Parallel Simulation of Performance Petri Nets: Extending the Domain of Parallel Simulation*

Gregory S. Thomas
John Zahorjan

Department of Computer Science and Engineering
University of Washington
Seattle, WA 91895

April 4, 1995

Abstract

We present a parallel simulation protocol for Performance Petri nets, Petri 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 Petri nets 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 Petri net models being a particular example. Second, Performance Petri nets are currently analyzed using numerical techniques that have time and space requirements exponential in the size of the Petri net. Simulation, and particularly parallel simulation, is thus a practical alternate analysis method for these models.

Our new protocol is derived from the rules of conservative parallel simulation. While this approach normally relies on use of the actual or "future" timestamps on the message paths into a component of the model to determine the latest simulation time to which that component can safely progress, we have introduced a new technique that loosens this restriction. Our technique, called *Selective Receive*, allows model components to sometimes ignore certain of their input channels and thus to determine their local clock times based on only a subset of their potential inputs. This technique is helpful in simulating Petri net models. We believe that it may be of use in speeding up the parallel simulation of other systems as well.

Finally, the development of a parallel simulation protocol for Petri nets suggested a small modification to their definition that, while not affecting the expressive power of the nets, allows for much more efficient simulations. We show that parallel simulations using this modification can achieve unboundedly high speedups relative to a sequential simulation that does not employ this approach.

Key Words: Parallel discrete event simulation, conservative and optimistic approaches, message-passing protocols, Performance Petri nets, conflict resolution, system modeling.

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

*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).

Authors' addresses: Department of Computer Science and Engineering, FR-35, University of Washington, Seattle, WA 98195; gthomas@cs.washington.edu, zahorjan@cs.washington.edu.

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 (often called 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. This assumption is valid for many systems. For example, the actions of a server in a queueing network depend solely on its service discipline and on the jobs enqueued at it, with servers interacting only by sending jobs from one to another.

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* that mimics the behavior of the physical processes. The simulation is parallelizable because the logical processes change their state based only on local information and messages received from other logical processes. Further, 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, 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. Instead, a new parallel simulation paradigm is required.

The first goal of this paper, therefore, is to present a parallel simulation protocol, which we call the *Transition Firing Protocol* (TFP), 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 TFP 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 techniques, is also addressed through measurement, using the GTPNA [Holliday & Vernon 86] analytic PPN package as the basis of comparison.

The remainder of this paper is organized as follows. Section 2 provides a brief review of Performance Petri nets and discusses various approaches to applying parallel simulation to them. The Transition Firing Protocol is presented in Section 3. This protocol exploits a fundamental modification to the standard algorithm for conservative parallel simulation. This modification, which we call *Selective Receive*, is described in Section 4. In Section 5 we discuss one aspect of PPN analysis, conflict resolution, for which a slight redefinition of the model semantics suggested by the parallel approach leads to greatly improved analysis performance. Section 6 discusses an implementation of TFP and its performance. Section 7 concludes by summarizing the contributions made by our work.

2 Performance Petri Nets and Parallel Simulation

In this section we briefly introduce PPNs, describe features of the class of nets we use, 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. For classical (i.e., untimed) Petri nets, [Peterson 81] is a standard reference.

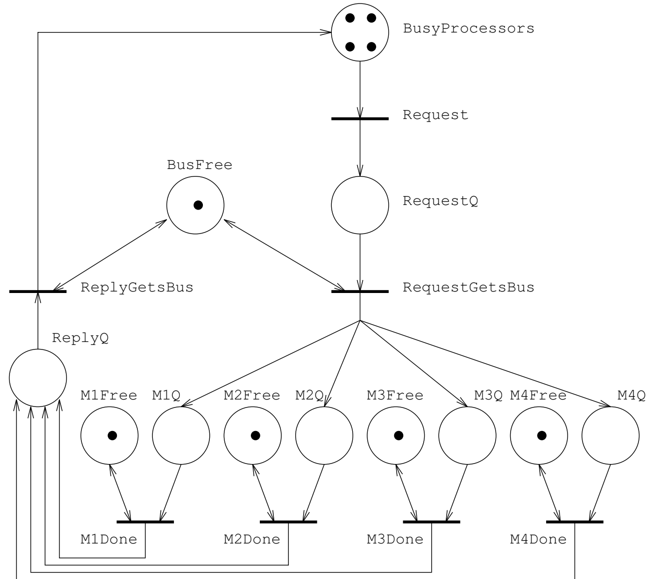

Figure 1: An example Petri net multiprocessor model.

2.1 Performance Petri Nets

Figure 1 gives a graphical representation of a PPN model of a bus-based multiprocessor computer system. The system consists of four processors and four memory modules connected by a single bus. This model might be used, for instance, to evaluate the effect of bus contention on effective processor speed as a function of the number of processors in the system. (We use this model here only to illustrate generic concepts of PPNs. A more detailed description of the model is deferred until Section 6.)

We view a PPN as a bipartite directed graph in which the two types of nodes are *places*, drawn as circles, and *transitions*, drawn as bars. There are twelve places and seven transitions in Figure 1.

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*.) In Figure 1, transition **ReplyGetsBus** has input places **ReplyQ** and **BusFree** and output places **BusyProcessors** and **BusFree**.

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. The marking shown in Figure 1 is (4, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0).

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. The sequence of transition firings that occurs describes the dynamic behavior of the net.

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). In the marking shown in Figure 1, only transition **Request** is enabled.

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 $p_i R p_j$ iff $\mathrm{OT}(p_i) \cap \mathrm{OT}(p_j) \neq \emptyset$. 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*. It is possible for multiple transitions in a static conflict set to become enabled at the same time, but for only some subset of them to begin firing at that time (due to the limited number of tokens available in the *decision* places—places with more than one output transition—of the static conflict set). In Figure 1, places `ReplyQ`, `BusFree`, and `RequestQ` and transitions `RequestGetsBus` and `ReplyGetsBus` form 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. (There are no dynamic conflict sets in the current marking of Figure 1.)

When a dynamic conflict set exists, a subset of its transitions must be chosen to fire. This is called *conflict resolution*. A number of different proposals have been made for selecting the subset to fire. We use a technique modeled on that proposed for Generalized Timed Petri Nets [Holliday & Vernon 87], which uses the notion of *maximal sets*. A maximal set is 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. In general, there may be many distinct maximal sets corresponding to a particular dynamic conflict set.

Under our conflict resolution scheme, the definition of the PPN is extended to include weights on all arcs from places to transitions. (These weights are not shown in Figure 1.) The *relative weight* of a particular maximal set is the product of the arc weights of all arcs from input places to those transitions that are chosen to fire in that maximal set. The maximal set that actually fires is then chosen at random in proportion to these relative weights.

Other extensions to PPNs are made to increase their modeling capability or their ease of use:

Arc Multiplicity For place p and transition t, an arc $(p,\ t)$ of multiplicity m means that m tokens are required at p to enable t. An arc $(t,\ p)$ of multiplicity m means that each time t finishes firing it deposits m tokens at p.

Deposit Branching A probability distribution may be associated with a subset S of the output places of a transition. When the transition fires, the probabilities are used for *deposit branching*, whereby a single output place from S is selected for a deposit. (Output places not in S receive deposits at each firing, as usual.) As an example, each time `RequestGetsBus` fires in Figure 1, it deposits a token at one of $S = \{$`M1Q`, `M2Q`, `M3Q`, `M4Q`$\}$, which is selected at random, and at `BusFree`.

Inhibitor Arcs A generalized inhibitor arc $(p,\ t)$ of multiplicity m inhibits t from firing if there are m or more tokens at p.

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 *logical processes* (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 maximizes the potential parallelism of the simulation, and, since the number of LPs scales directly with the number of PPN nodes, has the potential to address very large problems running on massively parallel machines. However, this decomposition has two drawbacks. First, because the amount of work each LP has to do "per message" (e.g., updating the marking of a place when tokens are deposited) is very small, the relative cost of message passing overhead is very large, leading to potentially poor performance. Second, neither the standard conservative nor optimistic method of parallel simulation can be used for this decomposition of PPNs.

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. As illustrated in Figure 1, if (decision) place `BusFree` has a token, it cannot on its own decide whether to send it to transition `RequestGetsBus` or transition `ReplyGetsBus` since it cannot determine whether either is enabled.

Note that this problem exists for optimistic as well as conservative parallel simulation. One might imagine optimistically sending the token both directions in the case above, and relying on the rollback feature of optimistic simulation to later undo the incorrect decision. However, this will not work. Under optimistic simulation the rollback mechanism is triggered by the arrival of a message timestamped in an LP's past. No such message will arrive in the case of PPNs. Further, even if messages conveying "I could not use the token you sent me at time T" could be sent, this would not solve the problem. If no such messages were sent from any of the receiving LPs, what would this mean? At most one of them should be allowed to ultimately consume the single token that existed, but there is no way to coordinate their actions so that this happens.

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. (Such arcs will always be directed from a transition to a place.) This mapping has characteristics that are just the opposite of the node-based mapping discussed above. In particular, because each LP represents a potentially large number of nodes, the number of events that occur internally to an LP per inter-LP message can be high, minimizing the relative cost of communication overhead. Also, this conflict set-based decomposition of the net allows the standard methods of parallel simulation to be used, since there is no interaction among conflict sets other than the depositing of tokens as a result of transition firings. On the other hand, the potential parallelism of this decomposition is poor, as the number of LPs scales with the number of conflict sets rather than with the number of nodes, and there is no direct relationship between the size of a PPN and the number of conflict sets it contains. (The entire net, or nearly the entire net, can be a single conflict set, for instance. In fact, one of the example PPNs used in Section 6 has two conflict sets, one containing two nodes and the other containing thirty-five nodes.)

In the end, the most appropriate decomposition of a PPN into a network of LPs is probably some combination of these two approaches. In those cases where a conflict set-based decomposition yields parallelism commensurate with the number of physical processors available to run the simulation, its low overhead makes it an attractive alternative. However, for PPNs with only a very few conflict sets or in situations where the simulation will be run on a massively parallel machine, the node-based decomposition described above must be adopted for at least part of the network.

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. It should be kept in mind, though, that while for simplicity of exposition we assume that the entire PPN is decomposed as an LP per node, in a practical situation a more sophisticated combination of node-based and conflict set-based decompositions is probably the best choice.

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. The basic reason for this is the behavior of decision places in the net. As explained in the remainder of this section,

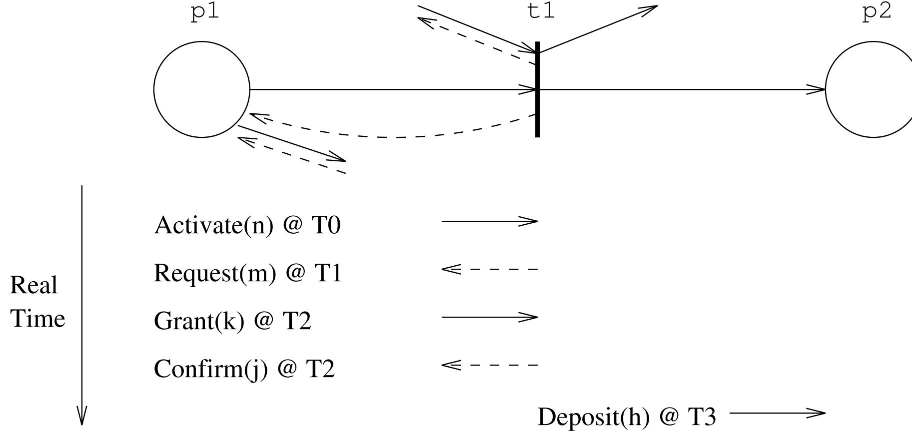

Figure 2: Overview of TFP.

it is the problems caused by decisions places that render invalid standard conservative parallel simulation.

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) when appropriate. The net fragment in Figure 2 contains place LPs `p1` and `p2`, transition LP `t1`, ordinary channels (`p1`, `t1`) and (`t1`, `p2`), and control channel (`t1`, `p1`) (as well as additional channels connecting with other, unspecified LPs).

TFP is basically a double handshake between transitions and their input places, as indicated in Figure 2. The notation *Message*(n)@Ti means *Message*, passing parameter n, is sent at simulation time Ti. Ti is the *timestamp* of *Message*; every message has a timestamp.

TFP's double handshake is motivated by decision places.[1] When a decision place receives tokens, it may cause many transitions to become enabled simultaneously. Because the decision place cannot know which transitions are now enabled without interacting with them, it cannot make an immediate decision about which transition should be sent tokens. The first of the two handshakes determines which transitions are enabled. A decision about which transitions this decision place would like to fire is then made, and in the first portion of the second handshake those transitions are offered tokens. Because each of these transitions may have other decision places as input places, and under the conflict resolution scheme we use (see Section 5) those places may make different firing decisions, in the final portion of the second handshake the transitions inform their input places whether or not they have actually fired.

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 is provided in the next section. 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.

[1] There are several cases in which TFP can be streamlined. For example, a place with a single noninhibitor output arc can send a *Grant* as soon as it has tokens. Additional special case optimizations are possible to further reduce the number of messages that need to be exchanged.

- *Activate*(n)@T0

  The first handshake starts when place `p1` receives a deposit of n tokens at time T0,[2] 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 $\mathrm{T1} \geq \mathrm{T0}$, 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. `t1` thereby asserts that it is ready to fire m times at time T1 if all of its input places (still) contain sufficient tokens at that time. This completes the first handshake.

- *Grant*(k)@T2

  `p1` waits to receive a *Request* from each output transition it has sent an *Activate*.[3] Let $\mathrm{T2} \leq \mathrm{T1}$ be the earliest timestamp among these *Request*s. (In Figure 2, $\mathrm{T2} = \mathrm{T1}$.) When `p1` has received all *Request*s timestamped T2, it executes a conflict resolution algorithm (explained in Section 5) to choose among competing earliest *Request*s, and then replies to each such *Request* 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 *Grant*s. (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 (and so has consumed a corresponding number of tokens from each input place).

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

The receipt of a *Confirm*(j) from `t1` by `p1` completes the second handshake between `p1` and `t1`. `p1` subtracts the j tokens from its marking that were consumed by the firing of `t1` and begins the next cycle of the protocol.

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 each of 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

[2] In general, a deposit is not needed to initiate the sequence—tokens already residing at `p1`, perhaps in the initial marking, can initiate it. We present it this way because it is the easiest situation to understand.

[3] This is why our approach is conservative.

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. For instance, imagine that a place has no tokens in its current marking and so has not sent out any *Activate*s. At this point it would like to receive any *Deposit* messages that its input transitions might send it. However, the semantics of the message receive operation under conservative simulation prevents it from doing so, even if there is a pending *Deposit* message from each of its input transitions, because its message acceptance horizon cannot proceed beyond the time of the last TFP cycle (which is the channel clock value on the channels from its output transitions).

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 *Request*s, and after it receives a *Grant* from LP_i until it sends *Confirm*s.

To better understand the use of Selective Receive in TFP, and the integration of *Deposit* messages into TFP, we now summarize the activity of the place and transition LPs. For simplicity, we again assume regular arcs of multiplicity one.

4.1 Place LP

An LP that represents a place may be thought of as a finite automaton. A state diagram for this automaton appears in Figure 3. In the discussion below, MRT is the *minimum Request time*—the timestamp of the earliest *Request* received.

A place LP, P, starts out in state 0, a transient boot state. It sends *Activate*s to its output transitions as warranted by its initial marking, then goes to state 1.

While in state 1, P collects *Request* responses to previously sent *Activate*s and accepts new *Deposit*s,

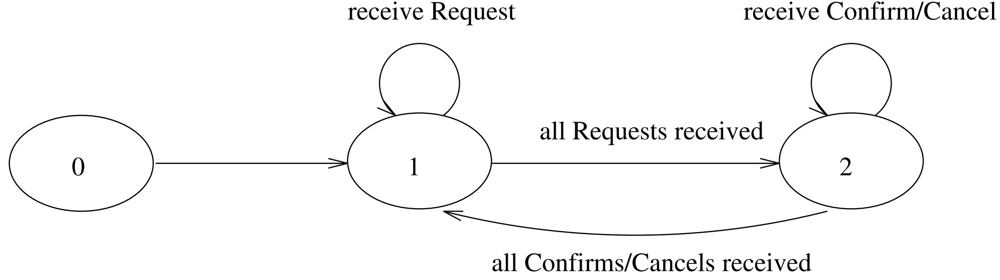

Figure 3: State diagram for place LP.

sending out any new *Activates* that they may allow. To do this, P uses Selective Receive, allowing input only on channels from its input transitions (messages received will be *Deposits*) and from any *Activated* output transitions—channels from non-*Activated* output transitions are ignored. If such a channel from an output transition T is not ignored, the progress of P is impeded because no messages can arrive along this channel until after P sends T an *Activate* and so the message acceptance horizon cannot advance.

P remains in state 1 until all earliest timestamped *Requests* are received and it knows its marking at MRT.[4] It then applies a conflict resolution scheme (explained in Section 5) to decide which of the earliest *Requests* to grant, and sends out an appropriate *Grant* in response to each earliest *Request*. P now moves to state 2, where it receives information about which transitions eventually do fire. Note that *Requests* timestamped later than MRT remain pending.

P remains in state 2 until it receives a *Confirm* in response to each $Grant(k)$, $k > 0$, it sent. (The response to a $Grant(0)$ will be a $Confirm(0)$, and thus will not affect the marking of P, so P need not wait for responses to $Grant(0)$ messages before returning to state 1.[5]) P ignores the input channel from output transition T after P receives a *Confirm* from T. Once again, if the channel is not ignored, the progress of P is impeded because no further message can arrive from T until after P sends an *Activate* to T.

When all required *Confirms* are received, P updates its marking, subtracting tokens for each transition that fired. If tokens remain after this update, P sends an *Activate* to each inactive output transition if allowed by the updated marking. Finally, P returns to state 1.

4.2 Transition LP

A state diagram for the transition LP automaton appears in Figure 4. In the discussion below, MAT is the *maximum Activate time*—the timestamp of the latest *Activate* received.

A transition LP, T, starts out in state 0 and remains there until it has received an $Activate(n_i)$ from each input place P_i. It then sends a $Request(\min_i n_i)$@MAT to each input place and goes to state 1. Between the time T receives an *Activate* from P_i and the time it sends *Requests*, T ignores P_i because P_i cannot send another message to T until after T sends a *Request* to P_i.

T remains in state 1 until it receives a $Grant(j_i)$@MAT from each input place P_i. It then sends a $Confirm(\min_i j_i)$@MAT to each input place, possibly schedules one or more *Deposits* to be sent to its output places after suitable firing delays have elapsed, and returns to state 0. Between the time T receives a *Grant* from P_i and when T sends *Confirms*, T ignores P_i because P_i cannot send another message to T until after T sends a *Confirm* to P_i.

[4] This involves sophisticated *lookahead* functions. Details are given in [Thomas 91].

[5] Transitions that were sent $Grant(0)$ messages are of secondary importance at this point in the protocol, and failure to receive *Confirms* from them will not prevent P from returning to state 1. Thus, a $Confirm(0)$ from some LP T may be received while in state 1. When this happens, an *Activate* is immediately sent to T if allowed by the current marking.

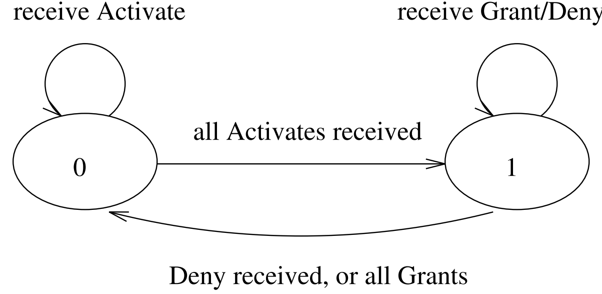

Figure 4: State diagram for transition LP.

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. As mentioned in Section 2.1, a number of proposals have been made for this decision procedure, and there is no standard agreement on this aspect of PPNs. However, one of the more flexible approaches is 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. (Recall that a maximal set is a firable set of transitions that is not a subset of any other firable set.) 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: after an event is processed the new PPN marking is examined, the maximal sets are enumerated and assigned probabilities, one is selected, and a new set of (transition firing) events are scheduled. This is a relatively straightforward procedure 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. Since in a parallel simulation each LP may have a different simulation time, determining the global state at a particular simulation time is not a simple matter. 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. Needless to say, this is a complicated protocol to implement, and tends to serialize execution of the simulation. (Note, however, that with such an implementation the *Confirm* messages of TFP could be eliminated since it would be guaranteed that the firing decisions of all decision places would be in agreement.)

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.[6]

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.

In the next section, which concentrates on the speedup characteristics of our TFP-based parallel simulation, we also provide some empirical information about the performance advantages of decentralized over maximal set-based conflict resolution.

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. For these measurements we use Persephone [Thomas 91], our prototype implementation of TFP.

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), yielding a series of models. We compare the elapsed time to evaluate each model using Persephone and GTPNA [Holliday & Vernon 86], which employs analytic techniques to evaluate PPNs. Our purpose here is to determine whether or not simulation can extend the applicability of PPN modeling to larger systems by facilitating the evaluation of systems larger than those amenable to analytic techniques.

6.1 Persephone

Persephone 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 parallel simulation. Synapse exports a basic LP class that provides the message delivery mechanism and ensures that messages are received in nondecreasing timestamp order, and a scheduler that handles the scheduling of LP objects. 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

[6] 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. Thus, while the decentralized approach has better average time characteristics in our experience, it is easy to generate artificial models for which the expected conflict resolution time exceeds any bound! This rather disconcerting aspect of this approach is the topic of current work.

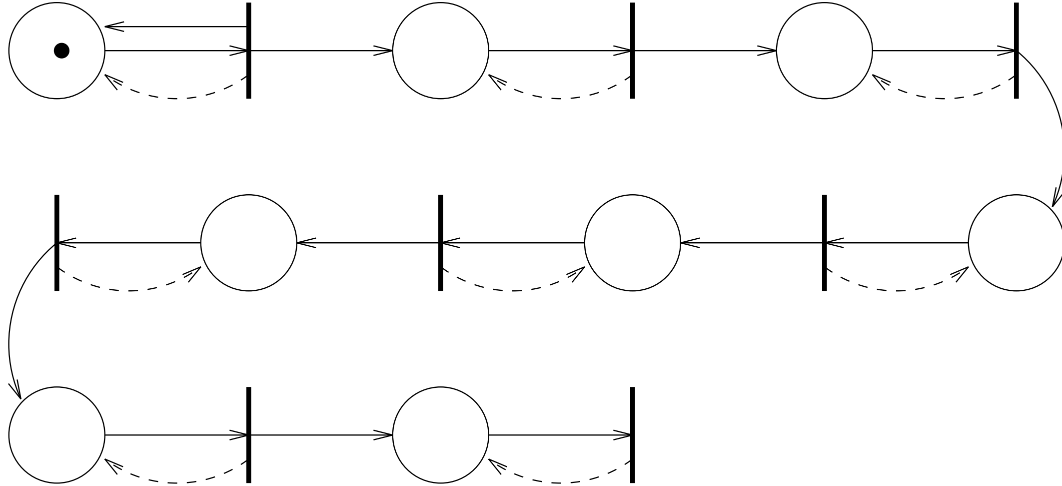

Figure 5: Instruction pipeline net.

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 and is the focus of continuing work.[7]

6.2 Exploiting Available Processors: Speedup Analysis

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.

Figure 5 shows our first PPN model, called Pipe, and its initial marking. 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. All transition firing times are deterministic with duration 1.0, reflecting the deterministic nature of pipeline stages. Control channels are shown explicitly, as dashed arcs, to emphasize the fact that the net is not truly feedforward. This is discussed later in this section.

Our second application is the bus-based multiprocessor computer system introduced in Section 2.1. The PPN model for this system, called Multiprocessor, is given in Figure 1 (in Section 2.1). Once again, the figure gives the initial marking of the PPN model, as do the figures for the remaining PPN models. By increasing the number of tokens in place **BusyProcessors**, Multiprocessor can be used to evaluate how bus contention grows with the number of processors in a multiprocessor. Similarly, changing the number of tokens in **BusFree** could reflect adding additional buses to the machine.

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 applica-

[7] The three systems—Persephone, Synapse, and PRESTO—together comprise 40,000 lines of C++ code.

Figure 6: Central server net.

tion [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 6, each token represents a job in a multiprogramming system, and circulates between the CPU and I/O devices acquiring service. The model may be used, for example, to answer questions about the effect of configuration on job performance.

A window flow controlled communication channel is our fourth application. The PPN model for this application, called CommChannel, is shown in Figure 7. 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 at a time, and for a window size W there can be up to W unacknowledged packets sent. This model might be used to evaluate how the window size (which is related to the number of buffers in the receiver) affects performance.

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

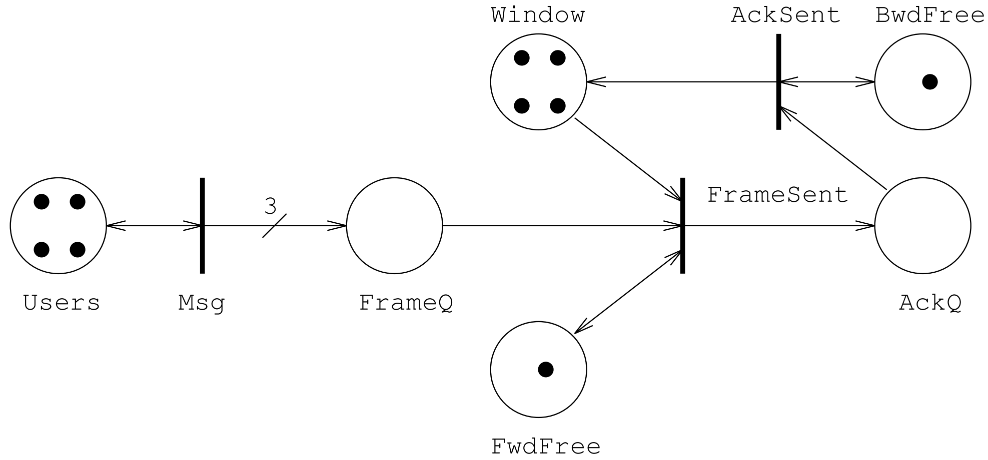

Figure 7: Communication channel net.

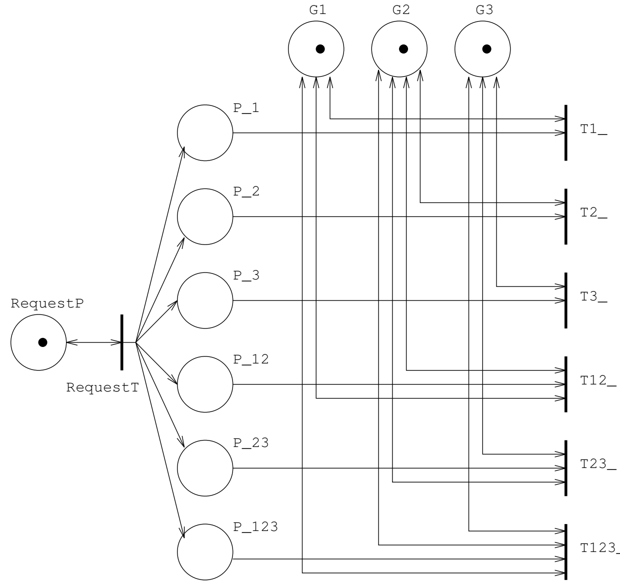

Figure 8: All-At-Once database locking net.

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. (We use ascending granule ID number.)

The PPN models for these schemes are shown in Figures 8 and 9. 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 slightly larger systems, those containing 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 10. 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.

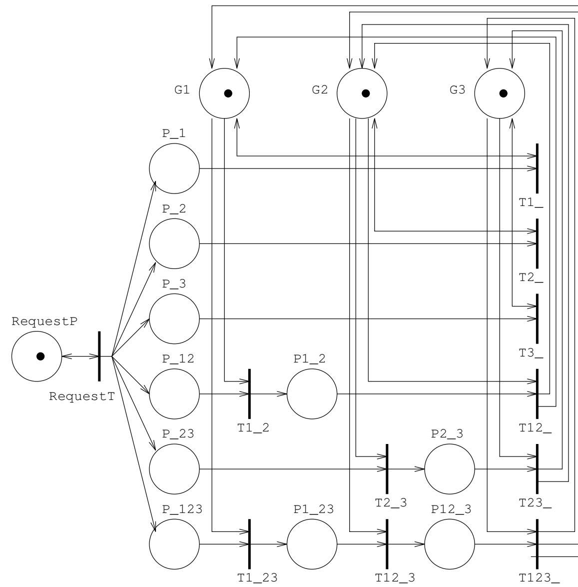

Figure 9: One-At-A-Time database locking net.

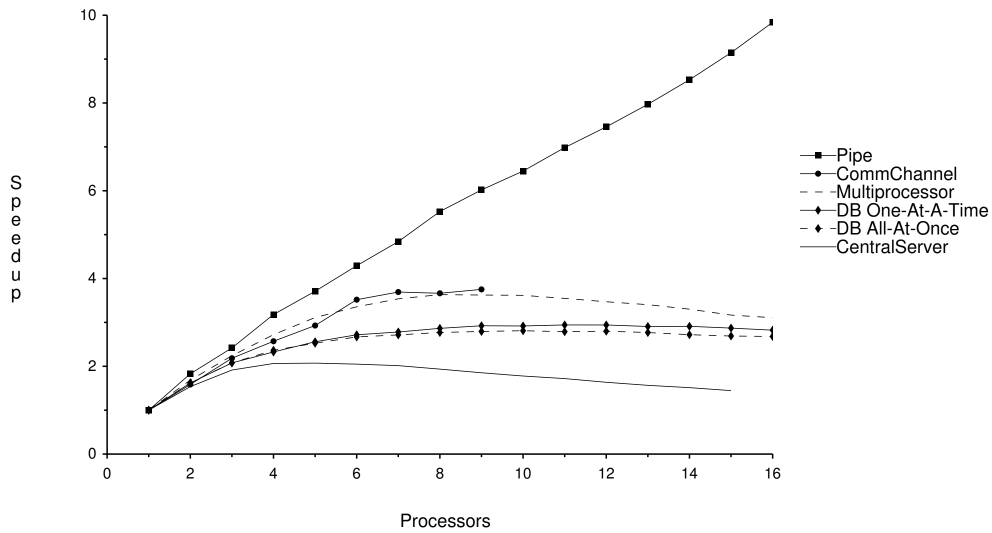

Figure 10: Measured speedups for Persephone on the PPN models.

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

Table 1: Number of conflict sets and maximum speedups of PPN models.

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

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 confirms 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 built, i.e., Synapse and PRESTO. 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. If our hypothesis is correct, this limit on speedup could be addressed by tuning of the runtime systems.

6.3 Execution Time Versus Model Size

For these measurements we compare the execution times of Persephone and GTPNA on the Multiprocessor PPN model as the number of processors in the model varies from 1 to 32. (Recall that this is achieved simply by varying the initial marking of place `BusyProcessors`.) All measurements are for Persephone running on eight physical processors and GTPNA running on a DEC VAX 8550. 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 11. 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 complexity from which the analytic techniques all suffer is evident in the curve for GTPNA: GTPNA requires over 37 times as long to evaluate the 3-processor model as it does the 1-processor model, and it cannot be used for larger models because it runs out of memory.[8] Persephone, on the other

[8] GTPNA uses a set of statically sized tables. We recompiled the program a number of times to increase these table sizes.

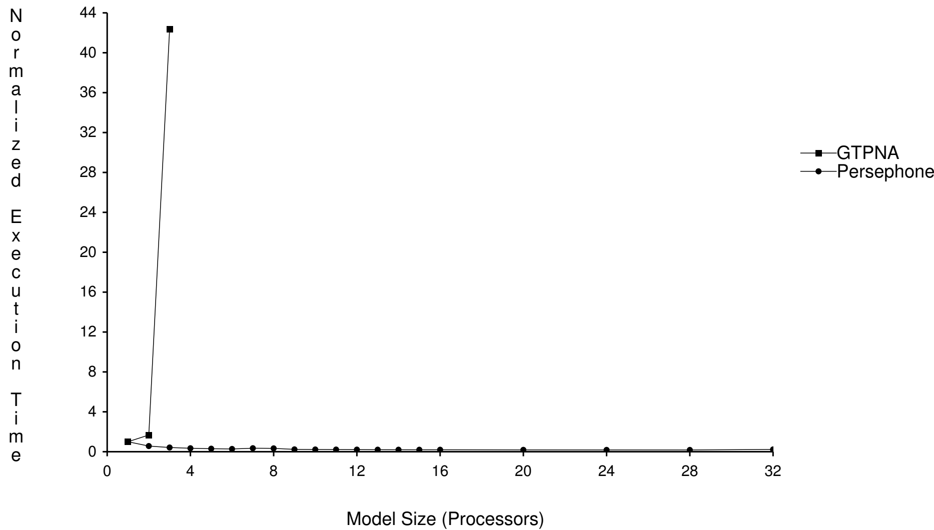

Figure 11: Normalized execution time for Persephone and GTPNA on Multiprocessor.

Table 2: Execution times (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

hand, exhibits very slow linear growth in execution time across the entire range of model sizes. 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.

Beyond a certain limit the run time system refused the program. For table sizes at the limit allowed by the run time system, we observed runs that consumed more than 150 CPU hours without finishing, probably due to enormous paging overheads or numerical convergence problems. We ran our experiments on an otherwise basically idle machine with 96 megabytes of memory.

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 some analytic insight into its relationship to the standard centralized approach. We are grateful to them for this assistance.

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.

[Peterson 81] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[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.