Proceedings of the 1989 Winter Simulation Conference
E.A. MacNair, K.J. Musselman, P. Heidelberger (eds.)

Efficient Aggregation of Multiple LPs in Distributed Memory Parallel Simulations

David M. Nicol Chris C. Micheal Patrick Inouye

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

ABSTRACT

The state of research in parallel simulation now demands that we experiment with a multitude of simulation models. It is evident that large-scale simulations involving many interacting logical processes should be a focal point of such experimentation, as large-scale simulations will benefit the most from parallelism. This realization raises a number of issues. Large-scale parallel simulations must aggregate many logical processes onto each machine in a distributed memory architecture. This fact creates internal management problems---how does one synchronize in such a setting? How does one efficiently find and manage the simulation workload? If we are to experiment with multiple models, what underlying functions can we extract to program once, and use many times? This paper describes YAWNS, *Yet Another Windowing Network Simulator*, for dealing with these problems.

1. INTRODUCTION

The present state of research in parallel simulation calls for extensive experimentation on a large number of simulation models. Naturally, we would like to develop each model as quickly as possible, by using common program components on all the models. Thus, one immediate problem is the development of an environment to support rapid development of parallel simulations. Another problem stems from the type of simulation which has the most promise for parallelization: large-scale simulation models with many logical processes (LPs). Given the additional difficulty and added overheads of parallel processing, only those models which are too large to be handled serially will likely be parallelized in practice. These models, when mapped onto a parallel architecture having only distributed memory, must aggregate the many LPs assigned to each processor. In addition to the usual problems of synchronization and load balance, these aggregated systems must use efficient means of managing the synchronization, communication, and event executions.

As researchers active in the study of synchronization methods and load balancing, we have undertaken the implementation of a parallel simulation testbed, *Yet Another Windowing Network Simulator* (YAWNS) which addresses these problems. YAWNS is being implemented on our Intel iPSC/2, and will serve as the basis for our studies in the parallel simulation of timed Petri-nets, queueing networks, Monte Carlo physical simulations, and highly reliable multiprocessor computing systems. YAWNS provides a software layer between the LPs, and the communication network. All communication and synchronization between processors is initiated and controlled by individual version of YAWNS running on each processor, as shown in figure 1. YAWNS uses a both a new synchronous synchronization protocol, and a new "multi-order" priority-queue structure. This paper describes the protocol, the general YAWNS design, and the multi-order priority-queue.

As a tool for studying parallel simulations, YAWNS is related to Reynold's SPECTRUM testbed [7].

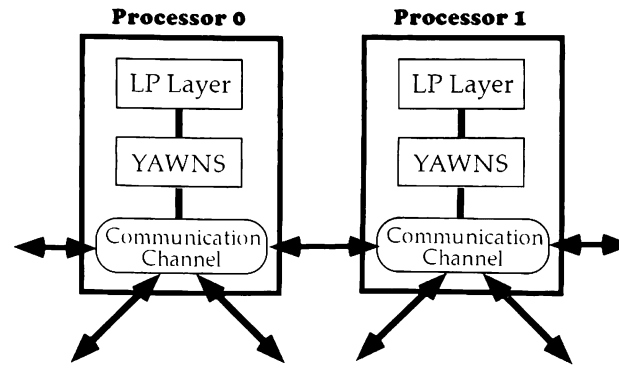

Figure 1: YAWNS Topology

SPECTRUM's intention is to provide a common testbed where a variety of synchronization protocols can be used on a single given simulation model. YAWNS is also related to Abram's common language interface [1], which takes a simulation model and parallelizes it using one of a specified set of synchronization protocols. The testbed developed by Wagner [8] is intended to support development of parallel simulation models on a shared-memory architecture. YAWNS notable advantage over all of these systems is that its synchronization protocol rests on theory which proves that for sufficiently large problems, YAWNS will achieve good performance.

2. SYNCHRONIZATION PROTOCOL

The YAWNS design is largely driven by its synchronization protocol. This protocol assumes the simulation is composed of *activities*, each with a distinct beginning and completion point. *Events* punctuate the beginning and ending of activities; the time difference between an activity's beginning and completion point is its *duration*. Like most protocols, YAWNS assumes that each LP manages its own event list. Unlike the most widely cited protocols, YAWNS is synchronous. In this it follows in the footsteps of [2,3,5]. Processors periodically synchronize globally, and cooperatively compute a time window within which they will process events. In YAWNS the window is chosen so that all LPs with events in the window can execute those events in parallel with other LPs. YAWNS ability to do so (at least in the simplest version, to be described) rests on some assumptions which are often met in practice:

1. Activities which advance the simulation clock have distinct beginning and ending points (e.g.,when a job enters service, and when it leaves service).

2. Once an activity has begun, its ending time is not affected by any other occurrence in the simulation (e.g., non-preemptive queueing).

3. Other LPs are affected by the activity only upon its completion.

4. The simulation time delay between an activity's beginning and ending points can be selected in advance.

5. The LPs to be affected by the activity's completion are known at the time the event begins.

6. Only one activity can occur at a time at an LP.

More sophisticated versions of the protocol are under development, and will relax some of these assumptions, notably 2, 4, 5, and 6.

In defining a window, YAWNS goes through three steps. The lower edge of the window is known, as it served as the upper edge of the previous window. In the first step, every LP determines whether it is active with an activity which began in the previous window. If so, the LPs which will be affected by the activity's completion are notified of that completion time. This step terminates with a global synchronization. In the second step each LP inserts any completion notifications it has just received as events into its event list. From here, each LP participates further in this window only if its event list is not empty. Each participating LP determines the completion time of its next activity *which has not yet begun*, under the (possibly false) assumption that no further completion notifications will be received. The key to this calculation is the assumed ability to pre-compute the difference between an event's beginning and ending time. The LP *can* determine the starting time of its next activity---that time is in the event list. It need only add the known duration of that activity to its starting time to compute the value of interest. The second step terminates with a global synchronization wherein the processors cooperatively compute the minimum "next-activity-completion-time" among all LPs in the system. That minimum time becomes the upper edge of the window, and is known to all processors. In the third and final step each LP executes all events with time-stamps strictly less than the upper window edge.

We have analyzed this protocol in a stochastic setting [6], where we have derived lower bounds on the number of events processed per window: these bounds grow arbitrarily large as volume of simulation activity increases. We have also shown that the performance achieved using the protocol approaches optimality as the number of events processed per window increases. The protocol's effectiveness is demonstrated on the following model. Consider a domain containing a countable number of points. An object resides at a point for 1+X units of time, where X is an exponential random variable.

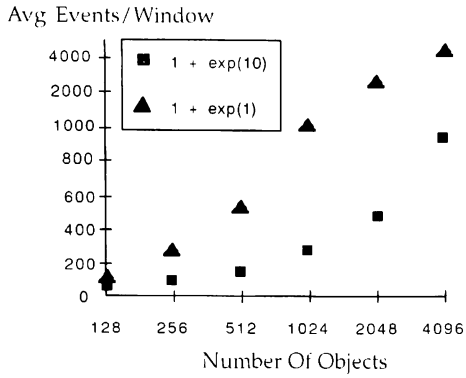

Figure 2: Avg Events/Window in YAWNS Model

The object then moves to any other unoccupied point in the domain. Events model the arrival and departures of objects from points in the domain. Figure 2 plots the average number of events processed in a window, as a function of the number of objects. Two plots are shown, one where X has mean 1, the other where X has mean 10. Here we see that the parallel workload increases linearly in the number of objects. It is evident that on large problems, a large amount of parallelizable activity is identified by the YAWNS , especially when duration times have a significant constant component.

The simulation modeler using YAWNS must be aware of its underlying demands and assumptions. Unlike many other protocols, the YAWNS protocol makes no attempt to parallelize, without programmer aid, any and all simulation models. Instead, the YAWNS philosophy is restrict attention to a class of useful simulation models, and to ask the programmer to provide a small amount of readily accessible information. As we have just seen, the benefits can be substantial.

3. YAWNS INTERFACE

YAWNS singles out *completion events* for special attention. A completion event is one which can cause other events to be scheduled at other LPs. For example, the departure of a job from a queue is a completion event. YAWNS interfaces with an LP through the structure below, which contains information needed by the YAWNS protocol.

```
struct LOGICALPROCESS {
    int LPId;
    double TimeOfNextEvent;
    long NextEventId;
    char CompletionEvent;
    double ActivityStartTime;
    struct CommList *AffectedLPs;
    double TimeOfFollowingCompletion;
    };
```

LPId records the LP's identity. TimeOfNextEvent records the time-stamp of the event at the head of the LPs event list; NextEventId is that event's identity (each event has a unique numeric identity). CompletionEvent is 1 if the LPs next event is a completion event, it is otherwise 0. If the next event is a completion event, ActivityStartTime records the time-stamp of the activity's start event. The CommList structure holds an LP id, a pointer to a message record to be passed to LP, and a pointer to another LPList structure. If the LP's next event is a completion event, AffectedLPs points to a null terminated list of LPs who are affected by the completion. TimeOfFollowingCompletion records a lower bound on the time at which the next event which hasn't yet begun can complete. This is the field whose minimum value over the entire simulation defines the upper edge of a YAWNS window.

YAWNS and the LP event execution routines interact in two ways. The first concerns event executions. Suppose that YAWNS identifies LP_1 as one with executable events. YAWNS calls a user-written routine *DoLPWork(LPId,t)*, passing an LP identity number, in this case 1, and a time limit *t*. *DoLPWork()* then executes all events in LP_1's event list having a time stamp strictly less than *t*. Upon return from the call to *DoLPWork()*, YAWNS assumes that the values in the LP record for LP_1 reflect its state following all of the simulation work just executed. YAWNS then uses this information to update its internal data structures.

A second interaction occurs when YAWNS passes messages to LPs. YAWNS calls a user-written function *AcceptLPMsg(LPid,MsgPtr)* to pass a message pointed to by *MsgPtr* to LP number *LPId*. It is anticipated that part of an LP's accepting a message involves inserting an event into its event list. If that insertion caused a change to any value in the LP's interface record, *AcceptLPMsg()* returns the value 1, it otherwise returns 0. Receiving a 1, YAWNS is warned to update its own data structures.

The only requirement YAWNS makes of the programmer is that the rules governing this simple interface be followed.

We turn now to a description of how YAWNS handles the synchronization, communication, and control aspects of the simulation.

4. YAWNS OPERATIONS

YAWNS is responsible for all synchronization, communication, and control in the parallel simulation. The YAWNS synchronization protocol determines when each of these activities occurs. Consider the first step in the processing of a window: each LP with an activity that began in the previous window sends completion messages to LPs affected by the completion. Under the interface protocol between YAWNS and LP processing, the information YAWNS needs to perform this communication has already been supplied to it, in the last window. YAWNS maintains a priority-queue of LPs organized by their latest activity start time. It consequently can find all LPs with activities which started in the previous window, go to their interface records, and retrieve the messages left there. These messages are passed to the YAWNS communication layer, who determines which processor is to receive each message, and places the message in a buffer specifically allocated for that processor. In the simplest, but most memory intensive case, each LP holds an array describing the complete LP to processor mapping. Other schemes are possible: for example, one might concisely define a *mapping function* which determines an LP's processor as a function of the LP index. Once all these messages are properly placed, the individual processors engage in a global *crystal-routing* [4] which causes all messages addressed to each processor to be routed to it. Each processor's YAWNS system then distributes the received messages by calling *AcceptLPMsg()*.

In preparation for finding the minimum TimeOfFollowingCompletion value, YAWNS maintains a priority-queue of LPs, organized by their TimeOfFollowingCompletion values. An LP's value can be affected by accepting a completion message, so YAWNS monitors the value returned by each *AcceptLPMsg()* call, and updates this priority-queue whenever a 1 is returned. Consequently, when all messages have been distributed, YAWNS can quickly identify the least TimeOfFollowingCompletion value among all LPs on the processor. This value is passed to a global minimum-computing routine, (this is the *gdlow()* system call in the Intel iPSC/2) where the processors cooperatively compute the minimum TimeOfFollowingCompletion value in the entire simulation. This minimum is returned to each processor, and is the upper edge of the current window.

YAWNS maintains a priority-queue of LPs, organized by their TimeOfNextEvent value. Once the upper edge of the window is known, YAWNS can initiate the processing of all events in the window. YAWNS enters a loop where it selects the LP with least TimeOfNextEvent falling within the window, calls *DoLPWork()*, and then updates the LP's entries in the ActivityStartTime, TimeOfFollowingCompletion, and TimeOfNextEvent
priority-queues using the new information found in the LP's interface record. This loop terminates when the least TimeOfNextEvent on the processor is greater than or equal to the upper window edge. At this point the processor has completed all processing associated with the window, and can begin processing for the following window.

We have seen that three distinct priority-queues are used by YAWNS in the course of its processing. Each priority-queue organizes the same subset of LPs---those with non-empty event lists---on different priority keys. We have combined those three priority-queues into a single structure called a *multi-order priority-queue*, which we describe in the following section.

5. MULTI-ORDER PRIORITY-QUEUE

YAWNS maintains three priority-queues over the set of LPs with non-empty event lists. These queues are organized around three different priority values. We could simply create three separate queues and manage them separately, but certain optimizations may be possible if we exploit commonality between the queues. First, these queues are over the same set of LPs. That set changes dynamically, but it is the same set for all three priority-queues. Secondly, when an LP changes one of its priority values, it is likely to have changed another. This means that when one priority queue has to be adjusted due to a change in the LP's value, the other ones may as well. The multi-order priority-queue is a means by which we can exploit this commonality.

The multi-order priority-queue is most easily explained by first describing its operations when there is only one priority value, say, TimeOfNextEvent. The idea is simply to build a binary combining tree over the collection of LPs. The LPs serve as leaves; each non-empty LP offers its TimeOfNextEvent value to the next tree layer. Each node in that layer is responsible for determining which of two LPs has the smaller time. It offers that time to the next layer up, and so on. This is illustrated in figure 3(a), which also illustrates the structural links between nodes. The multi-order priority-queue is the obvious generalization of this: each LP offers a vector of priority fields to the first tree layer, whose nodes select the minimum in each component position. This process is repeated throughout the entire tree, so that the root node contains the minimum value of each priority field. In practice, the tree nodes contain pointers to the LPs with minimum values, rather than the minimum values themselves. This is illustrated in figure 3(b), for the case of one priority field. With the extension to multiple priority fields, each tree node holds a vector of pointers to LPs; for each priority field, the LP with minimum value in that field is found by following that field's pointer in the tree's root.

Operations on the tree are straightforward. Whenever an LP changes its priority values, all interior nodes on the path between the LP and root must be re-evaluated. This operation has a linear cost in the depth of the tree. To insert an LP into the tree structure one simply finds a free leaf, and attaches the LP there. All nodes on the path from the LP to the root must then be re-evaluated. Likewise, when an LP is removed from the tree, all nodes on the path from the LP's leaf to the root need to be re-evaluated. Of course, it is also possible to selectively update the pointers associated with a subset of the the priority fields.

The number of LPs attached to the tree dynamically grows and shrinks. We want to keep the tree as small as possible, to minimize the depth. On the other hand, excessive tree re-organization should be avoided, as it involves reallocating all the leaves, and re-evaluating all the interior nodes of the tree. We adopt a policy which grows the tree to depth n+1 when we need to insert the 2^{n+1}+1st LP. The tree is shrunk to depth n-1 if the tree depth is n and the number of nodes falls below $2^n - 2^{n-2}$. The policy's philosophy is to grow the tree only when absolutely necessary, and to shrink the tree at sizes ``between'' the grow points---this to avoid re-organization thrashing.

Multi-order priority-queues have attractive time complexities. Changing an arbitrary LP's priority, or adding an LP exacts a constant expected cost, if we can assume that its priority value has only half a chance of defining the minimum value at each of the tree stages between the LP's leaf and the root. Changing the priority of the LP with least cost, or deleting an LP has logarithmic cost in the size of the queue. The cost of reorganizing the tree can be amortized over all queue operations which occur between two reorganizations, yielding at most a constant "amortized" reorganization cost each queue operation.

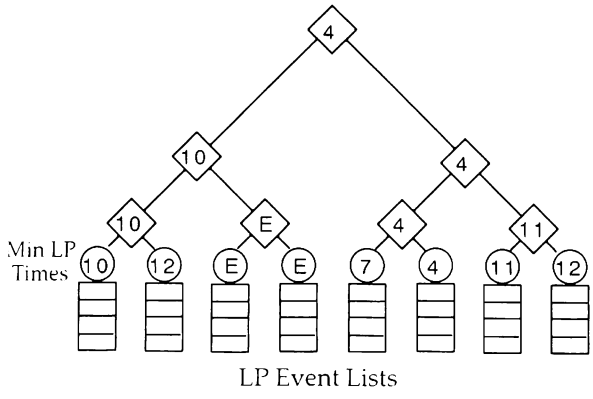

Figure 3(a): Min-Tree over LP Event Times

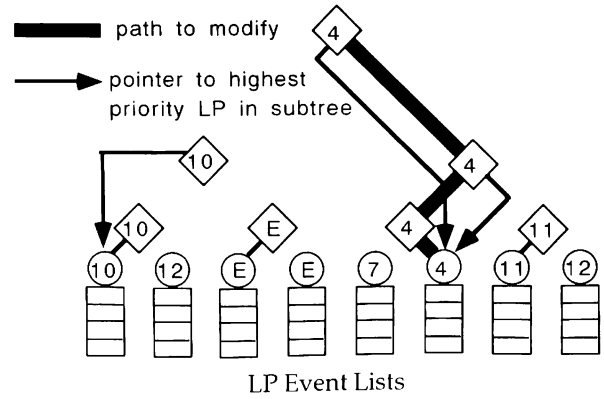

Figure 3(b): Min LPs found with pointers

7. SUMMARY

We have described YAWNS, a parallel simulator designed to allow rapid development of various large-scale simulation models used in parallel simulation research. YAWNS provides mechanisms for aggregating many LPs onto a single processor, for synchronizing those LPs and supporting their communications, and for controling the execution of events. YAWNS is built around a new synchronous synchronization protocol, and clearly defines an interface with the LP processing routines. YAWNS demands that the LP programmer provide it with a small amount of information which is often available. YAWNS is presently being implemented on an Intel iPSC/2, and will serve as a basic testbed for our experiments in parallel simulation on that machine.

Acknowledgements

This work was supported in part by the Army Avionics Research and Development Activity through NASA Grant NAG-1-787, and by CIT grant INF-89-014.

References

[1] Abrams, M. A common interface for Byrant-Chandy-Misra, Time-Warp, and sequential simulators. In *Proceedings of the 1989 Winter Simulation Conference*, Washington, D.C., 1989.

[2] Ayani, R. A parallel simulation scheme based on distances between objects. In *Distributed Simulation 1989*, SCS Simulation Series, 1989, pp. 113-118.

[3] Chandy, K.M., and Sherman, R. The conditional event approach to distributed simulation. In *Distributed Simulation 1989*, SCS Simulation Series, 1989, pp. 113-118.

[4] Fox, G. Johnson, M., Lyzenga, G. Otto, S., Salmon, J., and Walker, D. *Solving Problems on Concurrent Computers.* Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[5] Lubachevsky, B. Efficient distributed event-driven simulation of multiple-loop networks. CACM 32, 1(1989), pp. 111-123.

[6] Nicol, D. The cost of conservative synchronization in parallel discrete-event simulations. Submitted for publication.

[7] Reynolds, P.F. Jr. Comparative analyses of parallel simulation protocols. In *Proceedings of the 1989 Winter Simulation Conference*, Washington, D.C., 1989.

[8] Wagner, D. *Conservative Parallel Simulation: Principles and Practice.* Ph.D. thesis, University of Washington, 1989.