RC 20205 (9/27/95)
Computer Science

IBM Research Report

PARALLEL EXECUTION FOR SERIAL SIMULATORS

David Nicol

Department of Computer Science
The College of William and Mary
Williamsburg, Virginia 23185

Philip Heidelberger

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

Research Division
Almaden · T.J. Watson · Tokyo · Zurich

Parallel Execution for Serial Simulators *

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

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

September 27, 1995

Abstract

This paper describes an approach to discrete event simulation modeling that appears to be effective for developing portable and efficient parallel execution of models of large distributed systems and communication networks. In this approach, the modeler develops sub-models with an existing sequential simulation modeling tool, using the full expressive power of the tool. A set of modeling language extensions permit automatically synchronized communication between sub-models; however, the automation requires that any such communication must take a non-zero amount of simulation time. Within this modeling paradigm, a variety of conservative synchronization protocols can transparently support conservative execution of sub-models on potentially different processors. A specific implementation of this approach, U.P.S. (Utilitarian Parallel Simulator), is described, along with performance results on the Intel Paragon and on the IBM SP2.

*Portions of this paper are reproduced with permission from "On Extending Parallelism to Serial Simulators", in *Proceedings of the 1995 Workshop on Parallel and Distributed Simulation.*

†This work is supported in part by NSF grant CCR-9201195. It is also supported in part by NASA contract number NAS1-19480 while the author was a consultant at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA, 23681.

1 Introduction

Few in the parallel discrete event simulation (PDES) community would argue with the assertion that PDES has not yet made a significant impact on discrete-event simulation practitioners. Indeed, this was the topic of a panel discussion at the 1994 PADS (Parallel and Distributed Simulation) Conference. During this panel discussion, a number of issues were raised and discussed:

1. This lack of impact is partially due to a lack of adequate PDES modeling tools, see also Fujimoto's discussion in [9] and Bagrodia's comment "The tools, stupid" [2]. The fact remains that efficient PDES requires the use of sophisticated techniques that are often application specific, platform specific, or both.

2. Fujimoto's "Holy Grail" does not currently exist. The "Holy Grail" provides arbitrary modeling capability and automatically parallelizes to yield significant speedups. Although one person out of a total attendance of approximately 80 disagreed, Fujimoto asserted that such a "Holy Grail" would not be available "in this century".

3. The best way for PDES to have impact in the near future is to focus on specific applications. One way to have maximum impact is to provide application specific PDES libraries or tools.

There is little to suggest that the "Holy Grail" will exist any time soon. PDES is an unusually tricky branch of parallel processing, and the overwhelming experience to date in parallel software development is that high performance requires hand-crafted attention to application and platform specifics, especially with respect to load-balancing, communication, and synchronization costs. Only recently have tools been developed to automate the generation of efficient communication and synchronization (e.g., High Performance Fortran), and even there the domain of applicability is limited to regular problems, and the user must map the workload to processors.

The existing parallel simulation literature partitions synchronization methodologies along "conservative" or "optimistic" lines. A conservative methodology is one in which a processor is never allowed to process an event out-of-order; an optimistic methodology is one in which a processor may process an event out-of-order, but is prepared to repair the error made, if any, upon discovery that previously executed events were done so out of order. Surveys [8] and [14] expound upon this and other issues concerning PDES. However, the ideas found in the PADS literature frequently lack a clear link to the computational models and the type of tool to which a simulation practitioner is accustomed. In some cases, the hoped-for goal of parallel simulation is different in the academic and industrial communities. The academic PDES work has focused almost exclusively on speedup, whereas one very real industrial application of PDES is in integration of separate simulators [24]. Here the goal is simply the *capability* of joining simulators, acceleration of execution is a secondary issue. While one can understand that a researcher's goals and resources are different from an industrial simulation user, it comes as no surprise that parallel simulation theory has apparently not yet had an impact on industrial simulation practice.

Recognizing some of these problems, a number of tools for parallel simulation have been developed, with the goal of making the parallelism and synchronization more transparent to the user. The Army funded an effort at the Jet Propulsion Lab to develop the Time Warp Operating System (TWOS), to support the development of parallelized combat simulations. TWOS uses the event-oriented paradigm, and while the TWOS group no longer exists, TWOS is still available from NASA's COSMIC software clearing house. Jade Simulation developed sim++, a process-view system which was designed from the bottom to support Time Warp simulation. The Army also supported a port of the commercial simulation language ModSim (developed by CACI) onto TWOS, and then supported a port of ModSim to Jade's system. While we shall leave to the historians the task of analyzing the story of these early efforts, it is safe to say that the efforts were technically ambitious, but did not meet with the hoped-for level of success (e.g., see [20]). Despite early hopes, issues related to synchronization (e.g., state-saving) ended up migrating to the modeler level by necessity, to escape unduly large overheads when purely automated means were employed.

An optimistic simulation tool, SPEEDES[22], may be licensed from the Jet Propulsion Lab; SPEEDES is based in C++, and has the event-oriented world view. Many of the details of synchronization are the responsibility of the modeler. For instance, the modeler must identify all variables whose states must be saved, and must provide the means for saving them.

A new commercial parallel simulation product, CPSim, has been announced by GTU, Inc. [12]. CPSim is event-oriented, and makes available several conservative synchronization techniques. The CPSim user writes simulation event processing routines in accordance with interfaces specified by CPSim.

A variety of parallel simulation tools have been developed at universities over the years, including OLPS [1], YAWNS [16, 18], YADDES[19], Maisie[3]. Each was built to demonstrate some limited aspect of parallel simulation, with the exception of Maisie, which is advertised as a general purpose tool. A common factor among all of these tools is that the simulation modeler must develop a model in the specific confines of that tool. Yet while the universities build tool prototypes for research purposes, the commercial (but serial) simulation industry has produced flexible and polished tools such as CSIM, SES-Workbench, BONeS, RESQ, G2, and ModSim, to name a few (see [23] for an up-to-date comparison of commercial tools). From a modeler's standpoint, the gap between these commercial tools and the publically available university tools is large, and is a contributing factor to the lack of impact the university tools have had. In addition, university tools have tended to be targeted to specific computing platforms, thereby further limiting their potential for widespread use.

As the mountain is not coming to the Prophet, evidently the Prophet must go to the mountain. Towards this end, this paper identifies a technically modest but general approach for extending parallelism to existing commercial quality simulation systems. Our hope is that through such efforts we may meet industrial simulators "half-way," get the industrial practitioners acquainted with parallel simulation by using it, but with their own tools. We also wish to demonstrate that parallel simulation may be attempted with relatively low risk, as no major tool redesign and development is required. The approach we describe is suitable for integration of different simulators, provided minor modifications to the simulation are made. With a (hoped-for) base of demand from experienced users of parallel simulation, commercial vendors are more likely to consider including support for parallel simulation in future releases of their products.

In this paper we identify a small set of capabilities a simulation tool must have to support extension to parallelism. We show then how one may develop extensions to that tool to support parallelism. We illustrate the methodology by example, describing our U.P.S. (Utilitarian Parallel Simulator) library for the CSIM simulation engine. We demonstrate the method's promise by providing performance results on different simulation models of computer and communication networks, executing both on the Intel Paragon and on the IBM SP2.

The remainder of this paper is organized as follows. Section 2 details the requirements and limitations on models, simulators, and synchronization methods necessary to support our approach. Section 3 then outlines the general approach, while Section 4 describes our U.P.S. library for CSIM. Section 5 discusses the synchronization protocols available in U.P.S. Section 6 illustrates a simple example of how a CSIM model is transformed into a U.P.S. model. Section 7 provides performance results, Section 8 gives our conclusions.

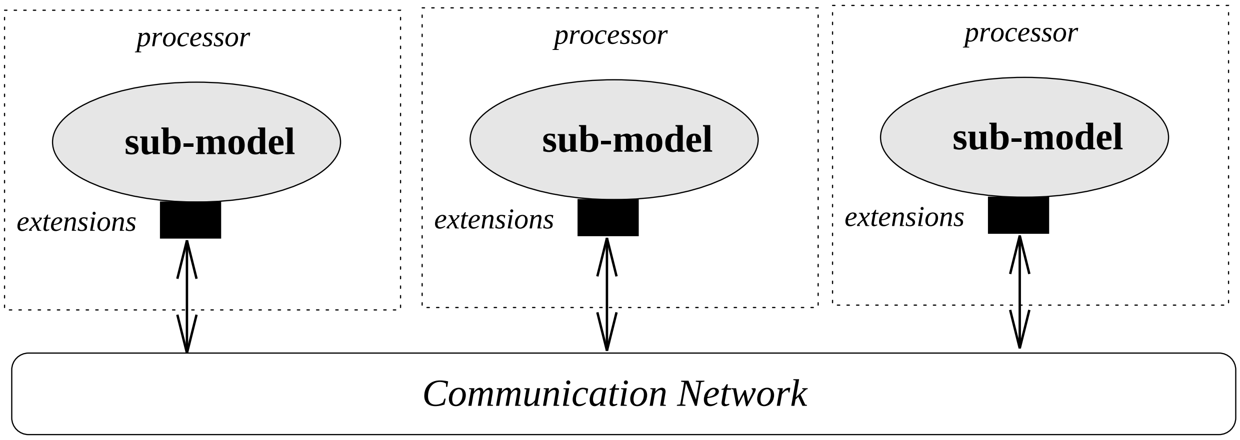

Figure 1: View of submodels and extensions

2 Requirements and Limitations

Industrial simulation tools are large, complex, and proprietary. If we are to extend parallel processing to such tools, we must accept as given that the tools be used with little or no modification. However, the other side of this coin is that a model developed for parallel simulation may be different than a model developed to execute serially, because the modeler must incorporate into the model constructs that trigger communication and synchronization. Our philosophical stand is that the parallel models ought to look very much like the serial ones, that one should be able to parallelize an existing model with a low level of effort. Our goal is to allow parallel simulations to be developed using the commercial tool to define sub-models on each processor, with extensions provided to define the interface between sub-models, and to automate the necessary communication and synchronization. (For model development and debugging purposes, one should also be able to place the submodels on the same processor, perhaps even within the same operating system process.) This approach is illustrated in Figure 1. A key point is that the interfaces identified have no knowledge of the simulation sub-model, except that which might be explicitly provided by the sub-model.

These considerations constrain the inter-processor synchronization to be conservative. This in turn constrains the models that may be parallelized, and constrains their partitioning. Furthermore, the burden of partitioning and mapping the simulation is the modeler's. This latter requirement is still usual for parallel processing in general; as for the former requirement, there is one essential model characteristic which allows us to synchronize automatically—lookahead. In this context, lookahead is the ability to predict (or to bound from below) a time in the future when one sub-model may affect another. A concrete example is a job entering service at a non-preemptive queue on sub-model i at time s and then being routed to another queue on sub-model j when its service is over at time t. Or, it may be that the routing is state-dependent, in which case i knows at time s that it may affect at time t any one of the queues to which it routes jobs, it simply doesn't know which one.

Lookahead is necessarily application dependent. However, in the context of computer and communication system simulations one generally uses standard abstractions such as queues, petri nets, or finite-state machines. The parallelizing extensions to such tools may provide special cases of these, designed in such a way that the lookahead calculation and dissemination may be automated.

The discussion above identifies restrictions and limitations on the synchronization method used by the extensions. The base simulation tool must also have characteristics that permit one to graft on extensions. Foremost is the need for the simulation to provide a "trapdoor" to a base computer language, such as C. This is absolutely required, to provide access to the synchronization and communication primitives available on the tool's platform. Second, the tool must provide a means for the extension to cause an extension-defined synchronization function or routine to be invoked at a specific future point in simulation time. That function must be able to block all further simulation processing until (through the extension's synchronization activity) it is logically safe to do so. A third requirement is that a tool permit one to reset the random number generator seed. This is required to allow the extension to create separate independent random number streams on each sub-model. We know these requirements are met by several simulation tools.

A last requirement is that the simulation be partitioned so that a message from one sub-model carries all of the information needed by the recipient to correctly continue processing of the passed information. As a counter-example, RESQ has mechanisms to "split" a job into sub-jobs, and later "join" those sub-jobs. RESQ's mechanism for identifying the siblings of a split job is to have them contain a memory pointer to their common parent. This constrains us from routing split jobs between sub-models, at least without modification to RESQ.

3 Approach

We now describe the basic approach for defining extensions.

Every commercial simulator designed for computer and communication networks provides a component from which queues may be built. The heart of our approach is to have the extension provide constructs that are functionally identical to a subset of the tool's own queueing constructs, but to build these constructs to provide the lookahead needed for the underlying synchronization method. Such a construct (which we call a *carrier*) is used when a job that enters service on one sub-model may be routed (upon completion of a non-zero service time) to a different sub-model. Carriers are the *only* mechanism through which one sub-model may directly cause simulation activity on another. A carrier on one sub-model deposits a *package* (message) at a *remote mailbox* associated with a different sub-model. The recipient sub-model must be designed to support receipt and processing of packages from other sub-models. Our approach also calls for support of a *query* between sub-models—sub-model i queries the state of sub-model j at simulation time t. To support this, i and j synchronize just as though i and j are sending messages to each other (which, in fact, they are). Queries are important, as the state of sub-model j can affect decisions made by sub-model i, yet it is sub-model i that must determine when those decisions are made. However, as queries must pass through carriers, there are non-zero delays between the sending, the receipt, and the satisfaction of the query. Note that a carrier can model an arbitrary delay of simulation time, using an infinite-server queue (which has no queueing delay) with a service time equal to the desired delay. One is permitted the full expressive power of the simulation tool—within a sub-model.

This restriction is a mild one in computer and communication models, which are composed primarily of queues. The simulation modeler just uses his/her understanding of the model to identify natural partitionings that satisfy this requirement.

The following questions are important when considering lookahead capabilities:

1. Can we predict future arrival times to queues (or lower bounds on them)?

2. Can we predict future service times (or lower bounds on them)?

3. Can we predict future departure times (or lower bounds on them)?

4. At what point do we know the composition of a message sent between sub-models?

5. Can we predict future routings?

6. Is there other special structure present that can be exploited? For example, is the queue's output uniformizable (see, e.g., [11])?

The answer to question (1) is "no" when the sole interaction between submodel and extensions is message arrivals, about whose behavior we have no future knowledge. One reasonable exception is if the simulation tool provides access to the time of the next event on its event list. This serves as a lower bound. Another reasonable exception is if the extensions explicitly include mechanisms for the submodel to predict lower bounds on future arrival times. We later discuss use of this very capability in our tool. The answer to question (2) is "yes" if all service times at a queue are random, are drawn at the queue, and are from the same state-independent distribution. In these circumstances we can pre-sample service times, even before the jobs whose service requirements are sampled arrive. A lower bound on service time is possible if every service time is at least as large as some $c > 0$. The answer to (3) is "yes", given restrictions on the queueing discipline. If the queue is FCFS, a job's departure time can be predicted exactly upon its arrival. Under other queueing disciplines (e.g. processor-sharing, round-robin) lower bounds on future departure times can be computed. Correct lower bounds can always be computed under any discipline such that a job's departure time cannot become earlier by virtue of an additional arrival to the queue. The answer to question (4) depends on the model. Sometimes the message reports a job transfer, with the identity of the job known at the time it enters service. Other times the message reflects the state of the sending sub-model at the time the job departs the server. The answer to question (5) is frequently "yes" in stochastic simulations where routings are chosen at random, independently of the state of the system. It is "no" if the routing is state-dependent, e.g., join the shortest queue among all destination possibilities. Question (6) asks whether special synchronization techniques can be applied. In particular, if the queue's output can be uniformized (which is true for a queue with a finite number of servers and Coxian phase-type service distributions), then techniques developed in [13] can be applied.

The degree to which a queue can predict its future behavior depends on the various considerations just listed. In our approach, when a sub-model specifies the carriers it will use, it also provides parameters that specify answers to the questions above. These parameters govern how lookahead is computed at the carrier. Only the author of the extensions needs to be concerned about the lookahead computations; the lookahead and synchronization based upon it is transparent to the simulation modeler. The single exception is when submodel-to-queue lookahead extensions are used, by explicit choice, in which case the model assumes a responsibility to report and maintain lower bounds on future arrival times to queues.

There are at least three well-studied conservative synchronization protocols suitable for the extension. The selection of a protocol is constrained by the model characteristics. The YAWNS window-based protocol [16, 18] is appropriate when the message associated with a job completion can always be generated and sent a minimum of some $X > 0$ (which may be randomly sampled) simulation time units before the job completes service. Barring any further refined lookahead, the YAWNS protocol will cause all processors to synchronize globally every $X_{\min}$ units of simulation time, where $X_{\min}$ is the minimum lookahead among all carriers in the simulation. YAWNS should be used if the connectivity between sub-models is high and the routing decisions cannot be predicted in advance. To attain good performance, this protocol does require that substantial sub-model simulation activity occur (on average) every $E[X_{\min}]$ units of simulation time, to amortize the cost of global synchronization.

The Appointments protocol [17] requires each carrier to maintain, for each sub-model to which it may route jobs, a lower bound (the appointment) on the next time at which it might next send a package there.

We will later describe a modification where this requirement is relaxed somewhat by placing the protocol in a window-based framework. The sub-model that creates the appointment is responsible for updating it. A sub-model that receives an appointment at time t will not advance past time t until the sub-model that established that appointment releases it to do so. Appointments should be used when a carrier's connectivity to other sub-models is low; they are especially effective if the service times and routing destinations can be pre-sampled.

The PUCS protocol [13] may be used when all carriers have Markovian service distributions that may be pre-sampled, and have routing distributions that may also be pre-sampled. The details are described elsewhere; essentially PUCS is the appointments protocol, with lookahead derived from the mathematical structure of the carriers.

Details of how one constructs carriers and communicates messages and appointments (and/or synchronizes globally) will vary with simulation system, and execution platform. However, the essential ingredients needed to support these extensions are access to an underlying computer language (e.g., C or C++), and the ability to schedule future events at known simulation times. Access to an underlying language permits access to communication and synchronization routines. An ability to schedule future events at known time-stamps gives a sub-model the ability to block at an appointment or synchronization instant. The sub-model just schedules an event that (in the case of appointments) waits for a message signaling release by the scheduling sub-model, or (in the case of YAWNS) engages in a global reduction to synchronize and establish the next synchronization point. Details of these operations in one specific case are provided in the following section.

4 U.P.S.

The Utilitarian Parallel Simulator (U.P.S.) is a library written to extend parallel processing to the CSIM simulation package [21]. CSIM is itself a library that allows one to write simulations in C or C++ using the process-oriented world-view. Given its embedding in these languages and its operations for scheduling future events, CSIM naturally satisfies the criteria we've identified as necessary to support a parallel extension package.

The key ideas in CSIM that we use and extend are those of *process*, *facility*, *mail-box*, *event*, and *storage*. CSIM is essentially a lightweight threads package for simulation; a process is a thread. Various CSIM calls manipulate the invocation and suspension of its processes. For instance, **hold**(t) causes the calling process to be suspended for t units of simulation time. Following the suspension, the process resumes execution at the statement following the **hold**. A process may also block waiting for an *event* (which is essentially a signal). Processes may either block on events, or set them. If one process sets a signal S at simulation t, then all processes that are blocked on S at time t are released. A *mail-box* is a place where one process can deposit a "message" (an integer or pointer to memory) for another process. A process may query the status of a mail-box, may block indefinitely waiting for a message to be deposited in a mail-box, or may block for a maximum certain (user-specified) duration waiting for a message. A *facility* is a queue; a process may block when calling **reserve** to acquire a facility, it must **release** the facility when it has finished. Facilities may have any number of servers, including infinitely many. A variety of queueing disciplines are supported. A *storage* models passive resources, such as memory. A process may block attempting to **allocate** some number of storage units; after using these the process **deallocates** the storage. Contention among processes for storage is resolved using process "priorities", which are user defined.

U.P.S. extends CSIM with the notions of *carrier*, *package*, *remote-mailbox*, and *station*. A carrier is simply a facility, with some restrictions that ensure an ability to compute lookahead. A carrier's declaration includes a constant c (possibly zero) and a pointer to a function r such that every service time may be computed by adding c to a random value obtained by calling $r()$. U.P.S. is therefore able to pre-sample service times. If the carrier has a finite number of servers, then the constant c may be equal to zero. If the carrier has an infinite number of servers we require $c > 0$; c (plus the next sub-model event time) is the only lookahead available in this case. Just as the modeler must tell CSIM about all the facilities with a **facility** call, a **carrier** call must be issued for each carrier.

Processes may create *packages* (variable-sized contiguous blocks of memory that hold a message) and pass them to a carrier using specific calls provided by U.P.S. Alternatively, a process may pass to the carrier a pointer to a function that is called to create the package, at the point in simulation time when the package is transferred from one sub-model to another. This mechanism allows the packages to contain state-dependent information that may not be known at the time the process originally submits a package request to the carrier.

A package ultimately ends up at a *station.* A station is declared by a recipient sub-model, with an ASCII name, and a CSIM mailbox as arguments. When a sub-model receives a package, the target station name is extracted from the package, and the list of stations is searched for the correct one. Upon discovery, the data portion of the package is copied into free memory, and a pointer to that memory is deposited in the associated CSIM mailbox precisely at the package's simulation arrival time.

To forge the connection between a carrier on one sub-model and a station on another, a sub-model declares a *remote-mailbox* for every station to which its carriers may route packages. A remote mailbox contains the name of the associated station, and the identity of the processor on which that station resides. Thus just as CSIM requires a **mailbox** call for each mailbox in the model, U.P.S. requires a **remote-mailbox** call for each remote-mailbox in the model. When a process gives a package to a carrier it may specify the remote-mailbox that describes the destination. U.P.S. then takes care of the actual package transmission and receipt.

It is also possible to associate a *routing distribution* with a carrier. In this case a process cedes to the carrier the selection of package destination, but provides a probability distribution that the carrier uses. This mechanism supports multi-casts; each element of the random routing table is a list of remote-mailboxes. When selected, every station associated with a remote-mailbox in the list receives a copy of the package.

In order to send a package, a CSIM process builds a package and calls an U.P.S. routine to cause this package to be delivered. In the case of a carrier without an associated routing distribution, this call is simply **ship**(carrier,msg,len,rbox) where carrier is the identity of the carrier, msg is the starting address of the package, len is the package's length, and rbox is the identity of the remote-mailbox to which the package is to be sent. In CSIM, messages are sent to ordinary mailboxes with the **send**(mailbox,msg) call, thus the U.P.S. call naturally extends the CSIM call. As regards time advancement, U.P.S. mimics normal CSIM usage by not returning control to the process until the delivery time of the package. Functionally then, passing a package to a carrier is no different from a CSIM process running through the sequence of acquiring a facility, holding it for the service duration, delivering a message in a mail-box at the service end, and releasing the facility. In order to receive a package sent by a process in a different sub-model, a CSIM process accesses a station just as though it is an ordinary CSIM mailbox, because it is an ordinary CSIM mailbox! The only purpose for distinguishing a station from a mailbox at declaration time is to allow U.P.S. to maintain a list of stations and their names, so that incoming messages can be properly delivered.

The lookahead used in carriers is derived automatically from carrier structure, and only from carrier structure. As a consequence, the lookahead available between submodels is limited to that of the carrier in the sense that the carrier lookahead calculations must also assume that a request to ship a package may occur *now.* This restriction may cause undo synchronization overhead when the carrier utilization is low. Better lookahead is possible if the submodel provides to carriers lower bounds on future times at which the submodel will ship a message. U.P.S. optionally provides such a mechanism, with three function calls. Immediately upon physical receipt of a message (e.g., possibly before the message's arrival time is reached) a sub-model calls **register_arrival()** once to register the message's existence, and bound from below the earliest time at which that message might enter (or cause another message to enter) any carrier in that submodel. This call returns an integer **handle** used by subsequent calls to **bound_arrival_time(handle, bound)**. This function may be called several times, as the bound is refined as a result of simulation activity. Finally, as a process terminates it calls **deregister_arrival(handle)**. U.P.S. maintains a min-priority heap of the registered processes, where the process's last bound is its key. Carriers use the top heap entry's bound as a lower bound on future arrivals, unless that bound is less than the current simulation clock. The use of such calls demands that the modeler have a deeper understanding of the model's behavior. However, these calls are optional; without them, the computation of lookahead is automatically handled by the U.P.S. carriers.

5 Synchronization Protocols

U.P.S. provides three synchronization protocols: YAWNS (Yet Another Windowing Network Simulator, see [16, 18]), WHOA (Windows, Halting On Appointments, see [6]), and PUCS (Parallel Uniformized Continuous-time Markov Chain Simulator, see [13]). One of these protocols is assigned to a carrier at its declaration; thus a U.P.S. model may use simultaneously any combination of these protocols. This feature allows one to tailor the synchronization method to the salient features of the carrier. For instance, we may use PUCS at carriers that have Markovian service times, YAWNS at carriers where package routing cannot be pre-sampled, and WHOA where carriers have general service distributions and stochastic routing. We now describe the mechanics of each protocol.

A carrier that uses YAWNS "pre-ships" its packages. This means that at the instance a package enters service at the carrier, the package is shipped to its destination, stamped with the package arrival time, i.e., the time at which the package leaves the carrier server. A YAWNS carrier pre-samples service times sufficiently far into the future (depending on the number of servers the carrier c has) to be able at any simulation time t to compute a lower bound $b_c(t)$ on the time of the next package it may pre-ship. If any carrier in the model is a YAWNS carrier, then all processors engage in the construction of and synchronization through YAWNS windows. If processors are synchronized at a YAWNS window at simulation time t, each processor i scans its YAWNS carriers and finds the earliest bound $b_c(t)$ among these (this minimum is arbitrarily large if the processor has no YAWNS carriers). The processors then compute the global minimum $w(t)$ of these values, synchronize again at simulation time $w(t)$, and repeat this procedure. Time $w(t)$ has the property that no package pre-shipped between synchronizations at t and $w(t)$ has an arrival time in $[t, w(t))$; the synchronization at time $w(t)$ ensures that all packages pre-shipped before time $w(t)$ are received and dealt with before the simulation times at which they affect their recipient submodels.

Both PUCS and WHOA use the notion of an "appointment"[17] between two processors, basically a time-stamped promise by one processor not to ship a new package with a receive-time less than the appointment. It is the sender's responsibility to update appointments as its state changes, it is the receiver's responsibility not to advance its simulation time clock beyond the least of its appointments. Freedom from deadlock is assured so long as appointments express positive lookahead (i.e., an appointment time is greater than the clock of the processor posting it). PUCS exploits the mathematical structure of Markovian servers. A PUCS carrier essentially establishes appointments based on the assumption that the server is always busy. Suppose that at simulation time t a job in a fully utilized server would depart. The PUCS carrier will have previously posted t as an appointment to the job's (pre-sampled) recipient. Upon reaching time t, the carrier randomly chooses whether to actually cause a job to leave, or not. The latter case is called a *pseudo event.* The probability of choosing a pseudo event is based on the carrier's actual state at time t. For instance, if the carrier has nothing in service at time t, the probability of a pseudo event is one. Following this choice, the carrier sends the result of the pseudo/non-pseudo decision to the recipient, and also sends it the time of the next possible job arrival—an appointment—assuming again that the server is fully utilized. Appointments are thus piggy-backed with packages, or with pseudo-event reports. No global synchronization is employed.

Like YAWNS, WHOA uses a global window. However WHOA's window width is fixed by the user. At the beginning of each such window, a WHOA carrier uses the probability distribution provided by the user to pre-sample the service times of all packages that might be shipped in the next window. For each such service time a routing destination is also pre-sampled. It is then possible to compute for each remote mailbox a lower bound on the arrival time of the next package sent there. If that lower bound falls within the next WHOA window, it is sent as an appointment. The upper edge of the next WHOA window serves as the de facto appointment for all other possible recipients. Suppose that time t is given as an appointment. At time t the sending carrier reassesses its state; it then either ships the warned-of package, or computes a new appointment based on the present carrier state. If a package is shipped, a new appointment time is computed and shipped with it, otherwise only a new appointment time is shipped. On the receiver's side, there is no difference between appointments generated by PUCS carriers, and appointments generated by WHOA carriers.

WHOA and PUCS are very similar in how they synchronize with appointments, but there is a critical difference. If ever a very small service time δ is sampled by WHOA, then appointments δ time apart will be generated until an actual package takes that service time. This leads to high overhead if the carrier is idle for a long period while its lookahead is only δ. Under PUCS, if a tiny service time δ is sampled, at the end of δ time PUCS will decide to ship a package or not (in which case a pseudo event is shipped), and then can resample another service time. PUCS does not get stuck with a small sample the way that WHOA can. On the other hand, PUCS is restricted in its service time distributions whereas WHOA is general. WHOA's deficiency is overcome when tiny service times are not possible.

6 Example

In this section we go through an example that compares a pure CSIM model with a functionally equivalent U.P.S. model. The example serves primarily to illustrate the U.P.S. model structure and how one sometimes needs to adapt a CSIM model for U.P.S.

The model is similar to one of the benchmark models of Section 7. The model simulates an open queueing network. There are C clusters of servers with F servers in each clusters. Server number 0 in each cluster is an "exit" server. Jobs arrive to each cluster in a Poisson stream; target queues for new arrivals are chosen uniformly at random. On leaving a non-exit server, a job is routed uniformly to another server in its cluster. When a job leaves the exit server, it departs the system with probability `xprob`, and it it otherwise is routed uniformly to a cluster (possibly the same cluster). Such a transferred job joins a queue in the new cluster that is chosen uniformly at random. CSIM code implementing this appears in Figure 2. In this model, but not shown, are variable and constant definitions. The variables `mean_iat`, `mean_s`, and `Termination` denote the the mean interarrival time per cluster, the mean service time at each queue, and amount of simulation time the model is to be run, respectively. Line (1) uses the CSIM type FACILITY to define an multidimensional array of facilities, one for each server in each cluster. Line (2) illustrates the first statement in any CSIM process, a call to `create`, a process creation statement. The first function executed in any CSIM model is named `sim`. Line (3) shows how one declares a single-server facility in CSIM, by calling CSIM

```
     #include <csim.h>
(1)  FACILITY f[C][F];     /* array of facilities */

     sim() {
     int i,j;
(2)     create("sim");
        for(i=0; i<C; i++)
           for(j=0; j<F; j++)
(3)             f[i][j] = facility("facility");
(4)     while( clock < Termination ) {
(5)             hold(expntl(mean_iat/C));
(6)             cust(random(0,C-1));
                }
     }

     cust(int starting_i) {
     int i,j,dead;
     {
     create("job");
     dead = 0;
     i = starting_i;
     while ( !dead) {
(7)    j = random(0,F-1);
(8)    use(f[i][j],expntl(mean_s));
       if (j == 0) {
(9)      dead =  (uniform(0,1) <= xprob);
(10)     if (!dead) i = random(0,C-1);
         }
       }
     }
```

Figure 2: CSIM model of open queueing network with uniform routing

routine `facility`, passing in a name (not necessarily unique). Normally one manipulates string variables to create unique names, but our examples omit this in the interest of brevity. Line (4) declares a loop where we continually (until the simulation is over) wait an exponential period until the next arrival and then create a job. In Line (6), we chose the starting cluster at random and then call a C function `cust` that will simulate the job starting at the initial cluster. The C function `cust` represents a CSIM process. The `create` statement creates a new process, enters that process on the ready queue, and returns control to the calling process. In this example the calling process (`sim`) immediately executes a `hold`, which is a directive to suspend the calling process for the indicated amount of time (a sample from the exponential distribution with mean `mean_iat/C`). CSIM then starts up a thread on the ready queue, which will be a `cust` process.

Lines (7) and (8) describe the life-cycle of a job. The next queue in the cluster is selected in Line (7) and the call to CSIM function `use` in Line (8) expresses the job's intent to get service at a facility. Here we choose one of the facilities at random, and tell `use` how much service is required by sampling a random number from the exponential distribution with mean `mean_s`. Function `use` suspends the process until its turn for the facility arrives, then resuspends the process for the amount of indicated service time, then reanimates the process. From the perspective of the job, the call to `use` advances simulation time by the (unknown) amount of time the job is enqueued, plus the service time. If the job is at an exit queue, Lines (9) and (10) represent the decision to leave the system, or select a new cluster. In CSIM, exiting a function that called `create` terminates that process.

Now consider an equivalent U.P.S. model, where the clusters are distributed among processors, one cluster per processor. This model is illustrated in Figures 3 and 4. The data declarations and constants of the CSIM model are assumed to be present also. The model is an SPMD (Single Program, Multiple Data) program, i.e., one copy of the program gets executed on each processor. In this U.P.S. model, the exit facilities get replaced by carriers, since each exit facility can feed jobs to facilities on another processor. In addition, an extra process is created on each processor that receives packages at its mailbox from other processors and routes the corresponding job appropriately.

Line (1) declares the facilities on the processor; since there is only one cluster per processor, the facility array is one dimensional rather than two dimensional as in the sequential model. Line (2) shows the U.P.S. declaration of a CARRIER. Line (3) declares a CSIM event, upon which a process may block. This one is used to allow U.P.S. to communicate termination to the main process. Line (4) declares variables that will hold the total number of processors involved in the parallel simulation and the identification of each individual processor. Line (5) declares a CSIM mailbox. In the U.P.S. model all job transfers from another processor to the subnet simulated by the processor will be reported through this mailbox. Line (6) declares an array of U.P.S. remote mailboxes. Line (7) is a user-defined function for sampling service times. The U.P.S. declaration of carriers requires a pointer to such a function, and to a memory block containing the function's arguments. In this case a block containing the service time mean will be passed. Entering `sim`, we first determine the number of processors in the parallel simulation (Line (8)) and a unique identifier for each processor (Line (9)). Each processor then creates its facilities in Line (10). The construction of a carrier in Line (11) requires a name, a pointer to a sampling function, a pointer to a memory block containing carrier-specific arguments for that function, the constant portion of the service time, the number of servers, and the chosen synchronization protocol. In a loop, one remote mailbox for each processor is created in Line (12); a remote mailbox is defined by the name that mailbox is given on its home processor, and the identity of that home processor. Line (13) declares the `Incoming` mailbox to be a U.P.S. station; the name declared here must match the name used by remote mailbox declarations on other processors that refer to this station. The call to `par_synch` on line (14) serves to initialize the U.P.S. internals. All carriers, remote mailboxes, and stations must be declared before this function is called. `par_synch` is not itself a process, although it does spawn some U.P.S. processes. Line (15) spawns a process `Transfers` that monitors `Incoming` for job transfers. The end of `sim` generates job arrivals, after which it awaits notification of termination from U.P.S. using the CSIM `wait` call (Line (16)).

Figure 4 illustrates the `Transfers` and `cust` processes. Line (1) tests whether the job is at an exit node; if not, it uses the corresponding facility. If the job is at an exit node, it stores in a variable, dead, the result of sampling whether the job exits the system after receiving service at the exit node. As U.P.S. itself performs the post-service routing, we cannot give service to the job and then decide whether to route it or eliminate it. Instead, we send the result of the exit decision as the package in Line (3) to a randomly chosen processor (cluster). (Alternatively, the carrier could have been defined to have a routing distribution

```
     #include <csim.h>
     #include <ups.h>
(1)  FACILITY f[F];
(2)  CARRIER c;
(3)  EVENT Done;
(4)  int procs, my_id ;
(5)  MBOX Incoming;
(6)  RMBOX rmbox[MB_MAX];
(7)  float extime(float *mean)  { return( expntl(*mean) ); }

     sim() {
     int j;
     create("sim");
(8)  procs  = ups_numnodes();
(9)  my_id = ups_mynode();
(10) for (j=1; j<F; j++) f[j] = facility("facility");
(11) c = carrier("car",extime,&mean_s,0.0,1,YAWNS);
(12) for (i=0; i<procs; i++) rmbox[i] = remote_mailbox("inbox",i);
(13) Incoming = station("inbox");
     Done = event("done");
(14) par_synch(Termination, 12345, Done);
(15) Transfers();
     while ( clock < Termination ) {
          hold(expntl(mean_iat));
          cust();
          }
(16) wait(Done);
     }
```

Figure 3: Main process for equivalent U.P.S. model

so that the carrier would select the next cluster itself.) Line (3) shows the U.P.S. call for shipping a message. The first argument is that of the carrier. The next argument is a pointer to the package, followed by the length in bytes of that package. The last argument selects a target remote mailbox, uniformly at random. After the package is shipped, the process corresponding to the job on the sending processor exits `cust`, thus terminating itself. Process `Transfers` loops through a sequence where it blocks using the CSIM function `receive` until a message is deposited in mailbox `Incoming` (Line (5)), then either spawns a `cust` process or not, depending on whether the package specifies that the job is dead (Line (6)).

Several points about the U.P.S. model should be emphasized. First, except for the initialization definitions and calls (e.g., `par_synch`), the U.P.S. model does not contain any explicit time synchronization calls. In addition, the complexity of dealing with a message-passing library are hidden from the user; these are replaced by the higher level functions `ship`, a U.P.S. construct that extends the CSIM `send` function, and the already existing CSIM `receive` function. Second, it is the user's responsibility to partition the model onto the parallel processors; in this case we chose one cluster per processor. Sometimes (but not always),

```
      cust() {
      int j,dead;
      create("job");
      dead = 0;
      while(!dead) {
        j = random(0,F-1);
(1)     if ( j != 0) use(f[j],expntl(mean_s));
        else {
(2)       dead = (uniform(0,1) <= xprob);
(3)       ship(c,&dead,sizeof(int),rmbox[random(0,procs-1)]);
          }
        }
      }

      Transfers() {
      int deadjob;
      create("transfers");
      while(clock < Termination) {
(4)         receive(Incoming, &deadjob);
(5)         if(!deadjob) cust();
            }
      }
```

Figure 4: `Transfers` and `cust` processes for equivalent U.P.S. model

this mapping involves re-numbering CSIM entities such as facilities from a global numbering system to a system involving the processor identification and a local numbering system. For example, in the above model, since there is only one cluster per processor, the parallel model used only a one dimensional array of facilities whereas the serial model used a two dimensional array. Third, the general structure of this model is representative of other larger, more complex models we have built. For example, there typically is a process very much like `Transfers` that simply picks up packages and distributes them appropriately.

7 Experiments

We now consider a number of examples of different U.P.S. models of distributed computing and communications systems. These examples serve both to demonstrate that U.P.S. can be used on a variety of problems and to explore performance issues.

7.1 Protocol Comparisons and Sensitivity

The first model is designed to allow direct parametric control over critical parameters, in order to conduct simulator performance sensitivity studies and to directly compare the performance of the the different synchronization protocols. The model is a network of central server models [4], similar to the models considered in [13]. The basic element in the model is a "cluster" of central server queuing network models, each comprised of one CPU and five IO devices. The CPU has exponential service with mean 2; the IO service time distribution uses a parameter $\alpha \in [0,1]$: the service time is the sum of constant 10α plus an exponential with mean $10(1-\alpha)$. One central server in a cluster serves as an exit point; a job entering this central server visits the CPU once, and visits an IO device once. On leaving the IO device that job departs the cluster. A job leaving any other IO device is routed to the exit central server with probability p_x. Failing that, it is routed back to the CPU device it last visited with probability p_f. Failing that, it is routed uniformly at random to a non-exit central server in the same cluster. Routing of jobs leaving an exit server depends on a "radius" parameter R. The system as a whole is viewed as a ring of clusters, with each cluster able to route jobs to clusters within R positions of it in the ring. On leaving a cluster a job is routed uniformly at random to one of the connecting clusters. At initialization, some J jobs are placed in the system. The network is closed.

The U.P.S. and CSIM versions of this model are very similar. The U.P.S. version declares the IO servers of one central server in each cluster to be a carrier with stochastic routing. A little bit of work goes into computing the routing distribution for that carrier. To establish a baseline, we measured the rate (in real time) at which jobs complete service at either a CPU or IO device, on one Paragon node. The CSIM version achieves a rate of approximately 2400 completions/second, over a range of problem sizes and parameter settings. The U.P.S. version (using YAWNS or WHOA) is about 10% slower, at 2150 completions/second. For the cluster parameters used below, both models are limited to about 64 clusters before paging sets in and performance plummets.

In all the data to following concerning this model, there are 100 central servers in each cluster, one cluster on each processor, and $p_f = 0.9$. The radius $R = 8$, except for runs where fewer than 8 processors are used, in which case the clusters are fully connected. In any given experiment all carriers use the same synchronization protocol. The first experiment considers the relative performance of all three protocols, on a problem set where IO times are purely exponential. The exit probability p_x is 0.01. We consider a low-load case where there is 0.1 job per central server on average, and a medium load case where there is 1 job per central server on average. Figure 5 presents data for these cases. The low-load case clearly shows that the exponential service time distribution is very disadvantageous for both YAWNS and WHOA. In YAWNS the exponential yields small window widths, in WHOA it occasionally allows very small inter-appointment times that take a long time to work past. YAWNS and WHOA performance improve markedly under higher load. This is understood as a natural consequence of amortizing the synchronization overhead over more workload. However, the same is true of PUCS; with increasing workload, increasingly more of the packages shipped report actual job transfers, not pseudo-events. The speedup (relative to the purely serial CSIM model) of PUCS using 32 processors is approximately 26, whereas the WHOA speedup is 18, and the YAWNS speedup is 12. It is evident that PUCS ought to be used on carriers with Markovian service times and stochastic routing.

Next we consider the sensitivity and relative performance of YAWNS and WHOA to workload, when the IO service time distribution has $\alpha = 0.5$, i.e., it is the constant 5 plus an exponential with mean 5. (Because of the constant, PUCS cannot be used in this case.) This is plotted in the left-hand graph in Figure 6. "Low" load is the same as in the previous experiments, "High" load is 5 jobs per central server, on average. Again we see that higher simulation workload better amortizes the synchronization overhead. But—significantly—we notice that both methods achieve vastly better performance under low load than they did with purely exponential IO service times. We also note that there is little to recommend one method over the other at these parameter settings. YAWNS does slightly better, probably due to lower overheads.

A clearer distinction between YAWNS and WHOA is seen in the right-hand graph in Figure 6 where

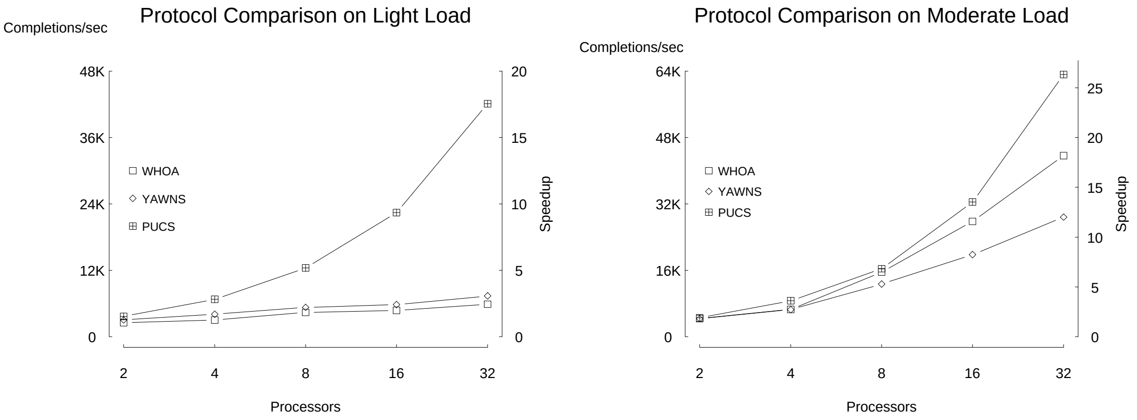

Figure 5: Comparison of all protocols on model with exponential service times

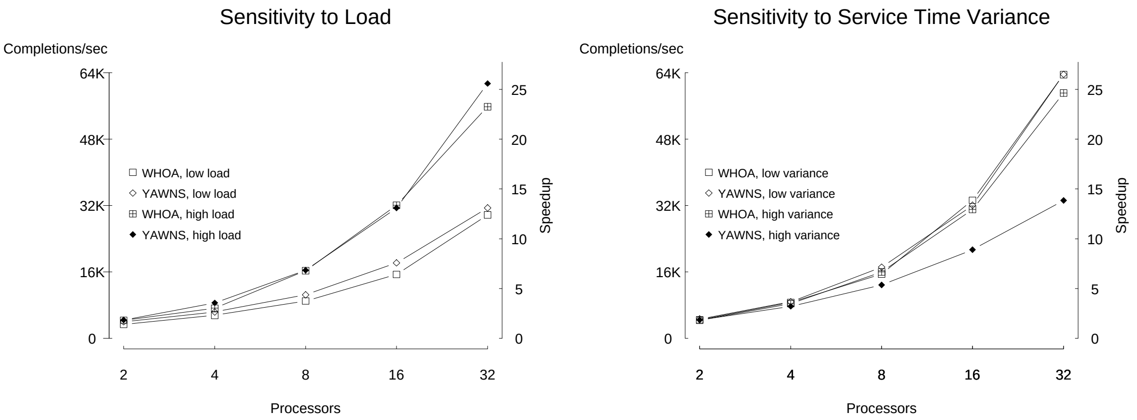

Figure 6: Sensitivity to load, and to service time variance (Intel Paragon)

we consider "Low" variance ($\alpha = 0.99$) and "High" variance ($\alpha = 0.01$) in the IO service time distribution. The load is medium, one job per central server on average. Under high variance, the YAWNS protocol does poorly with respect to WHOA, especially for large numbers of processors. Observe that the only difference between the high variance data in this graph, and the model represented in Figure 5 is a matter of a constant in the IO service time distribution that is only 1% of the mean IO service time. This difference has a large positive impact on WHOA's performance. Comparing the graphs in Figures 6 we see that WHOA has an advantage over YAWNS for high variance service times (provided there is some non-zero lower bound on them), but that the advantage is lost when the service time has a significant constant component.

We next consider the sensitivity of performance to the computation/communication ratio. This can be controlled directly by changing p_x. The left-hand graph in Figure 7 plots performance for high ($p_x = 0.01$), medium ($p_x = 0.1$) and low ($p_x = 0.5$) ratios. The load is one job per central server on average, the IO service time distribution is 5 plus an exponential with mean 5 ($\alpha = 0.5$), and $p_x = 0.01$. As one expects, the performance improves with increasing ratio. We also observe that the performance of YAWNS and WHOA relative to each does not change appreciably as the computation/communication ratio changes.

We also conducted experiments testing YAWNS and WHOA sensitivity with respect to connectivity

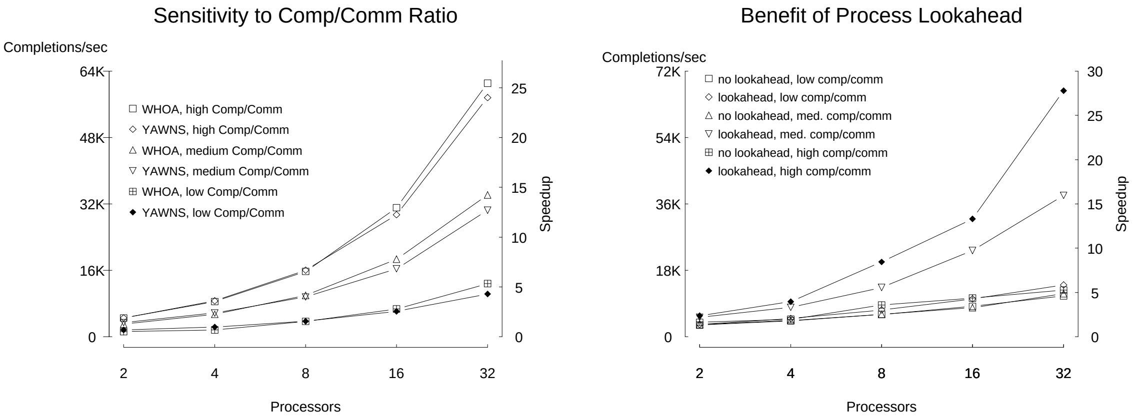

Figure 7: Sensitivity to computation/communication ratio, and effects of process lookahead (Intel Paragon)

parameter R. No appreciable sensitivity to R was observed as it ranged from $R = 1$ to $R = 32$.

We developed a new but closely related model in order to assess the benefit of the process lookahead extensions. Here, when a job first enters a cluster, we assign it $12/p_x$ units of total service in non-exit central servers (12 is the sum of the mean CPU and IO service times used in the earlier model). Then, as the job migrates between non-exit central server clusters, the sampled CPU and IO service times are subtracted from that service total, until the total is exhausted (note that the distribution of the time in the last central server before exiting is not the same as the others; this is an artifact of this purely illustrative example). At this point the job is routed to the exit central server. For every job, the remaining service total provides the process lookahead. **bound_arrival_time()** is called after every CPU or IO service completion, in order to update the appointment in case the job experienced queueing (which will increase the time before the job arrives at the carrier). The right-hand graph in Figure 7 illustrates the benefit of this lookahead as a function of p_x; YAWNS is used. Intuitively, the higher p_x is, the smaller the process lookahead will be. The other model parameters are ones under which YAWNS does poorly—exponential service times and low workload (0.1 job per central server, on average). We plot performance for high computation/communication ratio ($p_x = 0.001$), medium ($p_x = 0.01$), and low ($p_x = 0.1$). Performance is poor when the ratio is low, because overheads are high and there is very little process lookahead available. However, with a medium ratio we immediately see the benefit of process lookahead, and with a high ratio the benefits are remarkable.

7.2 Switching Network

A second problem is a network of switches, such as are used to establish long-distance phone calls. The model of a switch is very detailed; nearly fifty separate steps are involved in the processing at a switch of a single call. A network of N switches is full connected. External calls arrive at each switch in a Poisson process. The total processing associated with a call has three steps: (i) arrive at a switch, (ii) be routed to a separate switch and wait for the duration of the call, (iii) return to its source. The actual processing associated with each step is approximately the same.

The basis of the simulation is a single switch model written in CSIM, written in 1993. To this model we added carriers to provide interfaces between switches. Given the complexity of the switch model and its heavy reliance on global data structures, we found it expedient to construct a network model where each processor simulates exactly one switch.

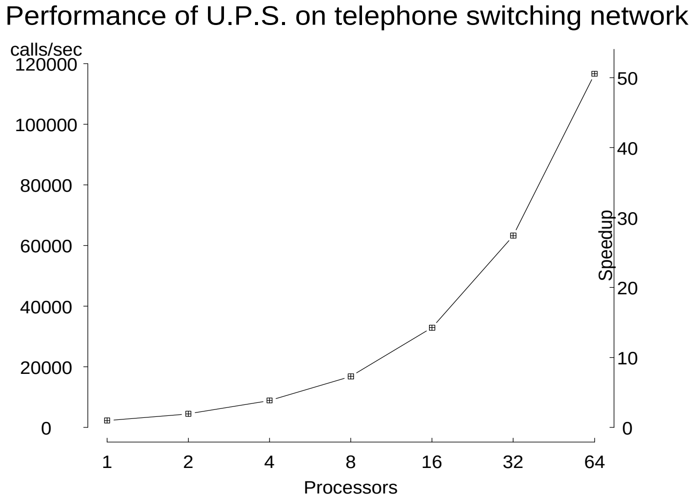

Figure 8: Intel Paragon performance of telephone switching network

We model the communication channel between switches with a pure-delay model, with a transmission delay approximately equal to the total processing time a call encounters at a switch. The appropriate synchronization method to use is YAWNS, whose window size will be this delay value. Figure 8 presents the performance data obtained from this model running on the Paragon. Earnshaw and Hind [7] have noted and demonstrated that communications networks are good candidates for parallel simulation; our experience with the switching network confirms this. There is a great deal of workload that can be simulated locally in the space of a single synchronization window, and hence one achieves good performance.

7.3 ATM Network

The third model we consider is a simplified version of an ATM switching network. The model consists of a mesh-connected array of switches. Packets are routed through the switches of the network using (X, Y) routing, i.e., by moving them in the X direction until the appropriate row is found, and then moving them in the Y direction until the appropriate column is found. Each switch is a 5×5 (buffered) crossbar. Four of the ports are used for routing traffic to different switches while the other two ports are used to accept externally generated packets and to deliver them once they have reached their destination. (This is similar to the switching elements used in the Intel Paragon.) The switch has both input and output buffers that are modeled by finite-sized storages. If a packet arrives at an input buffer that is already full, that packet is lost. In addition, there is a transfer storage (with one token) associated with each output buffer that is used to ensure that only one packet can be transferred into a given output buffer at a time. Movement of a packet through a switch requires 3 allocates, 3 deallocates, 3 holds, plus the acquisition and release of a facility modeling the the output port. In addition, there is a propagation delay for transferring packets from one switch to another. This propagation delay is modeled by a simple **hold** statement in the case that the sending and receiving switches are co-resident on the same processor while carriers are used if the switches are not co-resident. These propagation delays are deterministic, depending only on the distance between the switches. Thus the carriers have an infinite number of servers with the constant c equal to the propagation

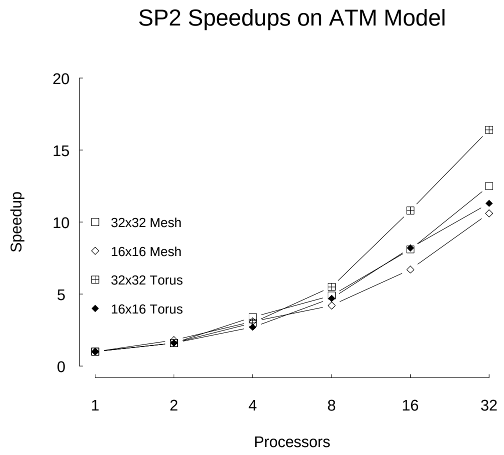

Figure 9: SP2 performance on ATM mesh-connected network model.

delay (and the random part equal to zero). Thus the model uses the YAWNS protocol.

Packets arrive from the outside according to a batch Poisson process; the batch size is uniformly distributed between one and some maximum value. This permits some burstiness in the arrival process. In this example, the Poisson arrival rate was 5,000/second/switch and the maximum batch size was 2 packets. When a batch of packets arrives, a destination switch is chosen uniformly among the other switches. This is something of a worst case as far as parallel simulation goes since it results in both a single packet crossing multiple processor boundaries and induces a load imbalance. (Switches in the center of the mesh are more heavily loaded than those at the edges.) More elaborate arrival processes and flow control (e.g., leaky buckets) could be added to the model; these would tend to increase the computation to communications ratio of the model.

Even at the speed of light, propagation delays down the fibers provide excellent lookahead. For example, suppose the output ports operate at 100 megabits/second and the switches are 50 miles apart (the parameters used in our model). Then it takes about 4 microseconds to put a 53 byte packet on the fiber, while it takes about 270 microseconds to transmit it from one switch to the next at the speed of light. Thus intra-switch events happen about 67 times faster than inter-switch events. This ratio provides excellent lookahead for the purpose of parallel simulation.

We simulated two versions of this model, one of a 16×16 mesh and one of a 32×32 mesh. The resulting models are quite large; e.g., the 32×32 model has up to 47,000 concurrently active CSIM processes. The Figure 9 illustrates the speedups when the model is run on the SP2. As expected, performance is better for the larger model, but the maximum speedup is 12.5 on 32 processors. The leveling off in speedup observed on more than 4 processors is due to several factors. First, the computation/communication ratio decreases since each processor has a smaller sub-mesh; for a given number of arrivals, the number of packages increases with the number of processors. Second, the load imbalance becomes more severe as the number of processors increases (beyond 4). For example, when a 16×16 mesh is simulated on 8 processors with 4×8 sub-meshes per processor, the busiest processor (simulating the interior of the mesh) carries about 40% more traffic than a lightly loaded processor (simulating a mesh corner). On 16 processors with 4×4 sub-meshes per processor, this ratio increases to about 90%.

To eliminate the load balancing effect, we modified the model so as to be torus-connected rather than mesh-connected (e.g., in an $M \times M$ torus-connected network, switch number $(0, j)$ is connected to both switches $(1, j)$ and $(M-1, j)$). While this is not necessarily a physically meaningful interconnection topology for an ATM network, it does allow us the opportunity to study the effect of load imbalance on U.P.S. performance. Packets are routed through the switches by selecting the shortest X distance, then the shortest Y distance. This equalizes the load on all processors. The Figure 9 also illustrates the speedups when the torus-connected model is run on the SP2. On more than 4 processors, the speedups are higher for the torus-connected model, reaching a maximum of 16.4 on 32 processors. On 4 or fewer processors, the speedups are slightly higher for the mesh-connected model since its load is already balanced and since its computation/communication ratio is somewhat higher; for a given number of arrivals, the number of packages in the two models is about the same, but more switches are traversed in the mesh-connected model.

We next examine in more detail some of the overheads involved in simulating a U.P.S. model using YAWNS. The major overheads are extra event processing, communications overhead due to sending and receiving packages, and window processing.

- **Extra Events**: In this model, each package `ship` is replaced by a CSIM `hold` in a purely sequential model. On the sending side, there is 1 event/ship just as in the sequential model. On the receiving side, there is an internal (i.e., not visible in the user's model) U.P.S. event to cause the package to arrive at the station at the right time, and two additional events in the user's model: a `receive` for the package at the station and a `create` to set up a process representing the newly arrived packet. Thus, each package requires 3 extra CSIM events to process that would not occur in a sequential model. In addition, there is an extra event scheduled at window times. For the 16×16 torus-connected model, U.P.S. executes approximately 10% more CSIM events on 8 processors than the sequential version. When run on 32 processors, the percentage of extra events increase to about 20% due to the increased number of packages for a given model size.

- **Communications Processing for Packages**: Each package `ship` requires at least two calls to a message-passing library routine, one for sending the message and (at least) one to probe for and receive the message on the destination processor. Thus a measure of a model's computation/communication ratio is the average number of CSIM events executed per package sent. For the 16×16 torus-connected model run on 8 processors, there are are about 29 events/package. This figure drops to 16 events/package when the number of processors is increased to 32. For the 32×32 model, there are 54 and 28 events/package on 8 and 32 processors, respectively.

- **Window Processing**: At each window, processors enter a reduction to both determine the next window time and to count messages so that a processor can be assured that it has received all the messages sent to it before it begins simulating the next window. This is a relatively expensive operation, especially on a large number of processors. A measure of the window processing overhead is the average number of events each processor executes per window. In the ATM model, the propagation delays are relatively high, so the window processing costs are small. For example, the 16×16 model executes about 2,800 and 800 events/processor/window when run on 8 and 32 processors, respectively. For the 32×32 model, these figures increase to about 20,000 and 5,000 events/processor/window on 8 and 32 processors, respectively.

These figures give an idea of the type of computation/communication ratios required to obtain reasonable speedups on the SP2. In particular, we need several tens of events/package and a large number of events/window (although not as large as above, as the next example will illustrate).

We should point out that the Paragon speedups for this model reported in [15] are somewhat overstated due to a problem in the CSIM calendar queue implementation that was recently uncovered. In CSIM, there is a minimum bucket width for the calendar entries. As the event rate (events/simulation time unit) increases, it is possible for more and more events to end up in the same bucket, and thus the performance of the simulator degrades. This problem is less pronounced in a parallel model, since each processor simulates fewer events than a serial model. This problem was avoided in this paper by appropriately scaling time.

7.4 Mobile Cellular Network

The final model we consider is a simplified model of a mobile cellular communications system. This is an important application area and models such as this have been used as benchmarks in other parallel simulation studies [5, 10]. In this model there are a number M of hexagonally shaped cells arranged in the plane. There are a fixed number of frequencies per cell and a fixed (Poisson) arrival rate of calls to each cell. Upon arrival to a cell, the call is accepted if there is an unoccupied frequency in that cell; otherwise it is blocked and discarded. With probability $(1-p)$, the call will stay in that cell for a randomly distributed service time and then exit the system, otherwise it will travel to one of its neighboring cells, chosen uniformly. There is a random travel time distribution. When such a traveling call arrives at the new cell, a hand-off is attempted; if the new cell has an available frequency, the hand-off is successful and the call allocates a frequency in the new cell and deallocates its frequency in the old cell. If no frequency is available, the hand-off is unsuccessful in which case the call deallocates its frequency in the old cell and exits the system. A call that is handed off successfully subsequently behaves like a newly arriving call. The purpose of the simulation is to estimate the fraction of blocked arrivals and and unsuccessful hand-offs.

We modeled the frequencies using CSIM storages and we used infinite server carriers (one per processor) to model the travel time. The parameters for our models were as follows. There were either $M = 1024$ or $M = 2048$ cells (arranged in 32×32 or 32×64 arrays), 20 frequencies/cell, and an arrival rate of 3 calls/minute/cell. The travel probability was $p = 0.25$. Non-traveling calls had an exponentially distributed service time with mean 3 minutes. The travel distribution was a constant k minutes plus an exponential distribution with mean $(6-k)$ minutes. The YAWNS protocol was used. The minimum travel time k determines the width of the windows. We used either $k = 0.1$ (low lookahead), or $k = 1.0$ (moderate lookahead). With these parameters, the average number of busy frequencies per cell is about 14 (utilization =0.70) and about 4% of the arrivals and hand-offs are blocked. These statistics are fairly insensitive to k. Thus on a single processor, with $M = 2048$ cells, there are approximately 28,000 calls in progress at a time which requires about 100Mbytes on a single node of an SP2. (Our machine has 128Mbytes/node.)

SP2 speedups for this model are shown in Figure 10. As seen earlier in the cluster model, the minimum service time at the carriers has a strong influence on speedup. In addition, speedup increases as the size of the model increases. For the moderate lookahead cases, the best speedups are just over 15 and 20 on 32 processors for 1024 and 2048 cells, respectively. For the 1024 cell model running on 32 processors, the computation/communication ratio is about 55 events/package, independent of the value of k. However, for $k = 0.1$, there are only 33 events/processor/window which accounts for the limited speedup. For $k = 1.0$, this increases by a factor of ten. For the 2048 cell model, the computation/communication ratio is approximately 80 events/package and there are approximately 63 and 630 events/window/processor for $k = 0.1$ and $k = 1.0$, respectively. Compared to the ATM model, the computation/communication ratio (as measured by the events/package) is higher, but the window processing costs are also higher.

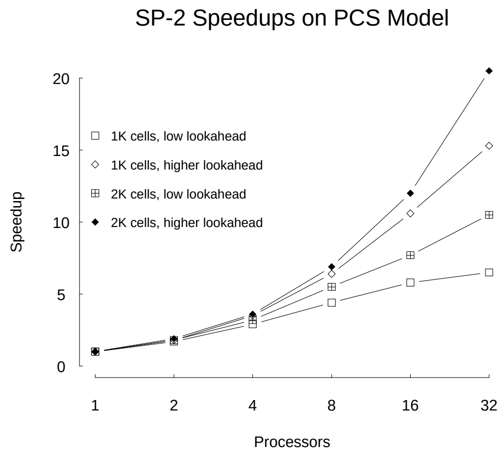

Figure 10: SP2 performance on mobile cellular network model.

8 Conclusions

If the techniques of parallel simulation are to make an impact on industrial simulation, those techniques must be used, and hidden, in the tools industrial simulators use. However, history teaches industry that writing a new simulation tool to support parallel simulation is a risky venture. We cannot expect industry to adopt parallel simulation until there is clear evidence of a need, and clear evidence of an ability to satisfy that need. We propose here a methodology for providing extensions to existing simulators, extensions that do not require modification to those simulators. The simulation modeler uses the tool as normal, with the additional tasks of partitioning the model into sub-models, one per processor, and incorporating the extension constructs into those sub-models. The extensions are designed to provide essentially the same functionality as basic constructs in the base simulator, but to do so in a way that the inter-processor synchronization and communication is automated. The distributed model looks very much like a non-distributed model. In U.P.S. , a small set of additional calls provide the parallel simulation capability. Almost all of the model is ordinary C and CSIM.

The methodology very much involves the modeler, who must partition the model, and must consider the performance tradeoffs between different synchronization strategies when performing that partition. However, the simulation modeler does not have to implement the communication and synchronization. That is done once, by the extension's author. This provides industry with a lower risk path to parallel processing, and an ability to experiment with it with low software cost. In view of the general perception of parallel simulation as tricky business, we believe this approach proves an important first step towards making parallel simulation safe for the masses.

We are exploring this methodology by writing an extension library, U.P.S. , for the commercial tool CSIM (now distributed by Mesquite Software). It is a natural match, as CSIM modelers must be C programmers, and should easily adapt to the interacting sub-models view of the simulation. A CSIM modeler needs to incorporate only a few U.P.S. constructs into their models. To use U.P.S. , it helps for a CSIM modeler to understand the fundamentals of SPMD programming, e.g., partitioning and mapping of submodels onto a parallel processor. U.P.S. currently runs on the Intel Paragon and on the IBM SP2 using the MPI message-passing library. As both CSIM and MPI have been ported to a variety of platforms, we expect U.P.S. to highly portable as well.

This paper provides results of U.P.S. performance using several different synchronization protocols on four models: a network of central server computer systems, a telephone switching network, an ATM network, and a mobile cellular network. We see that good performance is delivered on models where there is ample parallelism which is easily abstracted. On a message-passing parallel machine such as the SP2, our experience has been that in order to obtain significant speedups, models must

1. be large (hundreds of simultaneously active processes/processor),
2. have a high computation/communications ratio (several tens of events/package),
3. not impose excessive synchronization events (e.g., several hundreds of events/window/processor in YAWNS), and
4. be well load-balanced.

We are continuing to extend the capabilities of U.P.S. . Some of the extensions we are pursuing have been directly motivated by attempts to apply U.P.S. in practice. For example, we are adding support so as to make it easier to allocate a resource (CSIM storage) on a remote processor. We are also interested in expanding its user base and experimenting with U.P.S. ports to other platforms, especially networks of workstations and shared-memory multiprocessor workstations.

Acknowledgments

We gratefully acknowledge the contribution of Cathy Roberts who developed the detailed CSIM model of a switching center, and of Phil Dickens who ported CSIM to the i860 processor. Herb Schwetman provided valuable information about the internals of CSIM.

References

[1] M. Abrams. The object library for parallel simulation (OLPS). In In *Proceedings of the 1988 Winter Simulation Conference*, 210–219, IEEE Computer Society Press, 1988.

[2] R.L. Bagrodia. A survival guide for parallel simulation. *ORSA Journal on Computing*, vol. 5, no. 3, 234–235, 1993.

[3] R.L. Bagrodia and W.T. Liao. Maisie: A language for the design of efficient discrete-event simulations. *IEEE Transactions on Software Engineering*, vol. 20, no. 4, 225–238, April 1994.

[4] J.P. Buzen. Computational algorithms for closed queueing networks with exponential servers. *Commun. ACM*, vol. 16, no. 9, 527-531, September 1973.

[5] C.D. Carothers, R.M. Fujimoto and Y.-B. Lin. A case study in simulating PCS networks using Time Warp. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation (PADS '95)*, 87-94, IEEE Computer Society Press, 1995.

[6] 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)*, 32-38, IEEE Computer Society Press, 1994.

[7] R.W. Earnshaw and A. Hind. A parallel simulator for performance modeling of broadband telecommunication networks. In *Proceedings of the 1992 Winter Simulation Conference*, 1365–1373, IEEE Computer Society Press, 1992.

[8] R.M. Fujimoto, Parallel discrete event simulation. *Commun. ACM*, vol. 33, no. 10, 31-53, October 1990.

[9] R.M. Fujimoto. Parallel discrete event simulation: Will the field survive? *ORSA Journal on Computing*, vol. 5, no. 3, 213–230, 1993.

[10] A.G. Greenberg, B.D. Lubachevsky, D.M. Nicol and P.E. Wright. Efficient massively parallel simulation of dynamic channel assignment schemes for wireless cellular communications. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS)*, 187-194, IEEE Computer Society Press, 1994.

[11] D. Gross and D.R. Miller, The randomization technique as a modeling tool and solution procedure for transient Markov processes. *Operations Research*, vol. 32, no. 2, 343-361, March-April 1984.

[12] GTU, Inc., Arlington, VA. *CPSim 1.0 User's Guide and Reference Manual*, June 1994.

[13] P. Heidelberger and D.M. Nicol. Conservative parallel simulation of continuous time Markov chains using uniformization. *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 8, 906–921, 1993.

[14] D. Nicol and R. Fujimoto. Parallel simulation today. *Annals of Operations Research*, vol. 53, 249–286, 1994.

[15] D.M. Nicol and P. Heidelberger. On extending parallelism to serial simulators. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation (PADS '95)*, 60-67, IEEE Computer Society Press, 1995.

[16] D. Nicol, C. Micheal, and P. Inouye. Efficient aggregation of multiple LP's in distributed memory parallel simulations. In *Proceedings of the 1989 Winter Simulation Conference*, 680–685, IEEE Computer Society Press, 1989.

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

[18] D.M. Nicol. The cost of conservative synchronization in parallel discrete-event simulations. *Journal of the ACM*, vol. 40, no. 2, 304–333, April 1993.

[19] B.R. Preiss. The Yaddes distributed discrete event simulation specification language and execution environments. In *Distributed Simulation 1989*, volume 21, 139–144, SCS Simulation Series, 1989.

[20] D.O. Rich, R.E. Michelsen. An Assessment of the ModSim/TWOS Parallel Simulation Environment. In *Proceedings of the 1991 Winter Simulation Conference*, 509–518, IEEE Computer Society Press, 1991.

[21] H. Schwetman. CSIM : A C-based, process oriented simulation language. In *Proceedings of the 1986 Winter Simulation Conference*, 387–396, IEEE Computer Society Press, 1986.

[22] J.S. Steinman. Speedes: Synchronous parallel environment for emulation and discrete event simulation. In *Advances in Parallel and Distributed Simulation*, volume 23, 95–103. SCS Simulation Series, 1991.

[23] J. Swain. *Simulation Survey. OR/MS Today*, 64-79, August 1995.

[24] A.L. Wilson and R.M. Weatherly. The aggregate level simulation protocol: An evolving system. parallel simulations. In *Proceedings of the 1994 Winter Simulation Conference*, 781-787, IEEE Computer Society Press, 1994.