993.

[16] I. Mathieson and R. Francis. A dynamic-trace-driven simulator for evaluating parallelism. In *Proceedings of the* 21^{st} *Hawaii International Conference on System Sciences*, pages 158–166, January 1988.

[17] Message Passing Interface Forum, The University of Tennessee, Knoxville, Tennessee. *Draft: Document for a Standard Message-Passing Interface*, November 2, 1993.

[18] D. Nicol and P. Heidelberger. Parallel simulation of markovian queueing networks using adaptive uniformization. In *Proceedings of the 1993 SIGMETRICS Conference*, pages 135–145, Santa Clara, CA, May 1993.

[19] D. Nicol, C. Micheal, and P. Inouye. Efficient aggregaton of multiple LP's in distributed memory parallel simulations. In *Proceedings of the 1989 Winter Simulation Conference*, pages 680–685, Washington, D.C., December 1989.

[20] D. M. Nicol and R. M. Fujimoto. Parallel simulation today. *Annals of Operations Research.* To appear.

[21] 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.

[22] D.M. Nicol. The cost of conservative synchronization in parallel discrete-event simulations. *Journal of the ACM*, 40(2):304–333, April 1993.

[23] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vettering. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, New York, 1988.

[24] 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.

[25] R. Righter and J.V. Walrand. Distributed simulation of discrete event systems. *Proceedings of the IEEE*, 77(1):99–113, January 1989.