On Extending Parallelism to 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

December 9, 1994

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 using 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.

*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 [6] 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 is justifiably viewed from the outside as having little relevance to industrial simulation. The ideas found in that literature frequently lack a clear link to the computational models and the type of tool to which a simulation practitioner is accustomed. 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 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 (without support) 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 [12]). Even the most enthusiastic supporter of these efforts would agree with the assertion that 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. None of these products were widely used.

An optimistic simulation tool, SPEEDES[14], may be licensed from the Jet Propulsion Lab; SPEEDES is based in C++, and has the event-oriented world view. Little is known of its performance however, especially on large-scale parallel computers. 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 only just been announced by GTU, Inc. [5]. 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 [8, 10], YADDES[11], 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 and frequently idiosyncratic 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. 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.

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. 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 on the Intel Paragon.

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 provides performance results, Section 6 gives our conclusions.

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

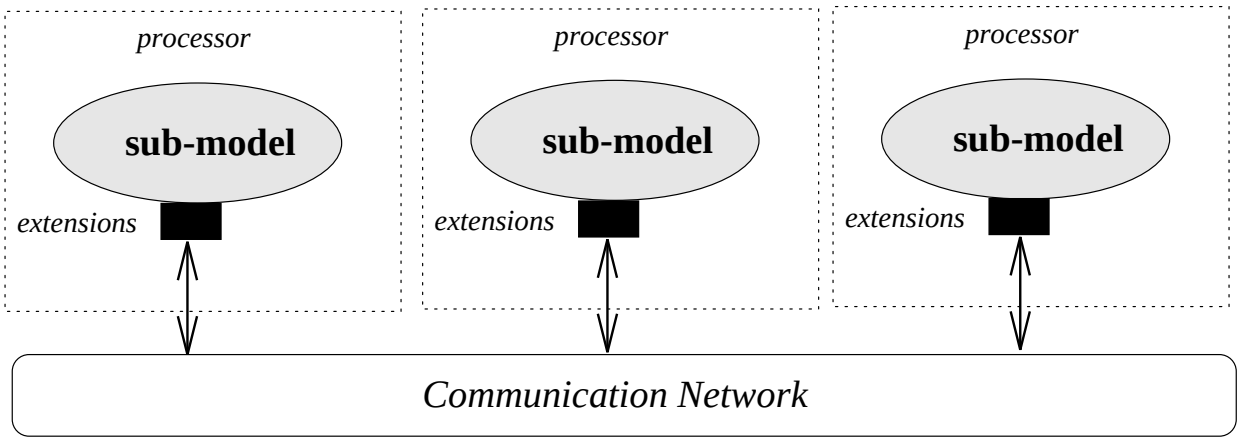

Figure 1: View of submodels and extensions

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 modelers. 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 lower bound) when in the future 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 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 (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?

The usual answer to question (1) is "no". Arrivals come from the sub-model, 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. 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 [7] 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 need be concerned about the lookahead computations; the lookahead and synchronization based upon it are transparent to the simulation modeler.

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 [8, 10] 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, 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 [9] 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. The sub-model that creates the appointment is responsible for updating them. 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 [7] 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 [13]. 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. The current implementation requires that a carrier's declaration include 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 which 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 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 name of the remote-mailbox to which the package is to be sent. In CSIM, messages are sent to ordinary mailboxes with a **send**(mailbox,msg), 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 U.P.S. handling of packages deserves comment. A sub-model can accept inter-processor messages only from within an U.P.S. extension routine. To avoid buffering problems we must ensure that an extension routine is called frequently to look for newly arrived messages, and move them into the simulator's address space. This is easily accomplished by creating a CSIM process that just loops through two activities (i) call **hold**(t) for some time t, (ii) check for, receive, and deal with any new messages. Currently, with YAWNS, this process is invoked only at end of windows at which time global synchronization is required; clearly this could (and should) be modified so as to pick up messages more frequently should there be heavy message traffic. Messages can also be sought from within any other U.P.S. routine. The routine that looks for messages just creates a CSIM process for that message, passing to the process the arrival time and other particulars. The newly arrived process does a **hold**() to suspend itself until the arrival instant, deposits the package into the appropriate station, and then departs. Appointments are handled slightly differently. A sub-model maintains a list of all exterior carriers that send it appointments. An "in" and an "out" appointment count is maintained for each. An appointment message always serves both to release the sub-model from the last appointment made by the carrier, and to establish a new appointment. Receiving an appointment message, a sub-model increments the "in" count associated with the sending carrier, holds until the appointment time, then enters a loop where it waits for the "in" count to increase beyond the "out" count. Inside of this loop the code looks for new messages, and calls **hold**(0), which gives any other process with the same activation time a chance to be processed. On exiting the loop, the carrier's "out" count is incremented, and the process terminates.

Presently U.P.S. supports the YAWNS and PUCS protocols. An U.P.S. statement declares which one is to be used. We are actively incorporating the appointments protocol, and considering how to allow mixtures

within the same simulation. We are also implementing support for queries, and for "remote-storages" which will allow one sub-model to contend for and acquire CSIM storage variables on another sub-model. Finally, the issue of transparently combining the statistics gathered at different sub-models is one that we must address.

5 Experiments

A preliminary version of U.P.S. is operational on the Intel Paragon. We report here on performance observed on that platform on three different models.

Our first model is of a large closed network of central server subsystems, similar to the model considered in [7]. Each subsystem has one CPU queue, which routes jobs to an I/O queue with 10 servers. Upon leaving an I/O server a job returns to that subsystem's CPU queue with probability p_r; it otherwise randomly chooses a different subsystem in the network, with equal probabilities. Each job's mean CPU service is 0.2 simulation time units, its mean I/O service time is 1 simulation time unit.

To provide a baseline, in a first experiment we developed a pure CSIM model of this system, and considered how performance varies as a function of problem size. Performance here is taken as the aggregate number of job service completions per second. We define a "basic network size" as 10 subsystems, initialized with 1 job per subsystem on average, with $p_r = 0.9$. We scale up the problem size by doubling the basic network size (all the while maintaining full connectivity between all subsystems in the network) several times, from 1 basic network to 128 basic networks. The effect is to vary the total number of subsystems from 10 to 1280, while maintaining the constant proportion of jobs to subsystems of 1. Figure 2 illustrates the results. The job completion rate drops as a logarithmic factor of problem size. This is to be expected, as the data structures used in the simulation's event list have logarithmic cost in the problem size. The runs associated with 640 and 1280 subsystems suffered paging overhead, evidenced by their non-proportional degradation from the other data points. Each Paragon node has 32Mb of memory, a sizable fraction of which is dedicated to its operating system.

In CSIM, the model above is constructed using a facility for each CPU queue, a facility for each multi-server I/O queue, and mailboxes where a CPU queue looks for job arrivals. Next we modified this model for U.P.S. just by making the multi-server I/O queue a carrier, and by setting up remote mailboxes and stations for every central server subsystem. Then, to investigate how well the YAWNS protocol might work under advantageous conditions, we assigned each I/O device a service time of 0.5 plus an exponential with mean 0.5. Figure 3 then plots the aggregate job completion rate as we vary the number of processors through powers-of-two, keeping the load on each processor fixed at one basic network size, as described earlier. The total problem size varies through the same values as were studied for the serial CSIM case. The 1 processor U.P.S. rate provides insight into the overheads of executing YAWNS on this problem, overheads due primarily to increased computation for lookahead and window calculations, and to extra logic in U.P.S. that is executed at the point a job leaves service at an I/O device. The model studied has a very high U.P.S. component, and the overhead slows the U.P.S. execution rate to about 70% of native CSIM. As we simultaneously increase the problem size and number of processors, we observe that the aggregate job completion rate increases. Compared with the corresponding native CSIM runs, we obtain speedups of 1.02, 1.8, 3.22, 6.20, 13.3, and 42.5 on power-of-two numbers of processors between 2 and 64.

By changing the I/O service distribution to be exponential we may use the PUCS synchronization mechanism. The results of doing so are presented in Figure 4 for two cases. The "light load" case is the same as the problem studied under YAWNS, except for the service times. The overheads in this situation are quite large for PUCS. Among the underlying synchronization messages, the ratio of overhead messages to messages

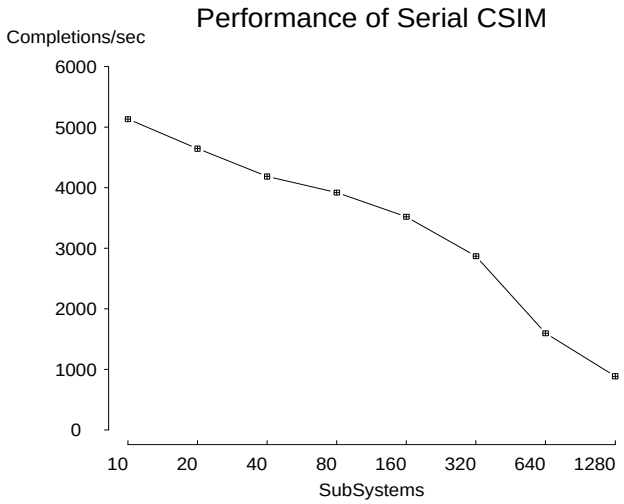

Figure 2: CSIM Job completion rate on model of fully connected network of central server subsystems, executing on one Intel Paragon processor.

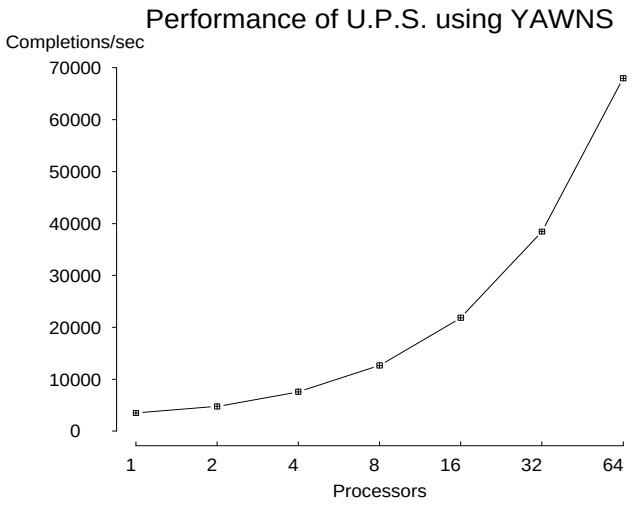

Figure 3: Performance of U.P.S. using the YAWNS protocol. I/O service distribution is 0.5 plus an exponential with mean 0.5.

that carry jobs is 10 to 1. By increasing p_r to 0.999 and quadrupling the total number of jobs, we achieve rather better performance. This is due both to a better computation to communication ratio (as a result of increasing p_r) and a better ratio of overhead messages to job-carrying messages, (1.5). The wide variation in performance shows the sensitivity of the method to problem characteristics. We also used YAWNS to simu-

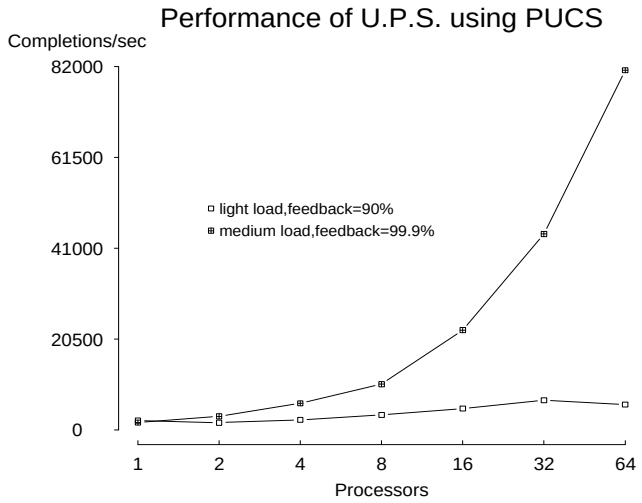

Figure 4: Performance of U.P.S. using the PUCS protocol. I/O service distribution is exponential with mean 1.0.

late the light workload model, and found it to be very disadvantageous. No problem/processor configuration achieved an aggregate rate larger than than 2000 completions per second.

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 two years ago. 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.

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 5 presents the performance data obtained from this model. Earnshaw and Hind [4] 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.

The third model we consider is a simplified version of an ATM switching network. The model consists of a mesh-connected array of switches. 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

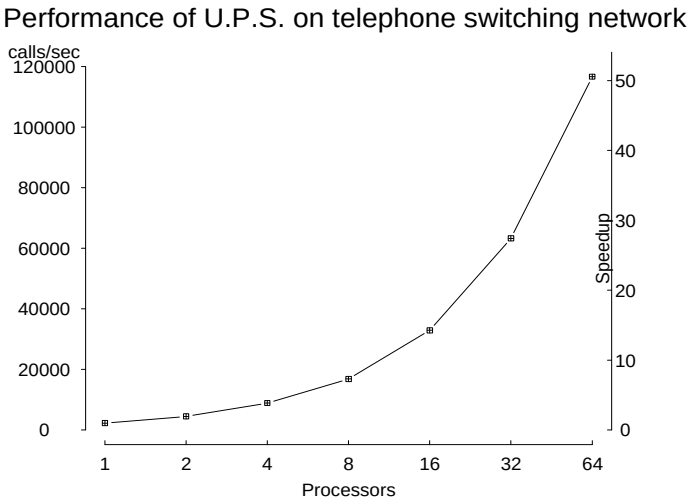

Figure 5: Performance of telephone switching network

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 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. 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.

A purely sequential version of this model (one that does not use any U.P.S. constructs) took several days to develop. Converting this model to incorporate the U.P.S. constructs took approximately another day. This primarily involved extending the model's data structures to support the notion of sub-models, which are basically just rectangular arrays of switches with one carrier and one station. This version of the model

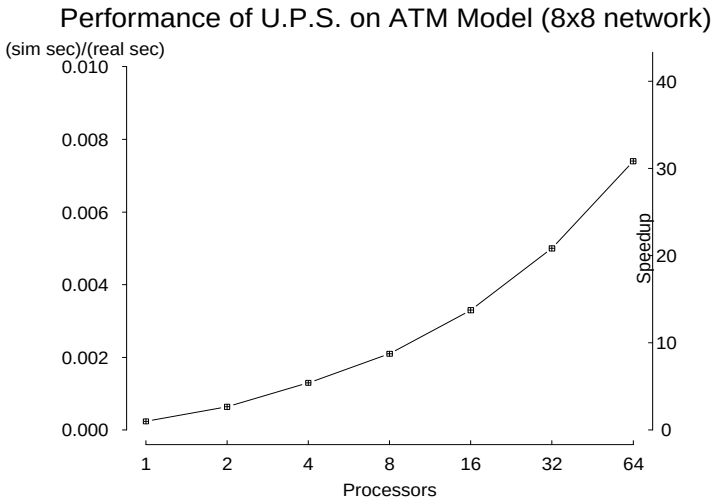

Figure 6: Performance on ATM network model

was debugged on a single processor. When put on the Paragon, it took an additional 15 minutes of debugging before the model ran correctly; the error was a simple indexing mistake involving the mapping of sub-models to processors. In developing this model, most of the attention was paid to model correctness aspects. (CSIM could benefit from a graphical interface!) No attention was paid to parallel simulation synchronization since U.P.S. provided that automatically.

We simulated two large models, one of an 8×8 mesh ("low load"), and one of a 16×16 mesh("high load") . Figure 6 illustrates how the overall rate of simulation time advance changes as a function of the number of processors used. Speedup (relative to the pure CSIM model) is proportional to these rates, and is also calibrated. We are pleased to again notice excellent performance, even when there is only one switch mapped to each processor. Some of this gain is due to using distributed memories, in addition to that due to concurrent execution. This is evidenced in Figure 7, where we consider the performance of the 16×16 switch model, under relatively light load. The metric plotted is the execution time, multiplied by the number of processors involved, normalized so that the 64 processor run has value 1. Under perfect speedup this metric will be flat at 1. With sub-linear speedup this metric will rise as a function of the number of processors used. In our case it does precisely the opposite, reflecting that the runs on large numbers of processors are super-linearly faster than the runs on small numbers of processors—the 64 processor run is 4 times faster than $(1/64)^{th}$ of the one processor run. Investigation into this behavior has shown that paging occurs on processor count up to 16 (which pages very lightly) whereas it does not on 32 and 64 processors. The flatness of the curve on large processor counts argues for essentially linear acceleration, when the effects of paging are discounted.

6 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

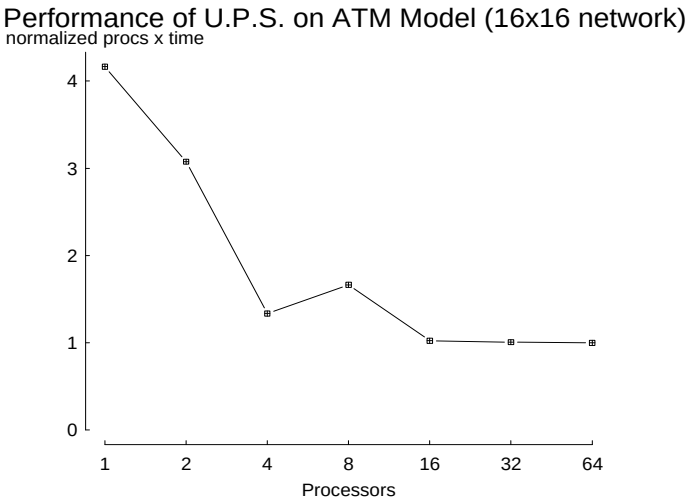

Figure 7: Normalized processor $\times$ time product, illustrating super-linear accelerations on large numbers of processors.

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, except he partitions the model into sub-models, one per processor, and incorporates 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. This paper provides preliminary performance results of its performance on three models: a network of central server computer systems, a telephone switching network, and an ATM network. We see that good performance is delivered on models where there is ample parallelism which is easily abstracted.

We are continuing to extend the capabilities of U.P.S. , and seek to expand its user base and to port it to other platforms. We are implementing an MPI (Message Passing Interface) version; MPI is the newly emerging standard for message-passing applications. When operational, U.P.S. will execute on any platform supporting both CSIM and MPI, which we expect to include most parallel systems as well as networks of workstations. In addition, we are particularly interesting in porting it to shared-memory multiprocessor workstations, as these are increasingly common.

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.

References

[1] M. Abrams. The object library for parallel simulation (OLPS). In In *Proceedings of the 1988 Winter Simulation Conference*, pages 210–219, San Diego, C.A., December 1988.

[2] R.L. Bagrodia. A survival guide for parallel simulation. *ORSA Journal on Computing*, 5(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*, 20(4):225–238, April 1994.

[4] R.W. Earnshaw and A. Hind. A parallel simulator for performance modeling of broadband telecommunication networks. In *Proceedings of the 1992 Winter Simulation Conference*, pages 1365–1373, Washington, D.C., December 1992.

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

[6] R.M. Fujimoto. Parallel discrete event simulation: Will the field survive? *ORSA Journal on Computing*, 5(3): 213–230, 1993.

[7] P. Heidelberger and D.M. Nicol. Conservative parallel simulation of continuous time Markov chains using uniformization. *IEEE Transactions on Parallel and Distributed Systems*, 4(8): 906–921, 1993.

[8] 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*, pages 680–685, Washington, D.C., December 1989.

[9] 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.

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

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

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

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

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