Proceedings of the 1991 Winter Simulation Conference
Barry L. Nelson, W. David Kelton, Gordon M. Clark (eds.)

PARALLEL SIMULATION OF TIMED PETRI-NETS

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

ABSTRACT

This paper considers the problem of using a parallel computer to execute discrete-event simulations of timed Petri-nets. We first develop synchronization and simulation algorithms for this task, and discuss a parallelized Petri-net simulator which has been implemented on an Intel iPSC/2 distributed memory multiprocessor. Next we describe a graphics-based front-end for the simulator, used to build timed Petri-net models. Finally, we discuss empirical studies of the simulator's performance on a variety of timed Petri-net models.

1 INTRODUCTION

Timed Petri-nets (TPNs) are an important modeling tool used to study the behavior of computer designs, communication networks, and other complex systems. While a great deal of study has gone into the analytic properties of TPNs (e.g., see [Murata, 1989] and its references), in practical settings TPNs are generally simulated. For example, TPNs are the basis for the performance analysis in the ADAS modeling tool [Frank *et al.*, 1985]. Discrete-event simulation of TPNs is thus an important modeling and analysis activity, and is one known to require a great deal of computational effort. Parallel architectures offer the possibility of decreasing the execution time of TPN simulations; the application of such architectures to discrete-event TPN simulations is the focus of this paper.

Parallelized discrete-event simulation has been actively studied over the last ten years; the survey in [Fujimoto, 1990] is an excellent introduction to the topic, and highlights current areas of research interest. Synchronization between processors has been and remains a subject of much interest, owing to the complexity of the synchronization requirements imposed by discrete-event simulations. The difficulty arises from the fact that the simulation model is typically partitioned among processors, each of which maintains its own simulation clock. An event associated with the submodel assigned to one processor may affect some portion of a submodel assigned to another processor, thereby necessitating an interprocessor communication. A parallel discrete-event simulation can be viewed then as a collection of communicating discrete-event simulations of submodels. In this context the notion of simulation time imposes synchronization requirements. Consider: an event (e.g. a job departure in a queueing network) completes at simulation time s on some processor, and affects the submodel on another processor (e.g. the job is routed to a queue on a different processor). If the affected processor has already simulated up to time $t > s$, it may have done so incorrectly as it has neglected to consider the effect of the event at time s. Synchronization protocols deal with this problem. Two fundamentally different styles of protocols have been studied. *Conservative* approaches (e.g. [Chandy and Misra, 1979, Lubachevsky, 1989, Nicol, 1988]) ensure that a processor does not advance its simulation clock until it is certain that it will not bypass some simulation time at which another processor affects it. Conservative protocols are known to require *lookahead* in order to avoid deadlock, and to achieve good performance. Lookahead is the ability of a processor to predict its future behavior, as regards when next (in simulation time) it may affect another processor's submodel. *Optimistic* approaches ([Jefferson, 1985]) permit a processor to simulate ahead under the anticipation that another processor will not affect its submodel in the "past", but then correct these temporal errors as they occur. Optimistic approaches require state-saving and rollback to function properly.

The approach we develop in this paper is conservative. A principal contribution of this paper is to show how to find and exploit lookahead in TPN models.

The parallelized simulation of TPNs has not received much attention. This is due in part to the fact the conceptual model of parallel simulations that is usually studied (based on the seminal work in [Chandy and Misra, 1979]) precludes Petri-net semantics, an observation detailed by [Thomas and Zahorjan, 1991]. This conceptual model (originally intended to support formal verification, not protocol design) ascribes fixed communication channels between *logical processes*; time-stamped messages are exchanged via these channels, and an LP's simulation clock is advanced as a result of consuming a message. The solution described in [Thomas and Zahorjan, 1991] involves extension of this model to support Petri-net semantics. The conceptual model we have used in recent years [Nicol, 1988, Nicol, 1990, Nicol, 1991] is simply that of communicating discrete-event simulations. Our model employs the same semantics of event list manipulation as does traditional serial discrete-event simulation, and so does not suffer from the limitations of the message-consuming model. Our treatment of TPNs differs from our treatment of a number of other simulation models only in how we compute lookahead. Lookahead is necessarily application specific.

The remainder of this paper is organized as follows. In Section §2 we discuss how TPNs work. Section §3 develops our synchronization and simulation algorithms. Section §4 describes, our X-Windows based tool **pntool** for developing TPNs to be simulated in parallel, and Section §5 presents the performance results of models developed under **pntool** which are simulated using the algorithms described in this paper. Section §6 summarizes this paper.

2 BACKGROUND

A Petri-net can be viewed as a bipartite graph, with each graph node classified as either a *place*, or a *transition*. The usual graphical conventions depict a place by a circle, and a transition by a straight line. Places may direct arcs to transitions, and transitions may direct arcs to places. Each place that directs an arc to a transition t is known as one of t's *input places*; likewise, each place to which t directs an arc is known as one of t's *output places*. Input and output transitions are similarly defined with respect to a place. A place may hold any number of *tokens*; the tokens may move from place to place in accordance with the *transition firing* rule. A transition t may *fire* if each of its input places has at least one token each. The effect of t's firing is to remove one token from each of t's input places, and to add one token to each of its output places. These graphical conventions and the effect of a transition firing are depicted in Figure 1.

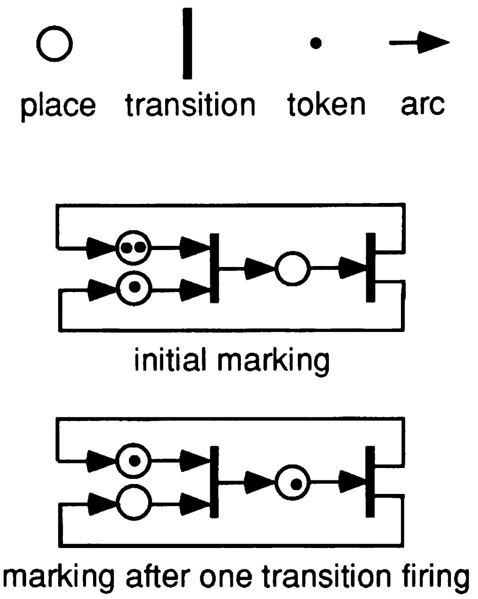


Figure 1: Graphical Petri-net Conventions

A place with more than one output transition is known as a *decision place*. The arrival of a token at a decision place may fulfill the firing requirements of more than one transition. However, only one of these transitions should actually be permitted to fire, as the firing of the first such will remove the enabling token from the decision place. A standard means of resolving this dilemma is to non-deterministically choose which transition (among those able to fire) will actually fire.

An ordinary Petri-net has no notion of "time". The most common variant of timed Petri-nets associates time with transition firings, as follows. Suppose the conditions to fire a transition are met at time s, and the *firing time* associated with that transition is δ. Then

- At time s, one token is removed from each of t's input places;
- From time s to time $s + \delta$ the transition is considered to be *firing*;
- At time $s + \delta$ a token is added to each of t's output places.

We say that the transition firing is *enabled* at time s, and *completes* at time $s + \delta$. A firing transition cannot be enabled (again) until the firing completes.

Using the rules above, we may construct a discrete-event simulation of a TPN whose events are **TokenArrival**, **BeginFire**, and **EndFire**, which denote the arrival of a token to a place, the beginning of a transition's firing, the removal of a token from a place, and the ending of a transition's firing, respectively. Assuming that the initial marking of tokens to places appropriately initializes the event list with **TokenArrival** events, the simulation may be implemented using the following sequence.

1. Fetch the next event from the event list, say with time T_{sim}. Advance the simulation clock to time T_{sim}.
2. Execute the event, in one of the following manners.

   Case: TokenArrival Let p denote the associated place. Increment the token count at p. If the previous token count was nonzero, then the event processing is finished. Otherwise, this token's arrival may enable the firing of some transition. In this case, among all of p's output transitions, identify those now enabled to fire due to the token's arrival. Choose one of these uniformly at random, say t, and insert a **BeginFire** event for t in the event list, with time-stamp T_{sim}.

   Case: BeginFire Let t denote the associated transition, and let δ_t denote its firing time. Decrement the token count at each of t's input places. For every one of t's output places p', insert a **TokenArrival** event with time-stamp $T_{sim} + \delta_t$ into the event list. Finally, insert an **EndFire** event with time-stamp $T_{sim} + \delta_t$ into the event list.

   Case: EndFire Let t denote the associated transition. If each of t's input places has at least one token, then refire the transition using the same logic as given for the **BeginFire** event.
3. Return to step 1 if termination conditions are not met.

Two features of this solution are noteworthy. First, it may seem curious that we insert **TokenArrival** events into the event list as a result of **BeginFire** processing, instead of **EndFire** processing. We deliberately formulated the solution this way in order to highlight the lookahead that exists in TPN simulations—at the time a transition begins its firing we can predict exactly when tokens generated by the firing appear in their new places. Our parallel solution will exploit this fact. Lookahead of this type is not necessary in purely serial simulations. Secondly, the correctness of the simulation depends on a two-level priority structure among events. Simulation time is, of course, the primary key. The event type serves as a secondary key; the sequence **TokenArrival**,**BeginFire**, and **EndFire** defines the event orderings, by decreasing priority. Without this priority ordering, a **TokenArrival** event could schedule a **BeginFire** event for transition t' at the same time, but an intervening **EndFire** event for t' that shares a common input place with t can refire t', and "steal" one of the tokens that enable t to fire. Of course, other solutions to this problem exist, but this seems to us to be quite clean.

In the section to follow we show how to implement this algorithm on a parallel computer.

3 PARALLELIZATION

We anticipate that parallel simulation will be practical primarily when large simulation models are distributed over a moderate number of processors. The usual use of discrete-event simulations is to construct confidence intervals from simulation output. Confidence intervals call for independent replications, and there is scarcely any easier way to exploit parallelism than to concurrently run independent replications. However, one rarely wants to run more than, say, twenty replications, because the width of a confidence interval decreases only in proportion to the inverse square root of the number of replications. The implication is that given a 500 node multiprocessor, one is more likely to devote 25 processors to each of 20 independent replications than one is to devote an independent replication to each processor. Thus we believe that techniques for medium-scale parallelism have practical interest. We also believe that parallel simulation will be useful primarily on large simulation models. Small simulation models are simulated sufficiently quickly on workstations or PCs. Parallel architectures offer increased main memory size over conventional architectures, which helps to avoid running the simulation in virtual memory and its attendant paging costs.

For the reasons outlined above we have concentrated on simulation techniques suitable for large simulation models on medium-grained multiprocessors. We have studied a conservative synchronous approach to synchronization and demonstrated analytically that it can achieve good performance when the size of the simulation model is large [Nicol, 1990, Nicol, 1991]. The approach we now develop for TPN simulations is an application of this approach to the TPN problem.

The remainder of this section is divided into three parts. The first part provides some general information needed to understand the parallelization. The second part discusses the parallelization itself, while the third part illustrates the method with an example.

3.1 Preliminaries

We assume that a TPN model is partitioned by first assigning each place and each transition to a *Logical Process*, or simply LP. An LP is usually composed of a set of places and transitions that are logically related; examples of this will be shown in Section §5. LPs are then mapped to processors. We assume that every LP maintains its own event list. For the sake of simplicity, our solution assumes that every output transition of a decision place is assigned to the same LP as the decision place, and that every input place for a transition t is assigned to the same LP as t. These restrictions serve to simplify the synchronization mechanism. They ensure that a single LP naturally has available all the information it needs to resolve conflicts over token consumption. For example, consider a token arrival to a place p, which is an input place for some transition t. The **TokenArrival** event processing described earlier needs access to the token counts in all of t's input places, in order to determine which of p's output transitions may begin firing, if any. Our restrictions ensure that all these places reside on the same LP (and hence processor) as p; the decision to begin firing a transition is based entirely upon on-processor information. Without these restrictions, the synchronization and communication needed to implement the simulation properly is rather more complex, as demonstrated by [Thomas and Zahorjan, 1991].

Automated techniques for LP construction and mapping are topics of interest to us, but are beyond the scope of the present paper.

In our framework, one LP sends another LP a time-stamped **TokenArrival** message whenever a firing transition places a token in an off-LP place. The event processing logic on every processor is identical to the event processing described in Section §2, save that the code must detect when to send a **TokenArrival** message rather than insert a **TokenArrival** event in an event list. Our synchronization protocol establishes control over these inter-LP message communications. The protocol relies on two key activities: the *pre-sending* of **TokenArrival** messages, and the computation of lower bounds on the time-stamp of the next message an LP might send. We have already introduced the notion of pre-sending **TokenArrival** messages—these messages are sent as part of **BeginFire** event processing, rather than **EndFire** event processing.

Given that **TokenArrival** messages are pre-sent, one can, at any time, compute a lower bound on the time-stamp of the next message an LP may send. For any LP, consider the set of *border* transitions, those transitions assigned to it which have output places assigned to a different LP. Let δ_{min} be the minimum firing time among all the LP's border transitions, and suppose that T_{sim} is the value of the LP's simulation clock after completing some event's processing (the lookahead we discuss here is computed between the processing of events, not during). The next message the LP sends cannot have a time-stamp smaller than $T_{sim} + \delta_{min}$, for only **BeginFire** events send messages, and the time-stamps on these messages are constructed by adding the LP's clock value to the transition's firing time. Thus, $T_{sim}+\delta_{min}$ always provides the desired upper bound. A potentially larger *conditional* bound can be constructed with very little extra cost by replacing T_{sim} with the least time-stamp on any event in the event list, say E_{min}. We take $E_{min} = \infty$ if the list is empty. The validity of this bound is conditioned on the LP not receiving a **TokenArrival** message with a time-stamp smaller than E_{min}. It turns out that bounds conditioned on the absence of further message arrivals suffice for our protocol. Our parallel solution assumes the existence of a routine **BoundNextMsgTime()** which finds E_{min} and returns the sum $E_{min}+\delta_{min}$. We turn next to a discussion of the protocol and its integration into the simulation algorithm.

3.2 Parallel Algorithm

The following is a brief overview of the protocol. Suppose that all simulation events in all LPs up to (but not including) time T_{sim} have been simulated. Our protocol will compute a simulation time $w(T_{sim})$, such that all events with time-stamps in the *window* $[T_{sim}, w(T_{sim}))$ can be executed without further communication between LPs. Messages generated in the course of processing **BeginFire** events are delivered at the end of the window processing; upon receipt a message is converted into a **TokenArrival** event and is inserted into the recipient's event-list. Once all events up to time $w(T_{sim})$ are known to have been simulated, a new window is computed, and the process repeats.

From the description above it is clear that the time $w(T_{sim})$ must be chosen carefully. Given that all LPs have simulated all events up to time T_{sim}, $w(T_{sim})$ is computed by having each LP call **BoundNextMsgTime()**. $w(T_{sim})$ is defined to be the minimum conditional bound returned, among all LPs. The global minimum computation can performed in $O(N/P + \log P)$ time on most multiprocessors, where N is the number of LPs, P is the number of processors, and we assume that each processor is assigned N/P LPs. We have proven elsewhere [Nicol, 1990] that every message the simulation will send after this point has a time-stamp of at least $w(T_{sim})$. Thus, all events that the simulation will generate in the interval $[T_{sim}, w(T_{sim}))$ have already been identified and reside in the appropriate event lists. Every LP may therefore simulate its submodel up to (but not including) time $w(T_{sim})$ without danger of receiving a "late" message. This property leads us to the protocol given below.

1. For every initial token, insert a **TokenArrival** event in the appropriate LP's event queue, with time-stamp 0.

2. Set $s_1 = 0$, set $i = 1$.

3. For every LP, call **BoundNextMsgTime()**. Use a logarithmic time min-reduction to compute $w(s_i)$—the minimum value returned by any **BoundNextMsgTime()** call. We may assume that every LP learns the value of $w(s_i)$.

4. Every LP may now simulate its submodel up to time $w(s_i)$, independently of and in parallel with all other LPs. Event processing is identical to that described in Section §2, save that **TokenArrival** events destined for off-LP places are passed as time-stamped messages.

5. The LPs synchronize globally. Following the synchronization, every LP may receive all messages sent to it during the processing of window $[s_i, w(s_i))$. An LP processes a received message simply by inserting the described event into the LP's event list.

6. Define $s_{i+1} = w(s_i)$, then increment i. Check termination conditions, return to step 3 if the termination conditions are not satisfied.

The reason why this protocol will work well on large models is quite intuitive. Imagine the simulation time line, and mark it wherever an event occurs. This protocol slides a window across the time-line, allowing LPs to be simulated in parallel during the span of a window. Now suppose that a transition's firing time is always at least δ_{min} time units long. Then the window width will always be at least δ_{min}. As we increase the size of the simulation model, the density of events on the simulation time line will increase, and so the number of events within a window will increase. The overhead of the protocol lies only in determining the size of the window; hence, as the model size grows the protocol's overhead is amortized over an increasing number of events. Of course, one still strives to keep the overhead low, but it is reassuring to know that regardless of its cost, it can be spread over the processing of many, many events on large models.

3.3 Example

The protocol's mechanics are more easily understood by way of an example. Figure 2 illustrates a TPN partitioned into three LPs. LP_1 models a producer of data. It has one place, *BufferFree* (BF) which holds a token when a buffer associated with LP_2 is free. The transition firing models the process of producing a data item, and is assumed to require 1 simulation time unit. LP_2 models a data processor. It has two places, *DataReady* (DR) and *BufferFree* (BF). The DR place holds a token following the production of a data item by LP_1; the BF place holds a token when a buffer associated with the third LP is empty. LP_2's transition is enabled once a data item has been produced by LP_1, and LP_3's buffer is ready to receive the results of LP_2's processing. We assume this transition's firing time is also 1 simulation time unit. Completion of this transition's firing serves both to provide a processed data item for LP_3, and to free a buffer for LP_1's output. Finally, LP_3 models a data consumer. It has a single place, *DataReady*, which holds a token after LP_2 makes a processed data item available. The single transition models the consumption of the data item, assumed to require 1 simulation time unit. Completion of the consumption activity frees a buffer for LP_2's output.

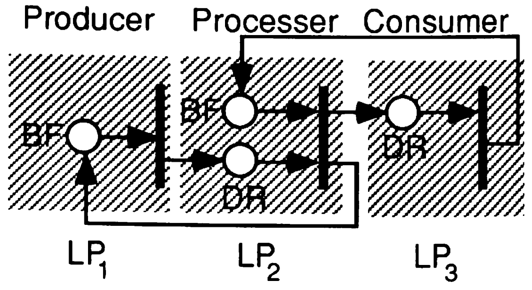


Figure 2: Three LP Petri-net Model of a Producer/Processor/Consumer System

Consider an initial marking that places a token in LP_1's place, and another in LP_3's DR place. LP_1 is thus free to produce a data item, and LP_3 is free to consume one. We may assume then that there is a **TokenArrival** event in LP_1's event list, and a **TokenArrival** event in LP_3's list, both with time-stamp 0. LP_2's list is empty. According to the protocol, $s_1 = 0$, and we compute $w(s_1)$ by having each LP call **BoundNextMsgTime()**. This function returns value 1 for both the LP_1 and LP_3 calls (since $E_{min} = 0$ and $\delta_{min} = 1$); the LP_2 call returns ∞ since LP_2's event list is empty. $w(s_1) = 1$, and all LPs are now free to simulate all events up to, but not including, time 1. LP_2 is idle during this window's processing. LP_1 processes its **TokenArrival** event, which generates a **BeginFire** event at time 0. The **BeginFire** event is processed, and as a result a **TokenArrival** message for LP_2's DR place is generated, with time-stamp 1. An **EndFire** event at time 1 is inserted into LP_1's event list. LP_3's processing is similar; as a result of the processing a **TokenArrival** message for LP_2's BF place is generated with time-stamp 1, and an **EndFire** event at time 1 is inserted into LP_3's event list. After processing all events in the window, the LPs receive the messages generated during the window processing. LP_2 is the only recipient; it transforms its two **TokenArrival** messages into **TokenArrival** events, both with time-stamp 1. s_2 is defined to be time 1, and the protocol is again engaged. Every LP has an event with time-stamp 1 in its event list, hence every LP's call to **BoundNextMsgTime()** returns the value 2. $w(s_2)$ is thus 2, and the LPs are free to simulate up to time 2. LP_1 processes its **EndFire** event at time 1; the transition's firing is not re-enabled, so no messages are generated as a result of this processing. The processing of LP_3's **EndFire** event also fails to generate messages. The processing for LP_2 is more interesting. LP_2 executes its first **TokenArrival** event, say, for the BF place. The processing of this event notes that the transition is not enabled to be fired, as there is no token in the DR place. However, a **TokenArrival** event for the DR place is processed immediately afterward, enabling LP_2's transition to fire. A **BeginFire** event at time 2 is scheduled, and then executed. Processing of this latter event causes **TokenArrival** messages with time-stamp 3 to be sent to LP_1's BF place, and to LP_3's DR place. At the end of the window's event processing LP_1 and LP_3 transform their respective **TokenArrival** messages into **TokenArrival** events, which are inserted into their event lists. The lower edge of the next window is defined to be 2, and the process repeats.

The first two windows in the sequence just described are illustrated by Figure 3. Each window summary lists the contents of each LP's event at the point the $w(s_i)$ computation is engaged, lists the lookahead value returned by **BoundNextMsgTime()** to each LP, lists the events executed by each LP after the window's edge is computed, and lists the messages sent by each LP as a result of this processing.

4 PNTOOL

One of the goals of our efforts is to develop algorithms for automatically clustering places and transitions into LPs, and for automatically assigning LPs to processors. We believe this study should be conducted on large, realistic Petri-nets, and recognize that the construction of text files describing such nets is tedious and error-prone. Consequently we developed **pntool**, an X-Windows based graphical tool for rapidly developing large Petri-net models. In this section we describe some of the more important features of **pntool**.

pntool allows a user to graphically create and edit a timed Petri-net. Panel buttons permit the creation/deletion of places, transitions, and arcs; these are graphically placed using the cursor, as are initial markings of tokens to places. Every transition and place is labeled automatically with an integer id; transitions are also labeled with their firing times, which again can be edited.

An important feature of **pntool** is its ability to *iconify* a net design and duplicate it. Many large petri-nets exhibit a great of replication. **pntool** exploits this by permitting one to design a portion of the net and iconify it. Iconification makes the design available for rapid inclusion into larger models. A **pntool** icon with the subnet name is created to represent the subnet. At any time a user may select an iconified design, and cause a duplicate of the network represented by the icon to be placed in the net design window. The user then must "connect" the subnet with places and transitions already resident in the design window.

pntool permits the construction of very large nets; however, only a portion of a net can be viewed at a time. **pntool** provides scollbars to position the user's view of the network.

Another important **pntool** feature allows the user to cluster places and transitions into LPs, and to assign LPs to processors. At present it is necessary to use these features if one wishes to simulate the developed net on our parallelized Petri-net simulator. LP clustering is accomplished using the cursor to create a polygon enclosing a region of the net design space; all

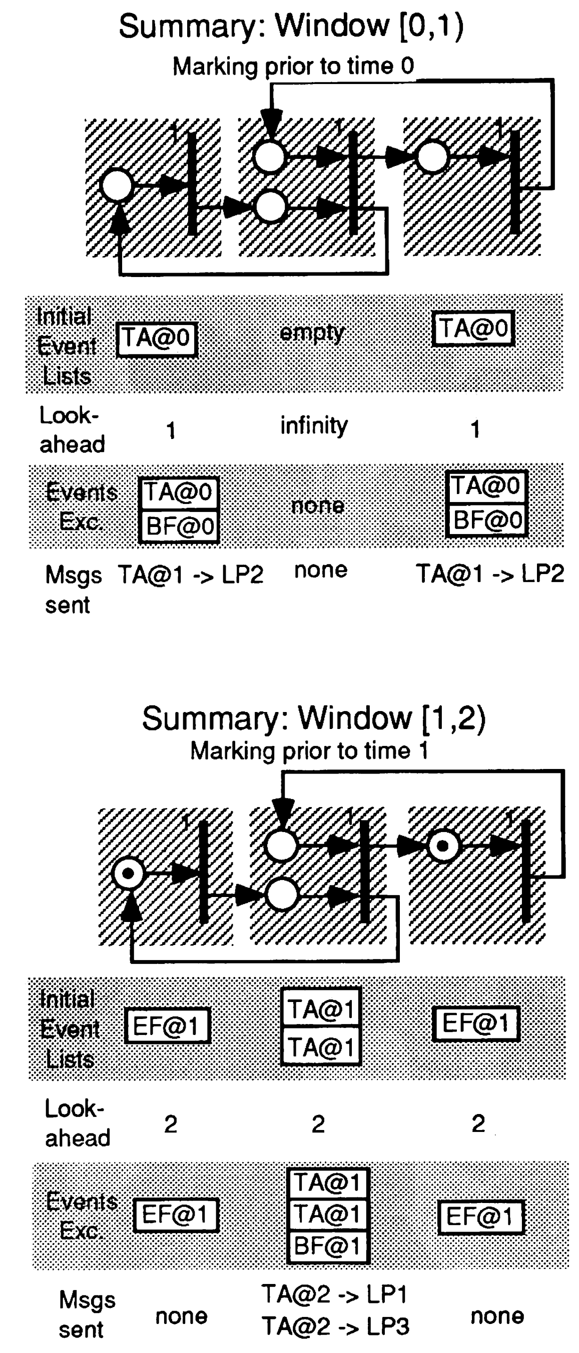
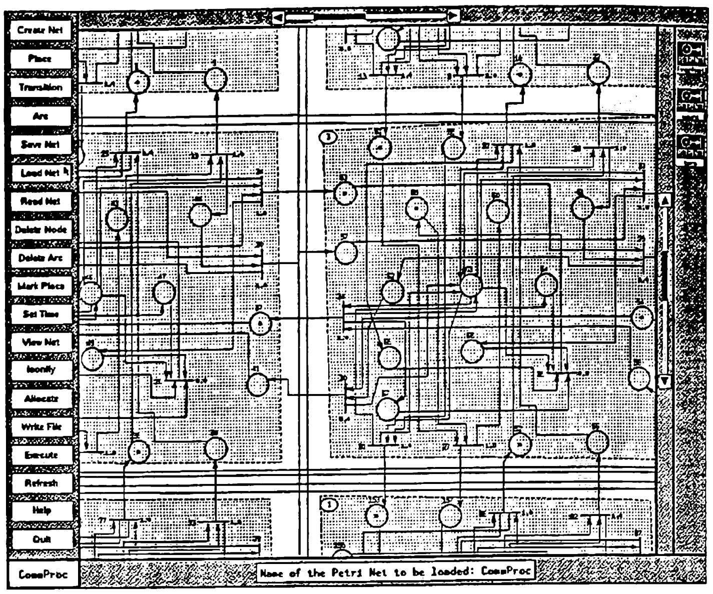


Figure 3: Example:Synchronization and Processing of Two Windows



Name of the Petri Net to be loaded: CommProc

Figure 4: Example of **pntool** Presentation Screen

transitions and places within the region are assigned to the same LP. The region one creates may be completely general. In theory one could cluster any set of transitions and places together into an LP; in practice it is much easier to cluster places and transitions that are contiguous in the net design window. The final step in clustering an LP is to assign the LP to a processor. An integer giving the processor's identity is then displayed in a corner of the LP. **pntool** permits one to edit this assignment.

Figure 4 illustrates an example of the **pntool** presentation screen. Icons representing iconified subnets are shown along the right side; the panel selection items are shown along the left side. The dark regions within the design window show how places and transitions have been clustered.

pntool produces a text file describing the net and its initial marking. This file serves as the input file to a parallel timed Petri-net simulator implemented on an Intel iPSC/2 distributed memory multiprocessor. Aside from this input file, all that is required of a user at run-time is a specification of the number of processors to use, and the length of simulation time one wishes for this run. The performance of the simulator on three **pntool** models is described next.

5 EMPIRICAL STUDIES

Three **pntool** models are the basis of our empirical studies. These models were developed under **pntool** and executed on a timed Petri-net simulation running

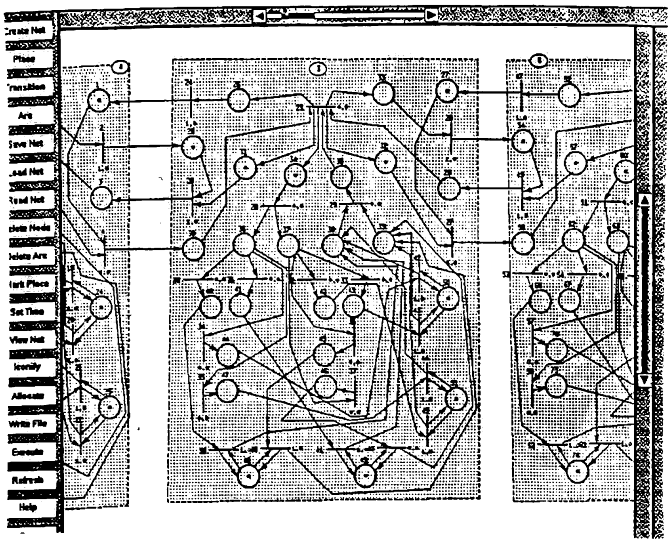


Figure 5: **pntool** representation of a *Model 2* LP

under the YAWNS (Yet Another Windowing Network Simulator) parallel simulation testbed [Nicol *et al.*, 1989]. These models are described briefly below.

Model 1 Each LP models a processing element in a mesh-connected parallel system. The element waits for four inputs (one from each of its four mesh neighbors), performs a computation, then sends the result to each of four mesh neighbors. The Petri-net ensures that no element sends another result to a neighbor until that neighbor has consumed the previously sent result. Each LP has 17 places, and 9 transitions.

Model 2 Each LP models a processing element, two memory controllers and four banks of memory; such ensembles are connected in a ring network. A processing element receives a data item from both of its ring neighbors, fetches two words from randomly chosen banks. The petri-net is constructed to handle memory contention in the event both words are chosen from the same bank. An LP has 26 places, and 23 transitions.

Model 3 Each LP models one stage in a feed-forward pipeline. A pipeline stage can execute when the results from the previous stage are available, and a buffer is free in the next stage. An LP has 4 places and 2 transitions.

A graphical description of a *Model 1* LP is found shown as the selected region in Figure 4. Figure 5 below describes one LP of a *Model 2* system. A *Model 3* LP is similar to LP$_2$ in Section §3.3.

These three basic models were simulated in configurations of 16,32,64, and 128 LPs. Each model was distributed evenly among 16 processors of an Intel iPSC/2 [Bomans and Roose, 1989]. Each model

	Model 1	*Model 2*	*Model 3*
16 LPs	0.26	0.40	0.28
32 LPs	0.40	0.52	0.36
64 LPs	0.52	0.66	0.48
128 LPs	0.65	0.77	0.63

Table 1: Measured Processor Utilizations (for 16 Processors) as Function of Model Type and Size

	Model 1	*Model 2*	*Model 3*
16 LPs	60	154	48
32 LPs	120	309	96
64 LPs	240	618	192
128 LPs	480	1237	384

Table 2: Average Total Events Processed Per Window, as Function of Model Type and Size

was executed for approximately 3 real-time minutes (within which a few millions of events are executed) and the performance measured. YAWNS summarizes performance via four distinct per-processor averages: the time spent sending/receiving inter-processor messages, the time spent in synchronization and window-edge computations, the time spent transforming LP messages into events, and the time spent executing events. Of these, the first two are considered to be overhead—costs which would not be suffered in a serial implementation. The last two measure activities which are part of serial implementations, event list manipulation and event processing. YAWNS computes *utilization* to be one minus the relative fraction of time spent in overhead activities, or equivalently, the fraction of time spent engaged in activity that a serial implementation would also perform. Sixteen times utilization thus provides a rough measure of "speedup", although speedup ought to be measured using an optimized serial implementation, which we have not done.

Table 5 gives the utilizations so measured, as a function of model type and size.

Table 2 gives the average number of events processed per window, also as a function of model and size.

These figures agree with the intuition that performance improves as the size of the simulation model increases. This is suggested both by scanning down a column, and by comparing the relative performance among models. A *Model 2* LP is significantly more complex than a *Model 3* LP, and can be expected to

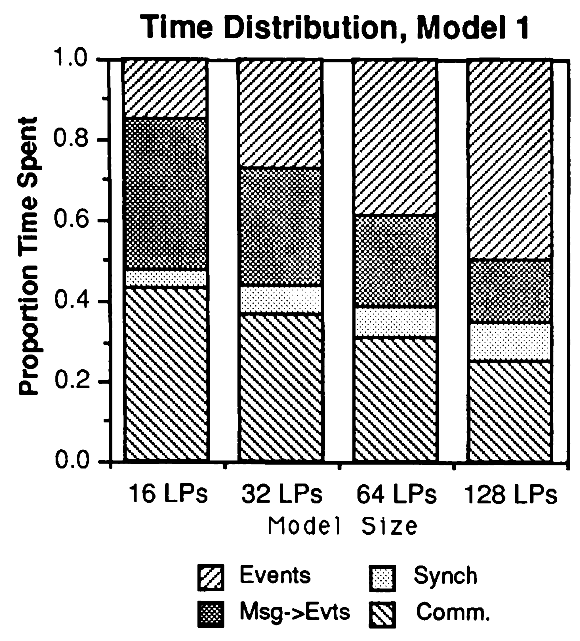


Figure 6: Distribution of Time Spent in YAWNS Activities

induce more simulation activity per window.

It is also interesting to note the proportions of time spent in each of the four activities YAWNS times. Figure 6 illustrates these proportions for *Model 1*, as a function of model size. It is apparent that as the model size increases, the proportion of time spent executing events increases. The proportion of time spent in message communication or in message-to-events conversion is decreasing, reflecting greater computation/communication ratios enjoyed under larger model sizes. The proportion of time spent in synchronization increases slightly, approximately as the log of the number of LPs.

The Intel iPSC/2 is a distributed memory multiprocessor. It is thus is not the optimal architecture for the parallel simulation algorithm, because of the relatively tight global synchronization the algorithm imposes. The significant proportion of time spent in inter-processor communication would be reduced dramatically if the messages were exchanged through a global memory. Nevertheless, despite the high cost of communication, our performance data shows that good performance gains are being achieved.

6 SUMMARY

This paper studies the problem of parallelizing the discrete-event simulation of large timed Petri-nets on medium-scale multiprocessors. We describe a synchronization algorithm, a graphical tool for designing TPNs to be simulated in parallel, and study the performance of the simulation on 16 processors of an Intel iPSC/2. Significant performance benefits are observed, with the promise of even better performance on shared-memory architectures.

ACKNOWLEDGEMENTS

This work was supported in part by NASA Grant NAG-1-060, the Army Avionics Research and Development Activity through NASA grant NAG-1-787, NASA Grant NAG-1-1132, and NSF Grant ASC-8819373.

References

[Bomans and Roose, 1989] L. Bomans and D. Roose. Benchmarking the iPSC/2 hypercube multiprocessor. *Concurrency: Practice and Experience*, 1(1):3–18, Sept. 1989.

[Chandy and Misra, 1979] K.M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. on Software Engineering*, 5(5):440–452, September 1979.

[Frank *et al.*, 1985] G.A. Frank, D.L. Franke, and W.F. Ingogly. An architecture design and assessment system. *VLSI Design*, pages 30–38, August 1985.

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

[Jefferson, 1985] D. R. Jefferson. Virtual time. *ACM Trans. on Programming Languages and Systems*, 7(3):404–425, 1985.

[Lubachevsky, 1989] B.D. Lubachevsky. Efficient distributed event-driven simulations of multiple-loop networks. *Communications of the ACM*, 32(1):111–123, 1989.

[Murata, 1989] T. Murata. Petri nets: Properties, analysis, and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

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

[Nicol, 1988] D.M. Nicol. Parallel discrete-event simulation of FCFS stochastic queueing networks. *SIGPLAN Notices*, 23(9):124–137, September 1988.

[Nicol, 1990] D.M. Nicol. The cost of conservative synchronization in parallel discrete-event simulations. *Journal of the ACM*, to appear. Available as Technical Report 90-20, ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23665.

[Nicol, 1991] D.M. Nicol. Performance bounds on parallel self-initiating discrete event simulations. *ACM Trans. on Modeling and Computer Simulation*, 1(1):24–50, 1991.

[Thomas and Zahorjan, 1991] G. Thomas and J. Zahorjan. Parallel simulation of performance petri nets: Extending the domain of parallel simulation. In *Proceedings of the 1991 Winter Simulation Conference*, Phoenix, Arizona, December 1991. To appear.

AUTHOR BIOGRAPHIES

DAVID M. NICOL received an B.A. in Mathematics from Carleton College in 1979, worked as a programmer analyst with the Control Data Corporation from 1979 to 1982, and received a Ph.D. in Computer Science from the University of Virginia in 1985. He spent the subsequent two years at the Institute for Computer Applications in Science and Engineering at the NASA Langley Research Center, and then joined the faculty of the College of William and Mary, where is currently an Assistant Professor. He is an associate editor for ACM Transaction on Modeling and Computer Simulation, and for the ORSA Journal on Computing; he has served as the 1989 Program Chairman and the 1990 General Chairman of the Workshop on Parallel and Distributed Simulation (PADS). His interests are in parallel simulation, performance analysis, and algorithms for mapping parallel workload.

SUBHAS ROY received the B.E. degree in Computer Science and Engineering from the Jadavpur University, India, in 1987. He received the M.S. degree in Computer Science from the College of William and Mary in 1990, and is presently pursuing a Ph.D. there on the topic of automated load-balancing of parallel simulations.