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

PARALLEL SIMULATION OF PERFORMANCE PETRI NETS: EXTENDING THE DOMAIN OF PARALLEL SIMULATION

Gregory S. Thomas
John Zahorjan

Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, WA 98195

ABSTRACT

We present a parallel simulation protocol for Performance Petri nets (PPNs), nets in which transition firings take randomly selected amounts of time. This protocol is interesting for two reasons. First, application of standard conservative or optimistic parallel simulation to PPNs results in either unnecessarily low (possibly no) parallelism or simply fails to produce correct results. Thus, this new protocol may be thought of as addressing a class of models not amenable to standard parallel simulation, with PPNs being a particular example. Second, PPNs are currently analyzed using numerical techniques that have time and space requirements exponential in the size of the net. Simulation, and particularly parallel simulation, is thus a practical alternate analysis method for these models, as we show by measurement of execution times.

We introduce a new technique called *Selective Receive* that loosens a fundamental rule of conservative parallel simulation by allowing model components to sometimes ignore certain of their input channels and so to determine their local clock times based on only a subset of their potential inputs.

Finally, we discuss a small modification to the definition of PPNs that, while not affecting their expressive power, allows for much more efficient simulations.

1 INTRODUCTION

Performance Petri nets (classical Petri nets augmented with the notion of time) are a powerful tool for the modeling of systems. Their application is limited, however, by the Markovian assumptions and exponential complexity of the analytic techniques typically employed in their analysis. For these models, simulation provides an attractive alternative: few, if any, restrictions need to be imposed and, as we will show, the execution time required to obtain model performance results can be much less than that required by the analytic techniques.

It is natural to consider the use of parallel simulation to reduce even further the elapsed time required to obtain performance measures from Performance Petri net (PPN) models. Using parallel simulation for this purpose, however, requires modification of the basic mechanisms on which it is structured.

Parallel simulation has been applied chiefly to systems in which the actions of a component (a *physical process*) are determined solely by its local state. Both the conservative [Misra 86] and optimistic [Jefferson 85] approaches to parallel simulation assume that interaction among components occurs exclusively via explicit messages.

When a physical system conforms to this paradigm of interaction, a parallel simulation can be built in a natural way by creating a set of *logical processes* (LPs) that mimics the behavior of the physical processes. The potential parallelism of the simulation scales with the problem size, since a larger physical system (i.e., one with more physical processes) maps to a simulation with a larger number of logical processes.

In contrast, as explained in Section 2.2, the dynamic behavior of each component of a PPN depends on both internal and external (possibly global) state. This reliance on external state prevents the standard paradigm of parallel simulation from being applied in a useful way.

The first goal of this paper, therefore, is to present a parallel simulation protocol that is both correct and provides parallelism that scales with the size of the PPN model itself. A second goal is to determine the amenability of PPNs to parallel simulation by implementing our protocol and measuring its performance. A final aspect of this problem, a comparison of the computational complexity of conservative parallel simulation of PPNs to that of analytic tech-

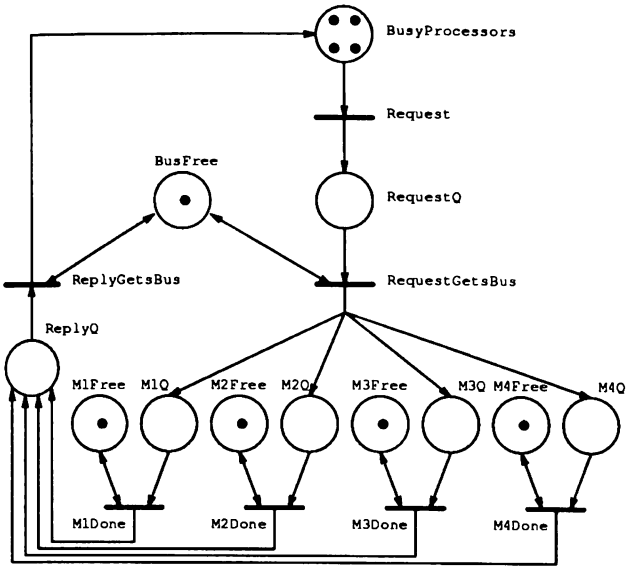

Figure 1: An Example PPN Multiprocessor Model

niques, is also addressed through measurement, using GTPNA [Holliday & Vernon 86] (an analytic PPN package) as the basis of comparison.

2 PERFORMANCE PETRI NETS AND PARALLEL SIMULATION

In this section we briefly introduce PPNs, discuss various possible decompositions of nets for the purpose of parallel simulation, and explain why the standard paradigm of parallel simulation does not work for the most promising decompositions. For more extensive introductions to PPNs, [Ajmone Marsan & Chiola 87, Murata 89] are recommended.

2.1 Performance Petri Nets

Figure 1 gives a graphical representation of a PPN model of a bus-based multiprocessor computer system consisting of four processors and four memory modules connected by a single bus.

A PPN is a bipartite directed graph in which the two types of nodes are *places*, drawn as circles, and *transitions*, drawn as bars.

If there is an arc (p, t) from a place p to a transition t, we say that p is an *input place* of t. Similarly, if there is an arc (t, p) from t to p, we say that p is an *output place* of t. (The analogous definitions hold for *input transition* and *output transition*.) A place that is an input place for multiple transitions is called a *decision place*.

At any point in time, each place in a PPN is marked with zero or more *tokens*, drawn as dots. The *marking* of a place is the number of tokens at the place, and the marking of a net is the vector of markings of its places.

A net's marking is affected by the firing of transitions. The firing of a transition t removes a token from each of t's input places and, after a delay chosen from the firing time distribution of the transition, deposits a token in each of t's output places.

A transition can begin to fire (i.e., consume tokens from its input places) only when it is *enabled*. A transition is enabled when each of its input places contains at least one token. An enabled transition may begin to fire even if it is already in the process of firing (due to earlier enablings).

Let P be the set of places of a PPN, $p_i \in P$, $\mathrm{OT}(p_i)$ the set of output transitions of p_i, and R^* the reflexive transitive closure of the relation $R \subseteq P^2$, defined by $p_i R p_j$ if and only if $\mathrm{OT}(p_i) \cap \mathrm{OT}(p_j) \neq \emptyset$. R^* partitions P into a set of equivalence classes (called *locksets* in [Taubner 88]). The places in such an equivalence class, along with their output transitions, constitute a *static conflict set*.

A *dynamic conflict set* is a set of transitions (and their associated input places) that are actually enabled and competing for tokens (at decision places) at some point in time. A dynamic conflict set is thus a subset of some static conflict set. When a dynamic conflict set exists, a subset of its transitions must be chosen to fire. This is called *conflict resolution*.

Under our conflict resolution scheme, explained in Section 5, the definition of the PPN is extended to include weights on arcs from places to transitions. Other extensions to PPNs have been made to increase their modeling capability or their ease of use: arc multiplicity, deposit branching, and generalized inhibitor arcs. These are described in [Thomas 91].

2.2 Parallel Simulation and PPNs

To perform a parallel simulation, the system to be simulated—a PPN, in our case—is mapped to a network of LPs that communicate through unidirectional *channels* [Chandy & Misra 81].

There are a number of ways to map a PPN into a network of LPs. In choosing one, the basic tradeoff of parallel computing applies: increasing the number of LPs working on the problem can decrease the elapsed execution time (a benefit) but can also increase inter-LP communication overhead (a cost).

Perhaps the most natural PPN-to-LP mapping is an isomorphism: one LP per PPN node and one channel per PPN arc. This node-based decomposition has the potential to address very large problems running on massively parallel machines. However, neither the

standard conservative nor optimistic method of parallel simulation can be used for this decomposition.

In standard parallel simulation, interaction among physical processes occurs exclusively via explicit messages, with messages in the logical system corresponding to messages in the physical system [Chandy & Misra 81]. The node-based decomposition of PPNs violates this paradigm because the behavior of a node of the net may be determined by state that is global with respect to that node rather than through explicit message exchange. For example, this is always the case for decision places. This problem exists for optimistic as well as conservative parallel simulation.

An alternative mapping of a PPN into a network of LPs is one in which each LP represents a static conflict set of the net, with channels corresponding to arcs between conflict sets. This mapping has characteristics that are just the opposite of the node-based mapping discussed above. The potential parallelism of this decomposition is poor, as there is no direct relationship between the size of a PPN and the number of conflict sets it contains.

In the end, the most appropriate decomposition of a PPN into a network of LPs is probably some combination of these two approaches. With this understanding, in the remainder of this paper we concentrate on the node-based decomposition, since it presents new challenges for parallel simulation. We first consider the problem of simply correctly simulating a PPN using this decomposition. Subsequently, we examine its performance using a prototype implementation.

3 THE TRANSITION FIRING PROTOCOL

A protocol for the conservative parallel simulation of PPNs, which we call the *Transition Firing Protocol* (TFP), is presented in this section. We chose to follow a conservative approach because we expected its performance characteristics to be more stable (and often better) than those of an optimistic approach.

Recall from Section 2.2 our chosen decomposition: each place and each transition is represented by an LP, and each arc in the net is represented by a channel. We augment this simulation model by introducing additional channels: there is a channel from LP t_j to LP p_i if there is an arc from place p_i to transition t_j in the PPN. To distinguish these additional channels from the "ordinary" channels (along which tokens may flow) we refer to them as *control* channels.

The graphical representation of the simulation model is identical to that of the PPN, except that we include control channels (drawn as dashed arcs)

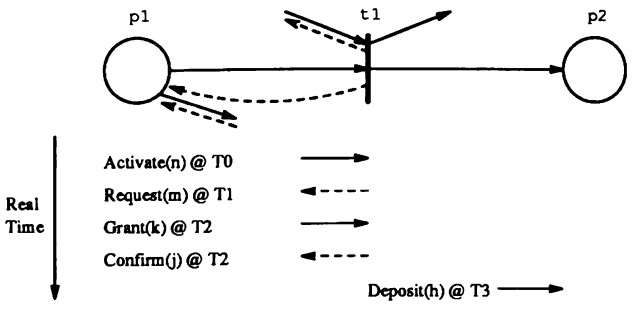

Figure 2: Overview of TFP

when appropriate.

TFP is basically a double handshake between transitions and their input places, as indicated in Figure 2. The notation *Message*(*n*)@T*i* means *Message*, passing parameter n, is sent at simulation time T*i*. T*i* is the *timestamp* of *Message*; every message has a timestamp. TFP's double handshake is motivated by decision places; a streamlined protocol can be adopted for certain simple net topologies.

In what follows, we describe the four steps in the handshake without explicitly considering how they are integrated with the messages corresponding to the token deposits due to transition firing. This integration, which is somewhat subtle, is provided in [Thomas 91]. For the sake of simplicity in the following discussion we assume regular (i.e., noninhibitor) arcs of multiplicity one. TFP handles the general case—regular and inhibitor arcs of any multiplicity.

- *Activate*(n)@T0

  The first handshake starts when place `p1` receives a deposit of n tokens at time T0, causing it to send an *Activate*(n)@T0 message to each of its output transitions, indicating that it has n tokens available at time T0.

- *Request*(m)@T1

  Once `t1` has received an *Activate* from each of its input places it calculates T1 $\geq$ T0, the latest timestamp of any of the *Activate* messages it has received, and $m \leq n$, the smallest of the *Activate* parameters. `t1` then updates its (local) simulation clock to T1 and responds to each input place with a *Request*(m)@T1 message, asserting that it is ready to fire m times at time T1 if all of its input places (still) contain sufficient tokens at that time.

- *Grant*(k)@T2

  `p1` waits to receive a *Request* from each output transition it has sent an *Activate*. (This is why

our approach is conservative.) Let T2 $\leq$ T1 be the earliest timestamp among these *Requests*. (In Figure 2, T2 = T1.) When `p1` has received all *Requests* timestamped T2, it executes a conflict resolution algorithm (explained in Section 5) to choose among competing earliest *Requests*, and then replies to each with a *Grant*(k)@T2 message, indicating that $k \leq m$ tokens are available to the sender of the *Request* for firing. (Note that k could be zero.)

- *Confirm*(j)@T2

  `t1` collects a *Grant* message from each of its input places. Let j be the minimum parameter of these *Grants*. (Thus, j could be zero.) `t1` ends the second handshake by sending a *Confirm*(j)@T2 to each input place, indicating that `t1` has fired j times.

  If appropriate, `t1` schedules one or more *Deposit*(h)@T3 messages, where $0 < h \leq j$ and T3 $\geq$ T2, to be sent (when the corresponding firing delays have elapsed) to one or more of its output places.

The protocol just described resembles in some ways several of the protocols developed by Taubner [Taubner 88]. That work was performed in the context of "Petri net driven execution" of distributed programs—the firing of a transition causes the invocation of a procedure, with the Petri net itself used to determine the flow of control (e.g., [Hartung 88])—rather than in the context of simulation of PPNs.

Our work differs significantly from Taubner's in at least three ways. First, Taubner assumes untimed nets, so there is no notion of simulation time. Second, our conflict resolution strategy is different from those proposed by Taubner. Finally, because each transition firing results in a procedure execution, the amount of overhead required to run the protocol is less important for execution of distributed programs than it is in our simulation context, where a transition firing involves very little inherent work. Thus, we have been more sensitive to these overheads in designing our protocol and prototype implementation.

4 SELECTIVE RECEIVE AND TFP

A basic tenet of the conservative approach to parallel simulation is that messages on each channel must be sent in nondecreasing timestamp order. It is this property that distinguishes conservative from optimistic simulation.

The standard conservative algorithm achieves this output ordering goal by imposing an input ordering restriction. Associated with a channel from LP_i to LP_j is a *channel clock value*, c_{ij}, which is equal to the earliest timestamp of any undelivered message pending on that channel, if one exists, or else the time of the last message sent on it. LP_j is prevented from receiving a message m having timestamp T_m unless T_m is equal to H_j, the *message acceptance horizon* of LP_j, where $H_j = \min_i c_{ij}$.

This input ordering restriction is a major source of performance problems for conservative simulations, including the possibility of deadlock. Consequently, a great deal of work has focused on minimizing its impact (e.g., [Reed et al. 88, Nicol 88, Wagner & Lazowska 89, Fujimoto 90]).

The problem of deadlock is particularly extreme for TFP. Because transitions send messages along their control channels only in response to messages received from the input place connected there, no messages can be received by places at the beginning of a TFP cycle.

Our solution to this problem is to change the rules governing the receipt of messages by LPs. In particular, we allow each LP to specify that certain channels should be ignored in computing that LP's message acceptance horizon, thus allowing the receipt of messages that would otherwise have to remain pending. We call this feature *Selective Receive*.

Selective Receive can be applied when an LP can deduce that the next message it will receive (in simulation timestamp order) cannot arrive on one or more of its input channels. Under TFP in particular, a place LP knows statically that any output transition not sent an *Activate* message by it will not send it any messages. Thus, the LP is free to receive the earliest message from the remaining channels.

Note that Selective Receive is distinct from approaches based on null messages, lookahead, and futures. Unlike null messages, Selective Receive does not involve the transmission of any extra control messages. Unlike lookahead and futures, Selective Receive can be used to avoid performance problems involving message channel loops even when the lower bound on the message propagation delay around the loop is zero. In fact, our use of Selective Receive in TFP serves exactly this purpose.

Under TFP, a place LP never ignores a channel from an input transition LP. It ignores a channel from output transition LP_j except between the time at which it sends an *Activate* to LP_j and the time at which it receives the corresponding *Confirm*. A transition LP ignores a channel from input place LP_i after it receives an *Activate* from LP_i until it sends *Requests*, and after it receives a *Grant* from LP_i until it sends *Confirms*.

5 CONFLICT RESOLUTION

When one or more decision places in a conflict set contain tokens, a decision must be made about which set of transitions to fire. While a number of proposals have been made for this decision procedure, there is no standard agreement on this aspect of PPNs. Our technique is modeled on a flexible approach offered by [Holliday & Vernon 87]. They allow the user to specify weights that are the basic parameters of a function that assigns probabilities to each *maximal set* (a set of transitions within a dynamic conflict set such that, if those transitions were to fire, no other transitions in the conflict set could also fire). A single maximal set is then chosen for firing according to this probability distribution. While Holliday and Vernon proposed this scheme for use in analytic approaches to PPN analysis, their technique is easily applied in a sequential simulation.

In a parallel simulation, however, applying this technique is much more complicated. The reason for this is that determining maximal sets requires a global view of the net at a particular simulation time, a difficult prospect in a parallel simulation. Further, because each LP has information about only its local state, no LP is naturally in a position to compute maximal sets. To do so requires the creation of a new LP for this purpose, and cooperation from place and transition LPs in registering state information with this LP. This is a complicated protocol to implement, and tends to serialize execution of the simulation.

When constructing a parallel simulation of PPNs, a decentralized approach is more natural. The one we use is based on "trial-and-error": each decision place selects at random according to the user-supplied arc weights one or more transitions that it would like to fire, and offers them tokens. If any transition is lucky enough to receive offers from all input places, it then fires. Otherwise, it replies that it cannot use any of the tokens and its input places try again.

This procedure, which is incorporated in the TFP policy described in Section 3, is completely decentralized: each place makes decisions based solely on information available to it either locally or through communication with its own output transitions only. Thus, we might expect the decentralized approach to have better performance than the more serial maximal set-based approach.

While this reduction in serialization may have important performance implications, experience with our simulator shows that an even more important effect is the change in computational complexity that results from the decentralized approach. Enumerating maximal sets is of exponential complexity [Holliday & Vernon 87], both in terms of time and space. Thus, if conflicts involving a large number of transitions or tokens occur with any frequency, conflict resolution by this approach can be extremely slow. On the other hand, the time requirement of the decentralized scheme depends on the arc weights. Since each decision place selects transitions independently, an unfortunate set of arc weights could lead to a very large expected time to determine a firing.

We note that the decentralized conflict resolution scheme has slightly different semantics than the maximal set-based approach, reflecting the difference between the "repeated trials" approach of the former and the one-time enumeration of the latter. In general, neither scheme is able to simulate the other, i.e., no choice of weights for the decentralized scheme results in a distribution identical to that of the maximal set scheme and vice versa. However, this is not considered a significant problem since there is no standard for conflict resolution semantics and neither scheme provides a significantly more convenient way for the user to express the desired behavior of the model.

We also note that while the decentralized scheme employed in TFP could be emulated in a sequential simulation, resulting in performance gains comparable to those we observed in the parallel simulations, this would be somewhat unnatural. It therefore seems unlikely that the new scheme would have been developed in a sequential environment.

6 IMPLEMENTATION AND PERFORMANCE

We have two goals in this section. The first is to examine how well a parallel simulation using a node-based decomposition of a PPN is able to exploit available processors. We address this by measuring the speedups obtained for simulations of a number of realistic PPNs. Our second goal is to evaluate the growth in the running time of our PPN simulation as the size of the PPN increases, and to compare these times with those obtained using analytic approaches to PPN evaluation. In this case, we take a single PPN model and increase its size in a natural way (as explained below), and evaluate these models using both analytic and simulation approaches.

6.1 Persephone

Persephone [Thomas 91] is a prototype implementation of TFP written in C++ [Stroustrup 86]. Persephone is built on top of a modified version of Synapse [Wagner 89], a library of C++ classes for conservative

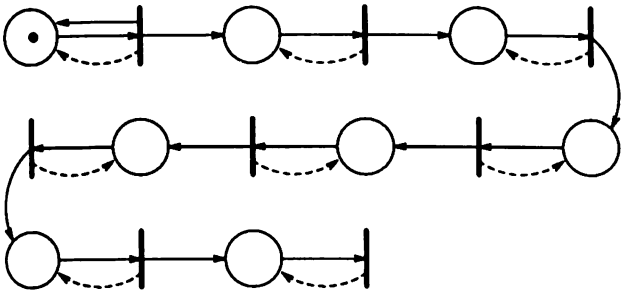

Figure 3: Instruction Pipeline Net

parallel simulation. Synapse runs on a Sequent Symmetry multiprocessor, and exploits the shared memory of this machine in an attempt to achieve good performance [Wagner & Lazowska 89].

The implementation of TFP required several modifications to Synapse. The primary changes involve its lookahead, deadlock detection and recovery, and message delivery mechanisms. Selective Receive—the major enhancement made to Synapse—violates the normal message delivery semantics of conservative simulation (and thus of Synapse), so many of the changes are directly related to Selective Receive.

Because Persephone is built on top of Synapse, and Synapse itself is built on top of yet another run time system (a user-level threads package named PRESTO [Bershad et al. 88]), determining where the bottlenecks are in a Persephone simulation is a significant challenge. The three systems together comprise 40,000 lines of C++ code.

6.2 Speedup Versus Number of Processors

In this section we examine the speedups achieved by Persephone on six distinct PPN models. Each of the PPN models represents an interesting computer system, and while certain aspects of these models have been given less attention here than would be appropriate if our purpose were in fact to model these computer systems, their basic structures could be used to answer performance questions about those systems. We first briefly present the six PPNs and then the set of speedups observed in their simulations. The initial markings of the PPN models are shown in their respective figures.

Figure 3 shows Pipe, our first PPN model. Pipe represents an instruction pipeline in a CPU. There is a token source, representing instruction issue, followed by a number of pipeline phases, each represented as a single place and transition pair. Control channels are shown explicitly, as dashed arcs, to emphasize the fact that the net is not truly feed-forward.

Our second application, called Multiprocessor, is

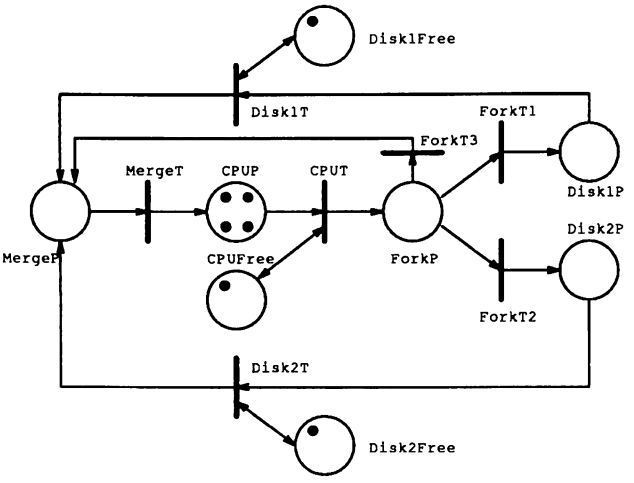

Figure 4: Central Server Net

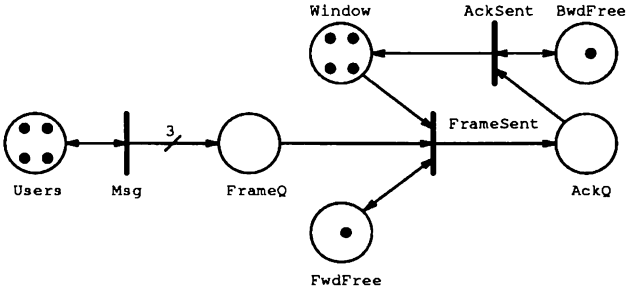

Figure 5: Communication Channel Net

the bus-based multiprocessor computer system introduced in Section 2.1 and appearing in Figure 1.

Our third application is the performance evaluation of a simple time-shared computer system. This is an application often addressed using queueing network models. We include it here because much of the work on parallel simulation techniques has used queueing models (and especially this one) as an application [Reed et al. 88, Nicol 88, Wagner & Lazowska 89, Greenberg & Lubachevsky 90] and because queueing models are easily represented by PPNs. In this model, named CentralServer and appearing in Figure 4, each token represents a job in a multiprogramming system, and circulates between the CPU and I/O devices acquiring service.

A window flow controlled communication channel is our fourth application. The PPN model for this application, called CommChannel, is shown in Figure 5. In this case, a source machine sends messages to a destination machine using a communication network. Each message is broken into a number of frames for transmission (in the model, each message creates three frames), and each frame is individually acknowledged. Packets are placed on the network one

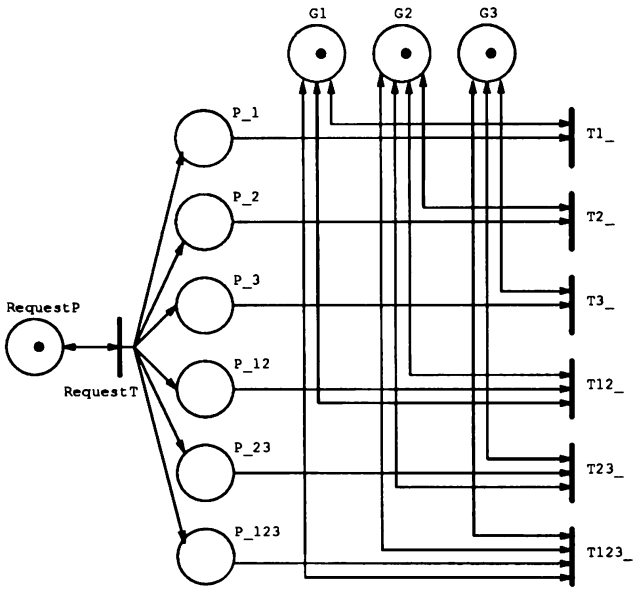

Figure 6: All-At-Once Database Locking Net

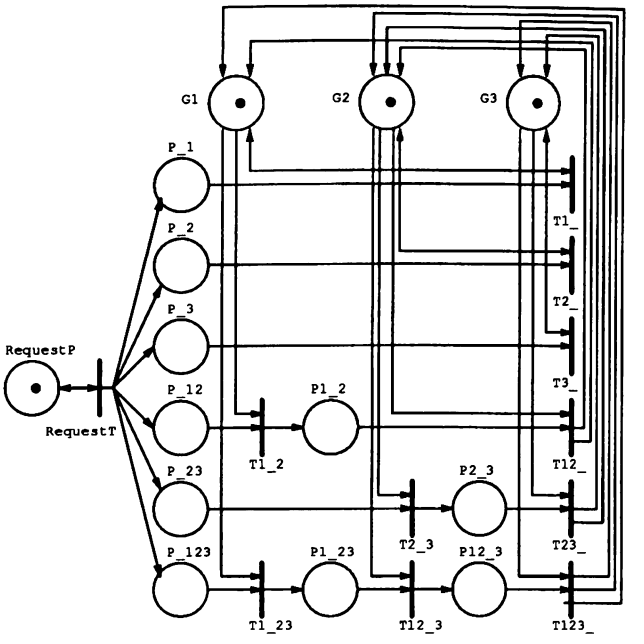

Figure 7: One-At-A-Time Database Locking Net

at a time, and for a window size W there can be up to W unacknowledged packets sent.

In our final application, which involves two distinct PPN models, we compare two schemes for acquiring multiple locks in a database system. We assume that the database consists of some number of lockable units, called granules, and that arriving transactions require one or more granules for execution. For simplicity, we assume a very simple and regular pattern to determine which granules an individual transaction will require. In particular, each transaction acquires a sequence of consecutively numbered granules. For a system with G granules, each arriving transaction acquires a sequence starting with granule g, $1 \leq g \leq G$, with probability $1/G$. A transaction requiring a granule r also requires granule $r + 1$ (if it exists) with probability 0.5.

The comparison we make is between a locking scheme in which an arriving transaction must wait until it can simultaneously acquire all the locks it will need (we call this scheme "All-At-Once") and a scheme in which each transaction acquires each lock it needs as it becomes available, but runs only once all locks are acquired (we call this scheme "One-At-A-Time"). For One-At-A-Time, to avoid deadlock the granules must be acquired by all transactions in a specific order.

The PPN models for these schemes are shown in Figures 6 and 7. Because the size of the PPNs grows quickly with the number of granules, the figures are for systems containing only three granules. Our experiments, however, are for systems containing

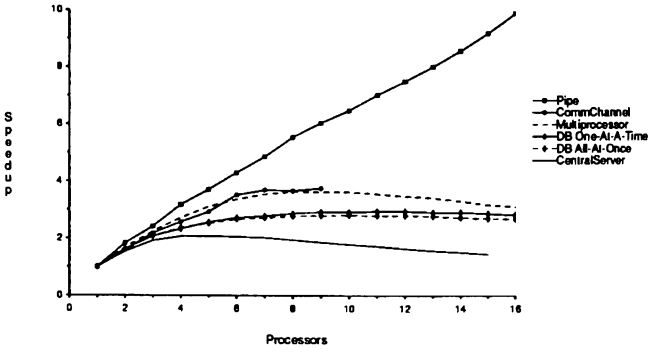

Figure 8: Measured Speedup of Persephone on the PPN Models

five granules. (In both models, deposit branching out of transition **RequestT** is used to determine which granules a transaction will require. The branching probabilities are set to reflect the transaction characteristics outlined above.)

The speedup curves for the aforementioned PPNs are given in Figure 8. We define speedup for a Persephone simulation using p processors to be the execution time for Persephone on one processor divided by the execution time for Persephone on p processors. Note that the CentralServer and CommChannel models have fewer than sixteen data points because they have fewer than sixteen LPs.

The linear speedup exhibited by Pipe would not ordinarily be surprising, since the net is largely feed-

Table 1: Conflict Sets and Maximum Speedup of PPN Models

Model	*Conflict Sets*	*Speedup*
Pipe	8	9.8
CommChannel	3	3.8
Multiprocessor	6	3.6
DB One-At-A-Time	6	2.9
DB All-At-Once	2	2.8
CentralServer	5	2.1

forward. The presence of the control channels, however, introduces a large amount of feedback into the topology of the simulation model, which makes the linear speedup somewhat unexpected. We hypothesize that the feedback due to the control channels is what prevents the speedup from being perfectly linear, i.e., of slope one.

The remaining models each exhibit speedups flattening out at between four and eight processors and, not coincidentally, contain large amounts of feedback in the PPN itself. There are several factors limiting these speedups. First, a model may have poor speedup potential regardless of the degree of concurrency in the system being modeled [Wagner 89]. As an example, Wagner shows that the queueing network model from which CentralServer is derived has a theoretical limit of 3.67 on its achievable speedup despite the fact that the queueing network simulation contains five LPs. Thus, for some models the limit on speedups is intrinsic to the problem and cannot be fixed by tuning.

Second, there are theoretical limits on the performance of conservative parallel simulation that are related to the topology of the simulation model [Lin 90]. Lin shows, for example, that under certain conditions the speedup of a conservative parallel simulation cannot exceed the number of strongly connected components in the simulation model. In the TFP simulation model used by Persephone, the strongly connected components are precisely the static conflict sets of the PPN. Table 1 gives the number of static conflict sets in each of the models and the maximum speedup achieved by Persephone on the model. Since Persephone does not meet Lin's criteria exactly, in some cases it does achieve speedup greater than the number of strongly connected components, but overall these results can be viewed as experimental evidence that supports Lin's work.

We hypothesize that another factor limiting speedup is serialization due to critical sections in the underlying software systems on which Persephone is

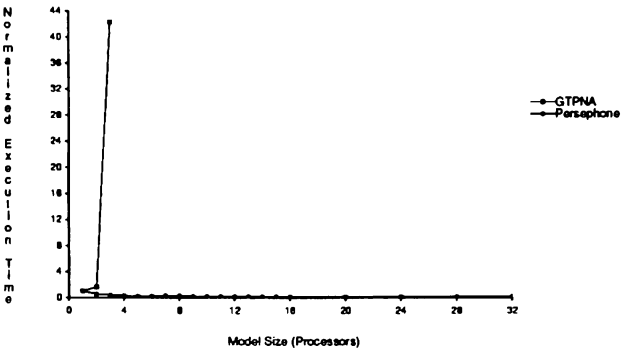

Figure 9: Normalized Execution Time of Persephone and GTPNA on Multiprocessor

built. For example, Synapse's scheduling of LPs on processors, which happens in concert with PRESTO, requires a queue of ready LPs. Access to this queue must be serialized to ensure correct execution. While we have been unable to verify directly (because of lack of appropriate measurement tools) that contention for this queue is limiting speedup, this hypothesis is consistent with our experience with other applications using the PRESTO system.

6.3 Execution Time Versus Model Size

For these measurements we compare the execution times of Persephone and an analytic solver, GTPNA [Holliday & Vernon 86], on the Multiprocessor PPN model as the number of processors in the model varies from 1 to 32 (achieved by varying the initial marking of place `BusyProcessors`). All measurements are of Persephone running on eight physical processors of a Sequent Symmetry and GTPNA running on a DEC VAXstation 3500 with 16 Mbytes of memory. The execution times from the different systems cannot meaningfully be compared directly, so the metric we use is normalized execution time—the time for model size s divided by the time for model size 1.

The normalized execution times for Persephone and GTPNA on the Multiprocessor PPN model are plotted in Figure 9. For each Persephone run, we used the sequential stopping procedure of Law and Carson (as described in [Law & Kelton 82]) to terminate the simulation when it reached a 99% confidence interval of relative precision 0.2 for an estimate of the average time spent by a token at place `BusFree`.

The exponential time complexity from which the analytic techniques all suffer is evident in the curve for GTPNA: GTPNA requires over 42 times as long to evaluate the 3-processor model as it does the 1-processor model, and it is infeasible to use it on larger

Table 2: Execution Time (Seconds) of Persephone and Seq on DB Models

Model	Persephone	Seq
DB All-At-Once	61.5	658.2
DB One-At-A-Time	104.4	129.6

models. (We observed runs of the 4-processor model that consumed more than 150 CPU hours without finishing.) The execution time of Persephone, on the other hand, is nearly constant across the entire range of model sizes. (Actually, as explained in [Thomas 91], the execution time *decreases* as model size increases.) Although we cannot claim, based on this single example, that simulation will always dominate the analytic methods so convincingly, we are encouraged by these results because they demonstrate that such domination is at least possible.

One of the advantages Persephone has over GTPNA is its decentralized conflict resolution scheme, presented in Section 5, which allows it to avoid computing maximal sets. Note, however, that *any* PPN evaluation technique that resolves conflicts in the standard centralized way is at a similar disadvantage. To illustrate this concretely, we constructed a sequential PPN simulator (call it "Seq") that provides a subset of the functionality of Persephone but that resolves conflicts by computing maximal sets. Seq is written in C++ and runs on the same machine as Persephone, so its execution times are directly comparable. Table 2 lists the execution time in seconds for Persephone (running on a single processor) and Seq to simulate two of the PPN models. Persephone running on one processor is *faster* than Seq, a simulator that provides *less* functionality, and the gap in execution times widens as the sizes of the conflict sets grow.

7 SUMMARY

We have examined the use of parallel simulation for the analysis of Performance Petri nets. Because the actions of the "physical processes" of PPNs require global information, these networks present new challenges for parallel simulation methodologies, which have assumed that physical processes interact only through the explicit exchange of messages.

We have developed a parallel simulation protocol for PPNs, the Transition Firing Protocol, that copes with the global nature of their actions. This protocol is based on the conservative approach to parallel simulation. We have introduced a new technique to conservative simulation, Selective Receive, that relaxes the traditional message receipt rules by allowing a logical process to sometimes ignore specific input channels when attempting to receive messages. Using information about the PPN simulation known statically, we use Selective Receive to allow receipt of messages that would normally have to remain pending, leading quickly to deadlock of the simulation.

We have also introduced a new conflict resolution procedure for PPNs that was inspired by the parallel simulation approach. This conflict resolution procedure has been observed to be much faster in practice than the exponential procedures used previously.

Finally, we have created a prototype implementation of our parallel PPN simulator and measured its performance. Its speedup characteristics are similar to those obtained in other applications of parallel simulation, despite the apparently sequential flavor of PPNs induced by their global decision making procedures. More importantly, we have demonstrated that simulation can in fact be used to evaluate PPN models which are too large for the analytic PPN evaluation techniques, thereby extending the applicability of PPN modeling to larger systems.

ACKNOWLEDGEMENTS

Dave Wagner answered numerous questions regarding simulation issues in general and Synapse in particular. Mary Vernon made GTPNA available to us. Tom Anderson assisted with the ideas behind the decentralized approach to conflict resolution and offered analytic insight into its relationship to the standard centralized approach. Derek Eager provided insightful comments on an earlier draft of this paper. We are grateful to them for this assistance.

This material is based upon work supported by the National Science Foundation (Grants DCR-8352098, CCR-8619663, and CCR-8703049), the Washington Technology Center, and Digital Equipment Corporation (the External Research Program and the Systems Research Center).

REFERENCES

[Ajmone Marsan & Chiola 87] M. Ajmone Marsan and G. Chiola. On Petri nets with deterministic and exponentially distributed firing times. In [Rozenberg 87], pages 132–145.

[Bershad et al. 88] B. N. Bershad, E. D. Lazowska, H. M. Levy, and D. B. Wagner. An open environment for building parallel programming systems. In *Symposium on Parallel Programming: Experience With Applications, Languages, and Systems*,

pages 1–9, July 1988.

[Chandy & Misra 81] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(4):198–206, April 1981.

[Fujimoto 90] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):31–53, October 1990.

[Greenberg & Lubachevsky 90] A. G. Greenberg and B. D. Lubachevsky. Unboundedly parallel simulations via recurrence relations. In *Proceedings of the 1990 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 1–12, May 1990.

[Hartung 88] G. Hartung. Programming a closely coupled multiprocessor system with high level Petri nets. In [Rozenberg 88], pages 154–174.

[Holliday & Vernon 86] M. A. Holliday and M. K. Vernon. The GTPN analyzer: Numerical methods and user interface. Technical Report 639, Computer Sciences Department, University of Wisconsin at Madison, Madison, WI, April 1986.

[Holliday & Vernon 87] M. A. Holliday and M. K. Vernon. A generalized timed Petri net model for performance analysis. *IEEE Transactions on Software Engineering*, SE-13(12):1297–1310, December 1987.

[Jefferson 85] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[Law & Kelton 82] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, New York, NY, 1982.

[Lin 90] Y.-B. Lin. *Understanding the Limits of Optimistic and Conservative Parallel Simulation*. PhD dissertation, University of Washington, Seattle, WA, 1990. Available as Department of Computer Science and Engineering Technical Report 90-08-02.

[Misra 86] J. Misra. Distributed discrete-event simulation. *ACM Computing Surveys*, 18(1):39–65, March 1986.

[Murata 89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[Nicol 88] D. M. Nicol. Parallel discrete-event simulation of FCFS stochastic queueing networks. *SIGPLAN Notices*, 23(9):124–137, September 1988.

[Reed et al. 88] D. A. Reed, A. D. Malony, and B. D. McCredie. Parallel discrete event simulation using shared memory. *IEEE Transactions on Software Engineering*, 14(4):541–553, April 1988.

[Rozenberg 87] G. Rozenberg, editor. *Advances in Petri Nets 1987*, volume 266 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, NY, 1987.

[Rozenberg 88] G. Rozenberg, editor. *Advances in Petri Nets 1988*, volume 340 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, NY, 1988.

[Stroustrup 86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1986.

[Taubner 88] D. Taubner. On the implementation of Petri nets. In [Rozenberg 88], pages 418–439.

[Thomas 91] G. S. Thomas. Parallel simulation of Petri nets. Master's thesis, University of Washington, Seattle, WA, 1991. Available as Department of Computer Science and Engineering Technical Report 91-05-05.

[Wagner & Lazowska 89] D. B. Wagner and E. D. Lazowska. Parallel simulation of queueing networks: Limitations and potentials. In *Proceedings of the 1989 SIGMETRICS Conference and Performance '89*, pages 146–155, May 1989.

[Wagner 89] D. B. Wagner. *Conservative Parallel Discrete-Event Simulation: Principles and Practice*. PhD dissertation, University of Washington, Seattle, WA, 1989. Available as Department of Computer Science and Engineering Technical Report 89-09-03.

AUTHOR BIOGRAPHIES

GREGORY S. THOMAS received a dual B.A. in computer science and statistics from the University of California at Berkeley in 1988, and completed an M.S. in computer science at the University of Washington in 1991. He is currently a Research Scientist at Battelle Pacific Northwest Laboratories.

JOHN ZAHORJAN is a Professor of Computer Science and Engineering at the University of Washington. His research interests are in computer systems, including performance modeling as well as experimental evaluations, and focus on parallel systems: scheduling of general purpose and supercomputer parallel machines, and modeling techniques for evaluation of parallel systems. In 1984, Dr. Zahorjan received the Presidential Young Investigator Award from the National Science Foundation. He has served as an officer of ACM SIGMETRICS, and was Program Chairman of the 1991 SIGMETRICS Conference.